CustomProcessingUnit: Reverse Engineering and Customization of Intel Microcode

Pietro Borrello[1], Catherine Easdon[2,3], Martin Schwarzl[2], Roland Czerny[2], Michael Schwarz[4]

[1] Sapienza University of Rome, [2] Graz University of Technology,
[3] Dynatrace Research, [4] CISPA Helmholtz Center for Information Security

***Abstract*—Microcode provides an abstraction layer over the instruction set to decompose complex instructions into simpler micro-operations that can be more easily implemented in hardware. It is an essential optimization to simplify the design of x86 processors. However, introducing an additional layer of software beneath the instruction set poses security and reliability concerns. The microcode details are confidential to the manufacturers, preventing independent auditing or customization of the microcode. Moreover, microcode patches are signed and encrypted to prevent unauthorized patching and reverse engineering. However, recent research has recovered decrypted microcode and reverse-engineered read/write debug mechanisms on Intel Goldmont (Atom), making analysis and customization of microcode possible on a modern Intel microarchitecture.**

In this work, we present the first framework for static and dynamic analysis of Intel microcode. Building upon prior research, we reverse-engineer Goldmont microcode semantics and reconstruct the patching primitives for microcode customization. For static analysis, we implement a Ghidra processor module for decompilation and analysis of decrypted microcode. For dynamic analysis, we create a UEFI application that can trace and patch microcode to provide complete microcode control on Goldmont systems. Leveraging our framework, we reverse-engineer the confidential Intel microcode update algorithm and perform the first security analysis of its design and implementation. In three further case studies, we illustrate the potential security and performance benefits of microcode customization. We provide the first x86 Pointer Authentication Code (PAC) microcode implementation and its security evaluation, design and implement fast software breakpoints that are more than 1000x faster than standard breakpoints, and present constant-time microcode division, illustrating the potential security and performance benefits of microcode customization.

I. Introduction

Microcode is the hidden software layer between the instruction set and the underlying hardware. In most Complex Instruction Set Architectures (CISC), each instruction, or *macro-instruction*, is translated into one or more *micro-operations* (*μops*) that are executed by the underlying hardware [41]. In total, there are over 2700 distinct μops in Intel x86 [48]. Many simple instructions map to a single μop. However, more complex instructions are essentially entire programs and can map to > 50 μops [43]. Microcode is a crucial optimization for these instructions, as μops are much simpler to implement in hardware and can be pipelined more efficiently [41].

Microcode is notoriously challenging to verify [22]. Independent auditing of microcode, subjecting it to static and dynamic analysis, would supplement manufacturers' verification efforts and help build trust in this hidden software. Moreover, tools enabling such analysis would facilitate research into CPU behavior. Dynamic microcode tracing, for example, would provide the fine-grained microarchitectural control that is so elusive in microarchitectural attack research [30] and in CPU fuzzing for undocumented behavior [24], [25], [29], [51] or hardware defects [54]. Furthermore, the ability to modify microcode would enable research into microcoded security mechanisms such as microcode-assisted address sanitization and customizable `rdtsc` precision [52].

Unfortunately, x86 microcode is confidential. Intel partially documented the μop sequences used for instructions in the Pentium Pro [42], but only μop counts have been published for instructions on newer CPUs, making reverse engineering research necessary [1]. Microcode has been reverse-engineered to enable customization on AMD Opteron [53] and Intel P6 (Pentium Pro to Pentium III) [16]. However, newer x86 CPUs have much stronger cryptographic protection for microcode updates. The updates are encrypted and signed to prevent unauthorized patching or reverse engineering, and the decrypted microcode never leaves the internal buffers of the CPU. Beyond the protection of intellectual property, security is a powerful motivation for this cryptographic protection. Prior work has explored security concerns around microcode, such as the potential for backdoors [27], [68]. If the update mechanism were compromised, an attacker could maliciously patch microcode, for example, to introduce a backdoor to reveal cryptographic keys to JavaScript in the browser [68]. Reverse engineering research must, therefore, carefully balance the potential security benefits against the potential risks.

In recent years, microcode research has seen a considerable evolution thanks to the work of Ermolov et al. [32], [34], [35]. They achieved Red-Unlock on Goldmont and Goldmont Plus CPUs, a CPU mode that enables JTAG debugging of internal CPU components using external hardware [32]. Furthermore, they identified two undocumented instructions accessible on Red-Unlocked CPUs, `udbgrd` and `udbgwr`, that enable read/write access to internal microarchitectural components from software [35].

Motivated by this crucial breakthrough, in this paper we build upon their work, posing the following research questions:

1) What are the semantics of microcode in modern Intel CPUs?
2) Is the microcode update process secure?

3) Could microcode customization bring security or performance benefits?

To answer these questions, we reverse-engineer microcode semantics and reconstruct microcode patching capabilities. We develop the first decompiler for Goldmont microcode to analyze how the CPU interacts with its internal components during microcode updates. We leverage the undocumented instructions to mimic these interactions to create microcode read and write primitives. Building upon these, we design and implement CustomProcessingUnit: the first framework for static and dynamic analysis of Intel microcode, supporting the Goldmont microarchitecture. Our framework can assemble microcode patches, install these in the CPU and then trace microcode execution in real-time, enabling CPU debugging at the µop level without additional hardware.

We leverage our framework to reverse-engineer the confidential Intel microcode update algorithm and analyze it in depth. For the first time, we perform a public and independent security analysis of the design and implementation of the update algorithm on Goldmont, evaluating its attack surface and possible security holes. Our analysis reveals a minor weakness in the implementation: the update is stored in the L2 cache during decryption, which is potentially exploitable, although we did not succeed in doing so. Such analysis is a first step toward independent auditing of microcode to verify manufacturers' security claims.

Moreover, in three additional case studies, we explore the benefits of CPU customization at the microcode level for performance and security. First, we bring Pointer Authentication Codes [5] to x86 for the first time, presenting a fast microcode implementation of pointer signing and verification in ~25 clock cycles. We thus show how we can enhance x86 CPUs with fundamental security concepts from other architectures. We evaluate the security of our x86 PAC implementation by reproducing the PACMAN attack [65] for the first time on x86. Thanks to our framework, we deepen the analysis by investigating alternative PAC implementations that mitigate such an attack at the microcode level, providing the first public PAC implementation not vulnerable to PACMAN. Second, we design and implement fast software breakpoints, which we call *µsoftware breakpoints*, to execute the breakpoint handlers directly in microcode. This provides a speedup of ~1000x over `int3` instructions, showcasing how microcode customization can bring performance benefits. Third, we patch the `div` instruction to execute in constant time to prevent timing side-channel attacks [9], bringing a ~1.6x speedup over state-of-the-art constant-time implementations, improving both performance and security.

In brief, we present the following contributions:

1) We introduce the first framework for static and dynamic analysis of Intel Goldmont (GLM) microcode for Atom CPUs, featuring support for microcode tracing and patching to provide complete control.
2) We demonstrate how our framework aids CPU reverse engineering by uncovering the details of the confidential Intel microcode update algorithm.
3) In three further case studies, we illustrate how complete control over microcode can bring security and performance benefits. We implement Pointer Authentication Codes (PAC) for x86, fast software breakpoints, and constant-time hardware division.

With this work, we hope to make microcode research accessible to a broader audience and to help the community improve its understanding of microcode security guarantees. CustomProcessingUnit is open-source at https://github.com/pietroborrello/ghidra-atom-microcode (static analysis module) and https://github.com/pietroborrello/CustomProcessingUnit (dynamic analysis module). We hope that it will facilitate further auditing of Intel microcode and inspire the development of additional tooling for other CPUs.

Outline. We provide relevant technical background in Section II. Section III presents our framework for static and dynamic analysis and describes the reverse engineering we conducted to create it. Sections IV, V, VI, and VII provide case studies of our framework, including reverse engineering of the microcode update algorithm in Section IV. We conclude in Section VIII.

Ethical Considerations. Before deciding to publish our framework, we assessed its malicious potential. Our static analysis functionality requires decrypted microcode and does not compromise Intel's microcode update encryption and integrity validation. Our dynamic analysis functionality requires the CPU to be in Red Unlock mode. The only publicly-known method to achieve this requires exploitation of a patched vulnerability. It is only feasible on Intel's system-on-chip CPUs and, in practice, has been achieved on a small number of devices (see Section II). While Red Unlock may be achieved on more devices in the future, in our assessment, the potential security benefits of making microcode analysis accessible to a broader audience outweigh this risk.

II. Background

In this section, we introduce the relevant technical background required for understanding the rest of the paper. Note that the relevant background for our case studies is covered in their respective sections (IV, V, VI, VII).

1) Microcode Structure: The Instruction Decoding Unit (IDU) is the component responsible for translating CISC instructions to µops. Most Intel CPUs have multiple decoders: several simple decoders to translate x86 instructions that map to a single µop, a complex decoder to translate instructions that map to 1-4 µops, and a *Microcode Sequencer* responsible for translating microcoded instructions [47], [53]. Microcoded instructions are the most complex instructions that require advanced logic to be executed. Examples are `cpuid` that returns detailed information about the CPU and `wrmsr` that modifies internal settings in model-specific registers (MSRs).

The microcode is stored in a dedicated read-only memory inside the CPU (MSROM). Instruction definitions are organized in *triads*, consisting of three µops and a sequence word. Sequence words affect the control flow of the executed triad and can also act as synchronization primitives for the dataflow akin to `lfence` instructions. Goldmont (GLM) CPUs have space for 7936 triads in the MSROM [31].

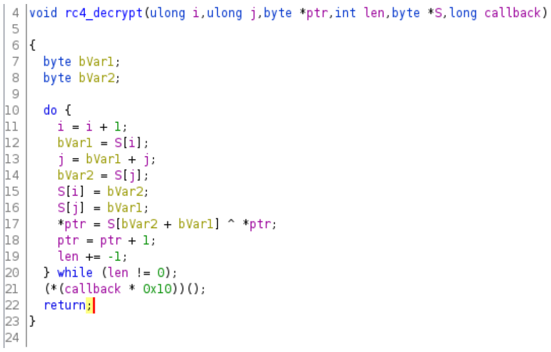
2) Microcode Patches: Modern CPUs allow the microcode to be updated at runtime with microcode patches. This enables patching of bugs in complex instructions [21] and implementation of new features. Thus, the CPU needs a dedicated writable region to hold the patches (MSRAM). On GLM, there is space for 128 triads in the MSRAM [31]. Updates are applied either by the BIOS at boot time or by the operating system. For Intel CPUs, the update routine is triggered by writing the virtual address where the microcode patch has been loaded to the MSR `IA32_BIOS_UPDT_TRIG`. Intel's updates are signed and encrypted [20], [34], [40]. For GLM and GLM Plus, Ermolov et al. documented that the update is RSA signed and RC4 encrypted, providing a decryption algorithm [33].

3) Microcode Hooks: To run patched microcoded instructions, a *microcode hook* redirects control flow from the MSROM to the MSRAM [34], [52]. To implement the hooks, microcode updates set the *match registers* in the CPU to the microcode addresses that need to be redirected. Each time the microcode is executed from the MSROM at an address contained in one of the match registers, the control flow is automatically redirected to the corresponding patch address in the MSRAM. On GLM, there is space for up to 64 hooks [31].

4) Control Register Bus (CRBUS) and Local Data Access Test Port (LDAT): The CRBUS is an internal bus that connects all internal CPU units and exposes core controls and configurations (e.g., control registers and some MSRs are mapped) [35]. Each unit has its own range of addresses on the bus that can be used to query its state and update its configuration. It is used by microcode but is not intended to be architecturally accessible to software. LDAT is a debug interface between local CPU arrays and the CRBUS that facilitates post-silicon validation [55]. Each array provides an LDAT port for read/write access over the CRBUS. Combined, the CRBUS and LDAT provide access to the internal state of CPU core units such as the Microcode Sequencer, Instruction Fetch Unit, caches, and TLB [15].

5) Red Unlock: This special mode provides access to the CRBUS and LDAT via JTAG using a USB debug cable or proprietary hardware [35]. In Intel's threat model, Red Unlock (which they refer to as 'Protection Class Intel') is only possible with Intel's authentication key [45]. However, Ermolov et al. published a proof of concept to Red Unlock GLM by exploiting a (now patched) vulnerability in the Intel Management Engine (ME) [32]. Leveraging Red Unlock, they exported the MSROM and MSRAM contents and reverse-engineered the format of µops, providing a disassembler for GLM microcode. The proof of concept has also been ported to Skylake and Kaby Lake [2]. In contrast to GLM, this only enables Red Unlock on the ME rather than the CPU because only Intel's system-on-chip designs have a shared DFX AGG unit (and, therefore, a shared unlock state) for the chipset and main CPU cores [35].

6) Undocumented Debug Instructions: Ermolov et al. discovered the existence of two undocumented instructions on Intel CPUs, `udbgrd` and `udbgwr` [35]. These instructions are the final puzzle piece for microcode customization. On Red-Unlocked CPUs, they can be used to read and write all of the internal components made accessible by the CRBUS and LDAT from software without any additional hardware. Crucially, this includes the Microcode Sequencer arrays [35]. While only GLM and GLM Plus have been publicly Red-Unlocked, the existence of these instructions on other microarchitectures has been inferred using performance counters [11]. In this paper, we leverage these instructions to create the first analysis framework for observing and modifying CPU microcode.



```
void rc4_decrypt(ulong i,ulong j,byte *ptr,int len,byte *S,long callback)

{
  byte bVar1;
  byte bVar2;

  do {
    i = i + 1;
    bVar1 = S[i];
    j = bVar1 + j;
    bVar2 = S[j];
    S[i] = bVar2;
    S[j] = bVar1;
    *ptr = S[bVar2 + bVar1] ^ *ptr;
    ptr = ptr + 1;
    len += -1;
  } while (len != 0);
  (*(callback * 0x10))();
  return;
}

```

Fig. 1: Microcode decompiled within Ghidra using our processor module. This function, `rc4_decrypt`, is used in the microcode update routine.

III. Framework

In this section, we present our framework for microcode reverse engineering and customization. For static analysis, we develop a Ghidra module to enable microcode decompilation and reverse engineering. For dynamic analysis, we implement a UEFI application to trace and patch microcode execution.

A. Static Analysis

Ermolov et al. provide decrypted GLM microcode and the contents of the Microcode Sequencer arrays on GitHub [34]. We leverage their findings to implement a processor module for the Ghidra decompiler. The Microcode Sequencer array content can also be extracted using CustomProcessingUnit (see III-B).

We define the semantics of each µop by reconstructing their effects from the naming scheme in the published disassembler (which, in turn, was constructed by observing the effect of µops on registers and from leaked opcode lists from Intel [34]). For the µops that were not straightforward to understand, we conduct dynamic analysis (see III-B) to observe the side effects of these µops in isolation. In total, we define semantics for 8350 µops in Ghidra's processor specification language, SLEIGH [28].

Our static analysis module takes as input the dump of microcode ROM and RAM (including sequence words) and packs them as a binary blob parsable by our Ghidra processor module. For each triad, the packer analyzes the sequence words relative to that triad, and encodes the sequence-word semantics in the respective µop of the triad. Thus, for each triad of four 48-bit µops (including the `nop` at the end) and one 32-bit sequence word, it emits four 128-bit µops to be analyzed by the Ghidra processor module. Removing the concepts of triads and sequence words simplifies the design of the Ghidra processor module.

CPU microcode is highly optimized: several basic blocks are shared between functionalities, there is no distinction between jumps and calls, and basic blocks are highly interleaved among each other to optimize for code reuse (and thus size) instead of code locality. The decompiler conducts basic analysis to identify function boundaries, reconstruct high-level control flow and internal data structures, and cross-reference these. Figure 1 shows an example of clean control flow reconstructed in Ghidra using our processor module. We leverage our decompiler to analyze microcode throughout this work.

B. Dynamic Analysis

We now introduce our framework's microcode dynamic analysis module, which is the first of its kind for Intel CPUs. In this section, we describe the reverse engineering we conducted to build it, present the implementation details, and customize `rdrand` as an example of instruction patching.

Execution Context. The module is capable of hooking, patching, and fully tracing microcode execution. It consists of a UEFI application that runs before the OS bootloader. This provides a noiseless environment for experiments and complete control over the system. Alternatively, the same MSR operations could be implemented in a Linux kernel module, trading noise for a more feature-rich execution environment.

Hardware Setup. All our tests are performed on GLM, namely Intel Celeron N3350 with cpuid `0x000506C9` and `0x000506CA`. We execute all our experiments with a fixed CPU frequency of 1.10 GHz.

1) Reverse-Engineering LDAT Accesses: We use our decompiler to reverse-engineer the CRBUS access patterns during microcode updates that allow the CPU to overwrite the MSRAM. Our starting point is the CRBUS address range mapping to the LDAT port of the Microcode Sequencer, which has been documented in prior work [15]. By cross-referencing these addresses with the decompiled microcode, we can find and analyze the microcode update routine. This lets us interact with the Microcode Sequencer by reproducing the commands that the update routine sends to its LDAT port. Listing 1 shows our primitive to write to the Microcode Sequencer's arrays. We build our dynamic analysis module upon our `ucode_sequencer_write` primitive, developing a similar primitive to read from the microcode arrays.

2) Microcode Hooks: To modify the behavior of an instruction in a custom microcode patch, we need to configure the match registers to 'hook' that instruction. By dumping the content of the match registers after a regular microcode update has been applied, we can reverse-engineer the format of the match-register entries. The following snippet shows how to compute a match entry register to hook an MSROM instruction address `match_address` to redirect execution to the MSRAM `patch_address`:

```
def compute_match_register(match_address, patch_address):
  patch_offset = ((patch_addr - 0x7c00) / 2) << 16;
  return (0x3e000000 | patch_offset | match_address |
    enabled)
```

```
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 Registers
4 -> UCODE Patch RAM
```

Listing 1: Slightly simplified sequence of commands to write the value `VAL` to the address `ADDR` of the selected (with `SELECTOR`) microcode array. Each `CRBUS` write is an invocation of the `udbgwr` instruction with `rcx=0`.

Note that the last bit of `match_address` overlaps with the enabled bit, which, when set to 0, disables the hook. The last bit of the match address is ignored by the CPU, and, to our understanding, only even addresses can be hooked.

3) Microcode Patches: Combined with our static analysis capabilities, we can modify instruction behavior. To better express the desired semantics of our patches, we develop a microcode assembler. Our assembler supports most μops and hides complex details such as instruction addresses and registers by supporting high-level constructs like labels and variables. Hooks can be generated with the `.patch` directive to set up a match-register entry automatically. As μop immediates are restricted to 16 bits, the assembler also supports macros to deal with 64-bit constants by emitting multiple instructions. Listing 3 shows an example of a microcode routine we can assemble.

4) Microcode Traces: By leveraging microcode patches and hooks, we can obtain microcode execution traces. We define a specific microcode patch that when executed reads the timestamp counter of the CPU, saves it in a specific location, disables itself and then continues execution. The hook disables itself by zeroing out the corresponding entry in the match register, accessing the CRBUS similarly to our `ucode_sequencer_write` primitive. The ability of the hook to disable itself is fundamental to making the microcode tracing work: the CPU would otherwise enter an infinite loop when resuming execution at the same address.

We apply hooks that redirect execution to our custom patch to every possible microcode address. Thus, when executing an instruction `I`, the framework dumps the timestamp at which each specific μop has been executed. Since there are a limited number of match registers, the framework iteratively executes the instruction `I`, each time registering a subset of the hooks it needs and collecting subtraces. In our implementation, we register one hook at a time for simplicity. In a post-processing stage, we reorder the μops based on the timestamp to obtain an instruction trace. Since the instruction `I` is executed multiple times, it must have a deterministic microcode control flow to obtain a coherent trace.

Algorithm 1 shows the pseudocode of the microcode hook that is installed in the CPU to collect the timestamp counter and resume execution. Algorithm 2 shows the pseudocode of the tracing collection stage algorithm. As we can only hook *even* microcode addresses, when two even addresses are

Algorithm 1: Pseudocode of the microcode hook that dumps the timestamp and resumes execution.

```
// Assume this hook is installed at index
// 0 of the match registers
function dump_ts_and_resume(addr)
    saved_ts ← read_clock()
    // disable hook by overwriting
    // the entry
    ucode_sequencer_write(
        sel: 3, // select match registers
        idx: 0, // assume idx 0
        val: 0  // 0 to disable
    )
    resume_execution(addr)
```

Algorithm 2: Pseudocode of the microcode tracing algorithm.

```
function trace_instruction(I)
    trace ← []
    // microcode addresses go
    // from 0 to 0x7c00 in GLM
    for addr in 0 .. 0x7c00 do
        install_hook(addr)
        saved_ts ← 0
        start_ts ← read_clock()
        // this triggers the
            dump_ts_and_resume hook if the
            microcode of I executes 'addr'.
        // the microcode hook removes itself
            and saves the timestamp in a
            global variable 'saved_ts'
        execute(I)
        end_ts ← saved_ts
        if end_ts > 0 then
            trace.append(end_ts - start_ts,
            addr)
        end
    end
    return sort(trace)
```

executed contiguously, we infer in the post-processing stage that the *odd* address between the two has also been executed.

5) Customizing `rdrand`: With CustomProcessingUnit's microcode hooking, patching, and tracing capabilities, we can customize the semantics of x86 instructions. As a proof of concept, we customize the behavior of the `rdrand` instruction. In most x86 processors, this generates a hardware-generated cryptographically secure random number. The carry flag in the `rflags` register indicates the success or failure of the operation after execution.

We trace the execution of `rdrand` to determine which μop to hook. Listing 2 shows the full execution trace. Analysis of the trace shows the semantics of the instruction reflected in its μops: it reads the hardware-generated random number by reading I/O port `0x40004e00` at address `0x1866`, whose value is returned in the specified register (μop `0x186a`). The carry flag is updated based on the value returned, conditionally assigning the bit 1 to the temporary register `tmp2` (address `0x1869`) and updating the `rflags` register (address `0x18c`).

```
rdrand_trace:
0428: tmp4:= ZEROEXT_DSZ32(0x0000002b)
0429: tmp2:= ZEROEXT_DSZ32(0x40004e00)
042a: tmp0:= ZEROEXT_DSZ32(0x00000439) SEQW GOTO U1861
1861: tmp1:= READURAM(0x0035, 64)
1862: TESTUSTATE(SYS, 0x20)? SEQW GOTO U1866
1866: tmp1:= PORTIN_DSZ64_ASZ16_SC1(tmp2)
1868: tmp1:= OR_DSZ64(0x00000000, tmp1)
1869: tmp2:= SELECTCC_DSZ64_CONDNZ(tmp1, 0x00000001)
186a: r64dst:= ZEROEXT_DSZ32N(tmp1)
186c: MOVEINSERTFLGS_DSZ32(tmp2) SEQW UEND0
```

Listing 2: μop execution trace of the `rdrand` instruction.

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

Listing 3: `rdrand` patch that makes the instruction return "Hello World!" in the registers `rax`-`rbx`.

To patch the instruction semantics, we only need the instruction entry point in the MSROM: address `0x0428`. We use CustomProcessingUnit to hook this entry point and redirect execution to our custom patch in MSRAM. Listing 3 shows our patch to make `rdrand` return "`Hello World!`" in the registers `rax` and `rbx`. We verify the patch works by repeated execution of `rdrand`.

In the following sections, we demonstrate how CustomProcessingUnit facilitates microcode reverse engineering and customization. We reverse-engineer the confidential microcode update algorithm and present three other case studies illustrating how microcode customization can improve software performance and security.

IV. Case Study: Reverse-Engineering the Microcode Update Routine

The details of the decryption and validation performed in the microcode update routine are not documented by Intel [43]. An Intel patent describes that the patch is validated in a secure memory separate from the microcode RAM and is encrypted using public-key encryption. In particular embodiments, a private key and public key hash value are embedded within the CPU, and the patch is signed with 2048-bit RSA [69]. Experimental timing and fault analysis in prior work supports the hypothesis that 2048-bit RSA is used to sign a padded SHA2 digest [20], [40].

We leveraged the microcode tracing and decompilation capabilities of CustomProcessingUnit to precisely reverse-engineer the full microcode routine for patch decryption and verification. We release our microcode decryptor and parser along with CustomProcessingUnit. Concurrently to our work, Ermolov et al. released a tool to decrypt microcode updates based on their GLM reverse engineering [33].

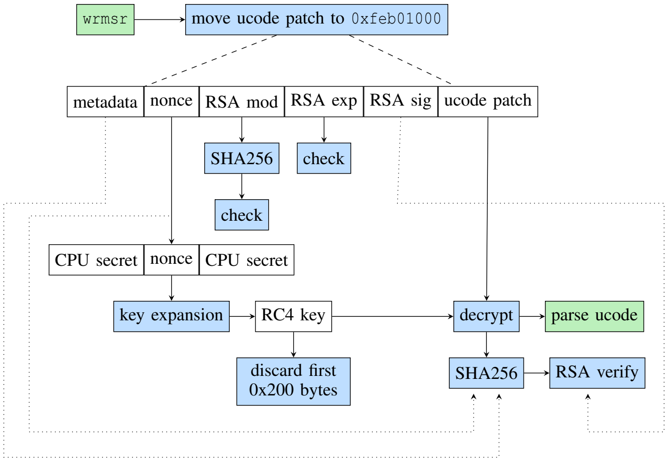


Fig. 2: High level overview of the µcode update algorithm for GLM CPUs.

A. Reverse Engineering

A microcode update is triggered by writing the linear address of the microcode update loaded in memory to the `IA32_BIOS_UPDT_TRIG` MSR. To trace the update with our microcode tracer, we need to repeat the instruction sequence (see Section III-B4). Since the same µcode update cannot be applied multiple times, and applying a µcode update would override the MSRAM and match register that we use for tracing, we must corrupt the microcode update so that the update fails.

We corrupt the single byte that produces a failing update with the maximum latency. Intuitively, the failing update with the maximum latency should provide a repeatable µop trace that most closely resembles a successful update. We repeatedly corrupt a single byte in the µcode update and track the time it takes the update to fail. We identify the maximum failure latency at the offset `0x1c0` in the update signature. This produces an update that fails at the signature verification stage, just before actually applying the update to MSRAM.

Using CustomProcessingUnit, we produce a full trace of the faulty microcode update routine and reverse-engineer it with the help of our decompiler. Figure 2 shows a high-level overview of the algorithm, illustrating how the important parts of the µcode update are parsed and verified by the CPU before applying the update.

Decryption Algorithm. The CPU first checks the validity of the pointer passed to the `wrmsr` instruction, erroring out for non-canonical addresses, addresses that may wrap around the address space, and patch sizes $> 256KB$ or $< 646B$. After these basic checks, the µcode update routine sets the last bit of the CRBUS address `0x51b`. As we will detail in the following paragraphs, this enables a secure memory region at physical addresses `0xfeb00000-0xfec00000`. It then copies the µcode patch from the user-specified address to `0xfeb01000` in the secure region to check and decrypt it in place. The range `0xfeb00000-0xfeb01000` of the secure memory is used as a scratch memory area to temporarily hold variables and metadata for the cryptographic algorithms used in the routine.

After copying the patch, the routine checks basic metadata. It ensures that the `cpuid` of the CPU is supported by the update and that the security version number of the new update is not lower than the number in the currently loaded update. It then initializes a SHA256 state in the scratch range and computes the SHA256 of the RSA modulus stored in the µcode update (bytes 0xb0 to 0x1b0), verifying that it matches a known hard-coded hash (a1b4b7417f0fdcdb0feaa26eb5b78fb2cb86153f0c e98803f5cb84ae3a45901d). It checks that the RSA exponent (bytes 0x1b0 to 0x1b4) is 0x11 (17).

The routine then proceeds to compute the decryption key for the µcode update. It generates a seed combining a 32-byte nonce from the µcode update (bytes 0x90 to 0xb0) with a 16-byte prefix and suffix. This is a hard-coded value from the µcode routine (0e77b29d9e91765da26648998b6813ab) which we call the CPU secret. The 64-byte seed is expanded to 256B by recursively computing the SHA256 hash eight times and saving each 32-byte internal state. The resulting 256B are used to initialize an RC4 keystream.

The first 512B of the RC4 keystream are discarded, and then the µcode routine proceeds to decrypt the microcode using RC4. Next, the decrypted µcode is cryptographically verified. The routine computes the SHA256 of the resulting patch, including patch metadata (e.g., µcode revision number, release date, length, cpuid target, and nonce), and verifies this against the RSA signature.

Patch Application. Once the update is verified, the patch is applied. Reverse engineering the decrypted microcode update patch routine shows that a microcode update is a custom bytecode that the CPU decodes in an interpreter loop to execute various commands while updating. Such commands include: resetting or writing microcode RAM, sequence words and match registers; sending commands to internal components through the CRBUS (e.g., to disable match registers during the update); writing internal buffers; invoking custom microcode routines; and even control flow commands to decode different commands based on the CPU state.

Secure Memory. To prevent the decrypted patch from being read or tampered with, the update process must use a secure memory region. We observe that microcode is decrypted at the temporary physical address `0xfeb01000`. Attempting to read this address during normal execution returns `0xff`, as occurs when trying to read other protected memory regions such as SGX enclave memory. The address appears to be dynamically enabled for the microcode update process by writing a bit to the CRBUS address `0x51b`. It has a fast access time (≈ 20 cycles), fits up to 256KB of data before a replacement policy is applied, and content is not shared between cores. Based on these observations, we conclude that this address is actually a special view on the L2 cache. This matches an embodiment described in an Intel patent, in which access to a cache is blocked to all other operations during decoding, validation, and installation of the update [69].

B. Security Analysis

The microcode is encrypted and signed to prevent reverse engineering and tampering. In this section, we investigate the effectiveness of the actual update-routine implementation towards those guarantees. Bypassing encryption would allow microcode to be analyzed on other Intel microarchitectures for which the CPU secret has not yet been leaked, while bypassing anti-tampering would enable the loading of arbitrary microcode on CPUs without Red Unlock, with all the associated security risks and customization benefits that entail. Although we specifically analyze our reverse-engineered GLM implementation, we expect many of these findings to generalize to other microarchitectures.

RC4 Encryption. RC4 is vulnerable to many known attacks [12], [49], although some mitigations are in place in the routine. The key (CPU secret) length of 16B is sufficient to prevent brute-forcing, and the additional use of a unique 16B nonce (*i.e.*, initialization vector, or IV) to initialize the key expansion prevents keystream reuse attacks [12]. As the IV is included in the RSA signature, the attacker cannot modify it, preventing chosen-IV attacks. RC4's weak key schedule leads to statistical bias in the generated keystream, making it vulnerable to related-key recovery attacks when the key and IV are combined together trivially, for example, via concatenation as they are in the microcode routine [49]. However, the routine does discard the initial 512B of the keystream to reduce bias. While attacks have been demonstrated exploiting bias in later bytes, these require high volumes of ciphertexts ($>$ 1000000 [49]), whereas very few microcode updates are published. Currently, only 453 Intel production microcode patches are available in a well-known repository [59], and as these cover multiple microarchitectures, they also do not all use the same key. Therefore, these mitigations are likely sufficient in practice, provided that (as is extremely likely) microcode updates continue to be produced in low volumes.

RSA Signature. The use of RSA provides strong anti-tampering protection. The 2048-bit modulus is sufficient to prevent brute-forcing, PKCS#1 v1.5 padding is used, and the signature check appears experimentally to be constant-time [40].

Several methods are known to bypass RSA signatures when the public key is not correctly checked [10], [60]. However, while the RSA modulus and exponent for the signature are provided directly in the update metadata, and are therefore attacker-controlled, they must match the expected values hard-coded in the update routine. Thus, it is not possible to pass different public key information, replacing the modulus or exponent to bypass the signature verification.

The hash that the RSA signature is computed on includes the µcode update metadata (see Section IV). This prevents metadata tampering, such as changing the security revision number to downgrade the microcode or modifying the cpuid value to apply a patch for a different CPU model. However, some of these metadata values are used before being verified. The length of the microcode update is included in the hash, but it is actively used before. Since the algorithm includes checks on the minimum and maximum values, we could not leverage such a potential time-of-check-time-of-use vulnerability to, e.g., cause integer overflows in the routine.

Corrupting Updates. An attacker could try to leverage a race condition and corrupt the decrypted microcode in memory before it is applied. However, the microcode is decrypted in place inside the reserved secure memory area that we hypothesize is the L2 cache. The view on the secure memory is only enabled during microcode updates, and cores trying to access its physical address during normal execution read only `0xff`. We verified that each core has a unique secure memory area (as the L2 cache is not shared); thus, its content cannot be modified by a different core during an update. Since the update is triggered by a serializing `wrmsr` instruction, the hyperthread parallel to the logical core executing the update is frozen and cannot modify the secure memory either.

We can obtain partial leaks of decrypted microcode. We achieve this by mapping the APIC MMIO region at the address `0xfeb01000` that the microcode will be moved to. This effectively makes the APIC MMIO region shadow the secure memory region and hijack it from the microcode routine, similar to the Memory Sinkhole attack [23]. However, we cannot fully leak or corrupt updates due to the limitations on the writable areas of the APIC MMIO region [43].

An alternative would be to map the APIC to the scratch region used in the secure memory to corrupt the SHA256 state used to compute hashes. While this is possible and corrupts the output of the hash algorithms (which would bypass the signature check), it also corrupts the hash of the RSA public key, which is checked, and thus the update fails.

Microarchitectural Attacks. One could try to leverage microarchitectural attacks to leak microcode updates during decryption. Most microarchitectural attacks require execution on either the same core or a hyperthread during the update and can only leak internal buffers close to the CPU, such as the L1 cache or Line Fill Buffers [17], [56], [66], [70], [71]. The secure memory is accessed in µcode using µops that access directly uncacheable physical addresses, thus avoiding caches. While the Line Fill Buffers could be filled with such data, hyperthreading cannot be used during the update, and internal buffers seem to be flushed afterward, making the attack ineffective.

ÆPIC Leak targets structures deeper in the memory hierarchy, *i.e.*, the superqueue [14]. However, as the superqueue holds entries flushed from the L2 cache to the LLC, the decrypted microcode does not pass through. An inverse attack could attempt to leverage ÆPIC Leak to insert values from the superqueue. By modifying the content of the superqueue before the update and once again mapping the APIC MMIO region over the secure scratch memory, an attacker could make the CPU read leaked values from the superqueue instead and thus corrupt computations with finer control. However, the µcode update routine also flushes internal buffers *before* the update.

V. Case Study: x86 Pointer Authentication Codes

A. Background

Pointer Authentication Codes (PACs) aim to protect sensitive pointers from attacks that may leverage memory corruption vulnerabilities to hijack the control flow. PACs were introduced in ARMv8.3 [5] and are used in several ARM-based systems to provide strong security guarantees [63]. A PAC is a message authentication code embedded in the high bits of the pointer it protects. The code depends on the pointer itself, a 64-bit context, and a secret key from a set of five possible keys. The algorithm used to compute the code is vendor-specific [8], but the standard recommends the QARMA family of tweakable block ciphers [7]. On most CPUs, PAC is implemented directly in hardware in a single μop [26].

The ARM instruction set provides different instructions to compute the PAC and embed it in the pointer (e.g., `pacia`, `pacib`), to verify and remove it (`autia`, `autib`), or to simply clear it. The suffix of the instruction selects which key is used to compute or verify the signature. Upon successful verification, `aut` instructions remove the PAC from the high bits of the pointer, while on verification failure, the bits are not cleared. Thus, once signed, a pointer can be accessed only after being verified, and causes a memory access fault otherwise.

B. Implementation

We present the first public implementation of PAC on x86, enabling cheap and strong hardware-based control-flow-integrity protection on Intel platforms. We implement instructions to sign and verify 64-bit pointers leveraging CustomProcessingUnit.

As a proof of concept, we define two new microcode routines that sign and verify a context and a pointer, with a PAC saved in the pointer's high bits. The size of the PAC can be customized in our design, provided that it does not conflict with the used bits in the pointers, and any unused microcoded instructions can be chosen as the signing and verification instructions. By default, we select a 16-bit PAC and patch `verw` and `verr`, programming the match registers to hook them with our routines.

Our microcoded signing algorithm uses a single round of SipHash [6]. The SipHash algorithm has been developed for keyed hashing optimized for small inputs, which is exactly our use case with 64-bit pointers. We favor it over QARMA as the latter uses bit shuffles, which are faster in hardware but costly in microcode. We define a 64-bit secret key to be kept in an internal buffer of the CPU, the staging buffer, so that it is never architecturally exposed.

Listing 4 shows our microcode for the PAC signing algorithm. The authentication routine is similar, but in addition, it verifies that the existing PAC on the input pointer matches the one generated from scratch, corrupting the high bits if they do not match. Our x86 microcode implementation of the PAC signature takes 25 clock cycles to execute, while the authentication operation takes 26. We verify that the implementation works as expected by signing pointers and verifying that they authenticate when not tampered with and cause an invalid memory access otherwise.

C. Security Analysis

For the security analysis of our x86 PAC implementation, we focus on the resistance of our implementation to speculative execution attacks such as PACMAN [65]. We refer the reader to the original SipHash paper for a security evaluation of the SipHash algorithm [6].

PACMAN. This attack bypasses pointer authentication by speculatively authenticating a corrupted pointer, using side channels to identify a correct PAC. Because the authentication routine is speculatively rather than architecturally executed, the attack can brute-force the PAC without causing the program to crash.

We re-implement the attack on our CPU, effectively developing the first x86 PACMAN attack. As in the original paper, we leverage a PACMAN gadget that authenticates and then accesses a pointer behind a branch whose direction we can determine. We train the branch prediction by executing the branch with a valid pointer with a correct PAC for 10 iterations. We then use an artificial memory-corruption vulnerability to override the pointer with an attacker-controlled value and PAC and change the branch condition so the authentication instructions are no longer architecturally executed. While the gadget is not executed architecturally, it is executed speculatively, and the pointer, if valid, is dereferenced speculatively. For simplicity, we make the pointer point to shared memory between the attacker and the victim and use Flush+Reload [74] to infer whether the pointer location was accessed, *i.e.*, the PAC was correct.

Implementing the attack on our x86 PAC implementation fails to leak valid PAC values. This is due to the smaller Reorder Buffer (ROB, 78 entries) and Physical Register File (PRF, 56 entries) in GLM CPUs [4]. Our x86 PAC implementation consists of 54 μops, filling up the entire speculative window and preventing subsequent speculative access. However, on a CPU with a wider ROB and PRF, the attack would succeed as it is the PAC design, not the implementation, that is vulnerable to PACMAN. To test this hypothesis, we write a weaker version of the PAC implementation in 27 μops, which only partially implements SipHash. With this shorter PAC implementation, the attack successfully brute-forces a valid PAC for a given pointer in less than a second.

Mitigation. We leverage CustomProcessingUnit to investigate how PACMAN could be mitigated in microcode. As a first attempt, we could add a speculation barrier to the PAC implementation. However, such a solution would incur considerable overhead, slowing the execution of the PAC instruction (by around 10 cycles) and other instructions in the pipeline. Moreover, an attacker could find a gadget where the pointer authentication occurs on a non-speculative path while the access occurs on a speculative one, re-enabling the attack.

Thus, we investigate a solution that does not involve fences. The core concept is to make the pointer-authentication operations fault when an invalid PAC is detected while also removing the PAC from the pointer. This means that the pointer is always valid in the speculative path, removing the side channel on PAC validity. Faulting ensures that memory corruption attacks using corrupted pointers architecturally still fail. Our mitigated implementation that triggers an exception when detecting an invalid PAC has no overhead with respect to the original, and is the first public PAC implementation not vulnerable to the PACMAN attack. However, one drawback is that this design re-enables Spectre attacks [50] using corrupted pointers in the speculative path. A further remaining attack surface is the use of a port contention side-channel [3] to detect the micro-operations that cause a fault in the speculative path. This attack surface could be closed by disabling hyperthreading or preventing an attacker from running parallel to the victim.

```
.org 0x7c00

# declare variables
let [ptr] := r64dst;       let [v0] := tmp1
let [ctx] := r64src;       let [v1] := tmp2
let [key] := tmp0;         let [v2] := tmp3
let [key_addr] := 0xba40;  let [v3] := tmp4
let [pac] := tmp5

# --- initialize ---
[key] := LDSTGBUF_DSZ64_ASZ16_SC1([key_addr])
# v0 = 0x736f6d6570736575 ^ key;
[v0] := ZEROEXT_MACRO(0x736f6d6570736575)
[v0] := XOR_DSZ64([v0], [key])
# v1 = 0x646f72616e646f6d ^ ctx;
[v1] := ZEROEXT_MACRO(0x736f6d6570736575)
[v1] := XOR_DSZ64([v1], [ctx])
# v2 = 0x6c7967656e657261 ^ key;
[v2] := ZEROEXT_MACRO(0x736f6d6570736575)
[v2] := XOR_DSZ64([v2], [key])
# v3 = 0x7465646279746573 ^ ctx;
[v3] := ZEROEXT_MACRO(0x736f6d6570736575)
[v3] := XOR_DSZ64([v3], [ctx])

# --- update ---
[v3] := XOR_DSZ64([v3], [ptr]) # v3 ^= ptr;
[v0] := ADD_DSZ64([v0], [v1]) # v0 += v1;
[v2] := ADD_DSZ64([v2], [v3]) # v2 += v3;
[v1] := ROL_DSZ64([v1], 0x0d) # v1 = RotateLeft<13>(v1);
[v3] := ROL_DSZ64([v3], 0x10) # v3 = RotateLeft<16>(v3);
[v1] := XOR_DSZ64([v1], [v0]) # v1 ^= v0;
[v3] := XOR_DSZ64([v3], [v2]) # v3 ^= v2;
[v0] := ROL_DSZ64([v0], 0x20) # v0 = RotateLeft<32>(v0);
[v2] := ADD_DSZ64([v2], [v1]) # v2 += v1;
[v0] := ADD_DSZ64([v0], [v3]) # v0 += v3;
[v1] := ROL_DSZ64([v1], 0x11) # v1 = RotateLeft<17>(v1);
[v3] := ROL_DSZ64([v3], 0x15) # v3 = RotateLeft<21>(v3);
[v1] := XOR_DSZ64([v1], [v2]) # v1 ^= v2;
[v3] := XOR_DSZ64([v3], [v0]) # v3 ^= v0;
[v2] := ROL_DSZ64([v2], 0x20) # v2 = RotateLeft<32>(v2);
[v0] := XOR_DSZ64([v0], [ptr]) # v0 ^= ptr;

# --- finalize ---
[v2] := XOR_DSZ64([v2], 0xff) # v2 ^= 0xFF;
[v0] := ADD_DSZ64([v0], [v1]) # v0 += v1;
[v2] := ADD_DSZ64([v2], [v3]) # v2 += v3;
[v1] := ROL_DSZ64([v1], 0x0d) # v1 = RotateLeft<13>(v1);
[v3] := ROL_DSZ64([v3], 0x10) # v3 = RotateLeft<16>(v3);
[v1] := XOR_DSZ64([v1], [v0]) # v1 ^= v0;
[v3] := XOR_DSZ64([v3], [v2]) # v3 ^= v2;
[v0] := ROL_DSZ64([v0], 0x20) # v0 = RotateLeft<32>(v0);
[v2] := ADD_DSZ64([v2], [v1]) # v2 += v1;
[v0] := ADD_DSZ64([v0], [v3]) # v0 += v3;
[v1] := ROL_DSZ64([v1], 0x11) # v1 = RotateLeft<17>(v1);
[v3] := ROL_DSZ64([v3], 0x15) # v3 = RotateLeft<21>(v3);
[v1] := XOR_DSZ64([v1], [v2]) # v1 ^= v2;
[v3] := XOR_DSZ64([v3], [v0]) # v3 ^= v0;
[v2] := ROL_DSZ64([v2], 0x20) # v2 = RotateLeft<32>(v2);

# pac =  ((v0 ^ v1) ^ (v2 ^ v3)) << 48;
[pac] := XOR_DSZ64([v0], [v1])
[pac] := XOR_DSZ64([pac], [v2])
[pac] := XOR_DSZ64([pac], [v3])
[pac] := SHL_DSZ64([pac], 0x30)

# sign ptr
[ptr] := XOR_DSZ64([pac], [ptr])
```

Listing 4: x86 PAC signature microcode routine leveraging a single round of SipHash.

VI. Case Study: µsoftware Breakpoints

A. Background

Software breakpoints are widely used both for debugging [39] and instrumentation [36], [58], [61], [76]. The `int3` (`0xcc`) instruction triggers an interrupt calling the debug exception handler with a breakpoint exception [43]. The interrupt routine handling the breakpoint exception in the kernel then generates a `SIGTRAP` signal to the user space process. Thus, for every breakpoint hit during execution, the application issuing a breakpoint incurs a context switch to the interrupt routine in kernel space and another context switch back to user space.

Applications can leverage interfaces provided by the operating system to debug child processes through breakpoints (e.g., `ptrace` for Linux). These make it easy to trigger the execution of specific instructions once a breakpoint is hit. While such interfaces are convenient, they incur significant overhead (up to 10 000 cycles in our system) due to the multiple context switches between processes. It is, therefore, crucial to work around this performance limitation for high-performance instrumentation.

One such high-performance application of breakpoint-based instrumentation is binary-level fuzzing. Fuzzing is one of the most prominent techniques to find memory-corruption vulnerabilities [37], [57], [62], [75]. It uses coverage-guided feedback to discover random inputs that exercise different paths of a program to expose bugs. While source-level fuzzing involves custom compilation passes to insert coverage collection instrumentation, binary-level fuzzing usually relies on binary instrumentation [72] or breakpoint-based instrumentation [36], [58], [61], [76].

Breakpoint-based instrumentation relies on aggressive optimizations to mitigate the performance hit of software breakpoints, e.g., removing the breakpoint completely once it has first been hit and an input reaching that coverage point has been collected [36]. Such an optimization eventually converges to zero performance overhead but sacrifices useful path information on branches that have already been hit [38].

B. Implementation

We leverage CustomProcessingUnit to implement a new type of software breakpoints that we name *µsoftware breakpoints* (bp). The idea is to run the breakpoint logic directly at the µcode level. To implement these, we change the semantics of the `icebp`/`int1` (`0xf1`) instruction, which should not be used by normal software. Placing the breakpoint logic directly in microcode avoids the cost of interrupts or context switches and is thus extremely fast. The OS would provide an interface to program µsoftware breakpoints, loading the required µcode in the running core. No further interaction is needed, as each `icebp` instruction will then execute the bp logic.

As a proof of concept, we implement µsoftware breakpoints for coverage collection in binary-level fuzzing. We save the address of the coverage map in an internal buffer of the CPU for ease of access inside microcode and simply update the coverage based on the current instruction pointer during breakpoint execution. Listing 5 shows an example implementation of µsoftware breakpoints for coverage-guided fuzzing. Each time an `icebp` instruction is executed, the coverage is updated

```
.org 0x7c00
.patch 0xc40 # icebp entry point
.entry 0

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)
```

Listing 5: µsoftware breakpoint to collect code coverage.

directly and saved into the coverage map with no interrupt or context switch.

As a microbenchmark, we measure the overhead of executing a single µsoftware breakpoint to collect coverage information, averaging 1 million executions. For each µsoftware breakpoint, the CPU has a latency of 10 cycles, which is mostly due to the switch to the microcode RAM by the Microcode Sequencer. This is around 1000x faster than ptrace-based instrumentation. To compare with the fastest non-microcoded implementation, we also implement the same logic for coverage collection directly in the kernel debug exception handler. While such an implementation is faster than user space coverage collection, we still measure a latency of 388 cycles, making µsoftware breakpoints 38.8x faster.

VII. Case Study: Constant-time Hardware Division

A. Background

Side-channel attacks allow adversaries to leak secret values by observing secret-dependent side effects of computation, such as differences in execution time or microarchitectural state [9], [56]. Constant-time programming aims to produce algorithms resistant to timing side-channel attacks, implemented so that the same instruction and memory access patterns occur regardless of the secret input [9], [44]. Several solutions have been proposed to automatically rewrite software to be constant-time [13], [19], [64], [67], [73]. They typically involve transforming programs during compilation to consistently execute the same sequence of operations irrespective of their input.

While this is effective for instruction traces and memory access patterns, there are some instructions whose latency depends on the input values [46]. Examples of such instructions are division or remainder operations and many floating-point operations. Executing these instructions on secret data may leak information about their operands or the result. Automated solutions to mitigate such side channels rely on software wrappers implementing these operations in constant-time. However, substituting a single instruction with a full software wrapper incurs substantial overhead. It increases the code size and requires complete recompilation of the program or precise binary patching.

B. Implementation

As a case study, we patch one of these instructions, unsigned integer division (div), to provide constant-time guarantees at the microcode level instead. Intel CPUs implement division directly in hardware [18], but we verified that the div instruction itself is microcoded and thus can be patched. The microcode of div simply calls the µops that control hardware operations to perform the division, taking between 22 and 40 cycles to execute.

We take the state-of-the-art 64-bit constant-time division software implementation from Constantine [13] and reimplement it in microcode. The Constantine software implementation takes 694 cycles. Our microcode implementation (depicted in Listing 6) takes just 438 cycles to execute, 1.58x faster than in software. This speedup is thanks to the reduced number of fetched and decoded instructions, higher instruction cache locality, reduced register pressure, and faster microcode jumps (that have control over the branch predictors).

This has the further advantage of not requiring any transformation of the input program. Once the patch is installed, any div instruction executed is constant-time, and it can be enabled and disabled as needed to prevent unnecessary overhead when timing guarantees are not required.

```
.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 [cmp]      := 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])
# cmp = (temp >= divisor);
[cmp]  := SUB_DSZ64([divisor], [temp])
# temp -= cmp? divisor : 0;
[temp3]:= SELECTCC_DSZ64_CONDB([cmp], [divisor])
[temp] := SUB_DSZ64([temp3], [temp])
# quotient |= cmp ? 1uLL << i : 0;
[temp4]:= SHL_DSZ64(0x1, [i])
[temp5]:= SELECTCC_DSZ64_CONDB([cmp], [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)
```

Listing 6: Constant-time div microcode routine.

VIII. Conclusion

In this work, we presented a static and dynamic analysis framework for reverse engineering of Intel x86 microcode for the Goldmont microarchitecture. We demonstrated our framework's utility and the potential security and performance benefits of microcode customization in four case studies.

Our framework builds upon research reverse engineering Intel's debug infrastructure [35]. Debug infrastructure is crucial for post-silicon validation, but its security should rely on transparent mechanisms rather than on security by obscurity. Given our increasing reliance on critical digital infrastructure, both manufacturer documentation [45] and reverse engineering efforts will play an important role in ensuring the underlying hardware is secure.

Acknowledgments

We would like to thank Mark Ermolov, Maxim Goryachy, and Dmitry Sklyarov for their extensive research into and reverse engineering of the Intel Management Engine, Intel debug infrastructure, and Intel microcode, without which this work would not have been possible. Any opinions, findings, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties or of the authors' affiliations.

References

[1] Abel, A., and Reineke, J. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *ASPLOS* (2019).

[2] Alaoui, Y. Exploiting Intel's Management Engine - Part 2: Enabling Red JTAG Unlock on Intel ME 11.x (INTEL-SA-00086), https://kakaroto.homelinux.net/2019/11/exploiting-intels-management-engine-part-2-enabling-red-jtag-unlock-on-intel-me-11-x-intel-sa-00086/ 2019.

[3] Aldaya, A. C., Brumley, B. B., ul Hassan, S., García, C. P., and Tuveri, N. Port Contention for Fun and Profit. In *S&P* (2019).

[4] Anton Ertl. Reorder buffer size of various CPUs, http://www.complang.tuwien.ac.at/anton/robsize/ 2019.

[5] ARM. Arm Architecture Reference Manual for A-profile architecture, 2022.

[6] Aumasson, J.-P., and Bernstein, D. J. SipHash: a fast short-input PRF. Cryptology ePrint Archive, Paper 2012/351, 2012.

[7] Avanzi, R. The QARMA Block Cipher Family. Almost MDS Matrices Over Rings With Zero Divisors, Nearly Symmetric Even-Mansour Constructions With Non-Involutory Central Rounds, and Search Heuristics for Low-Latency S-Boxes. *IACR Transactions on Symmetric Cryptology* (2017).

[8] Azad, B. Examining Pointer Authentication on the iPhone XS, https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html 2019.

[9] Barthe, G., Grégoire, B., and Laporte, V. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time". In *CSF* (2018).

[10] Bleichenbacher, D. Forging Some RSA Signatures with Pencil and Paper. *CRYPTO* (2006).

[11] Bölük, C. Speculating The Entire X86-64 Instruction Set In Seconds With This One Weird Trick, https://blog.can.ac/2021/03/22/speculating-x86-64-isa-with-one-weird-trick 2021.

[12] Borisov, N., Goldberg, I., and Wagner, D. Intercepting Mobile Communications: The Insecurity of 802.11. In *MobiCom* (2001).

[13] Borrello, P., D'Elia, D. C., Querzoni, L., and Giuffrida, C. Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization. In *CCS* (2021).

[14] Borrello, P., Kogler, A., Schwarzl, M., Lipp, M., Gruss, D., and Schwarz, M. ÆPIC Leak: Architecturally leaking uninitialized data from the microarchitecture. In *USENIX Security* (2022).

[15] Bosch, P. Intel LDAT notes, https://pbx.sh/ldat 2020.

[16] Bosch, P. Under the hood of a CPU: Reverse Engineering the P6 Microcode. In *Hardwear.io Netherlands* (2020).

[17] Canella, C., Genkin, D., Giner, L., Gruss, D., Lipp, M., Minkin, M., Moghimi, D., Piessens, F., Schwarz, M., Sunar, B., Van Bulck, J., and Yarom, Y. Fallout: Leaking Data on Meltdown-resistant CPUs. In *ACM CCS* (2019).

[18] Carter, T., and Robertson, J. Radix-16 signed-digit division. *IEEE Transactions on Computers* (1990).

[19] Cauligi, S., Soeller, G., Johannesmeyer, B., Brown, F., Wahby, R. S., Renner, J., Grégoire, B., Barthe, G., Jhala, R., and Stefan, D. FaCT: A DSL for Timing-Sensitive Computation. In *PLDI* (2019).

[20] Chen, D. D., and Ahn, G.-J. Security Analysis of x86 Processor Microcode. Bachelor's Thesis, Arizona State University, 2014.

[21] Coe, T. Inside the Pentium-fdiv Bug. *Doctor Dobb's Journal* (1995).

[22] Davis, J., Slobodova, A., and Swords, S. Microcode Verification - Another Piece of the Microprocessor Verification Puzzle. In *International Conference on Interactive Theorem Proving* (2014).

[23] Domas, C. The Memory Sinkhole. In *BlackHat USA* (2015).

[24] Domas, C. Breaking the x86 ISA. In *BlackHat USA* (2017).

[25] Domas, C. God Mode Unlocked: Hardware Backdoors in x86 CPUs. In *BlackHat USA* (2018).

[26] Dougall, J. Apple M1 Microarchitecture Research, https://dougallj.github.io/applecpu/firestorm.html 2021.

[27] Duflot, L., Etiemble, D., and Grumelard, O. Using CPU system management mode to circumvent operating system security functions. *CanSecWest* (2006).

[28] Eagle, C., and Nance, K. *The Ghidra Book: The Definitive Guide*. No Starch Press, 2020.

[29] Easdon, C. Undocumented CPU Behavior: Analyzing Undocumented Opcodes on Intel x86-64. Talk, https://www.cattius.com/images/undocumented-cpu-behavior.pdf June 2018.

[30] Easdon, C., Schwarz, M., Schwarzl, M., and Gruss, D. Rapid Prototyping for Microarchitectural Attacks. In *USENIX Security* (2022).

[31] Ermolov, M., Sklyarov, D., and Goryachy, M. glm-ucode, https://github.com/chip-red-pill/glm-ucode 2020.

[32] Ermolov, M., Sklyarov, D., and Goryachy, M. Chip Red Pill: How we Achieved the Arbitrary [micro]Code Execution inside Intel Atom CPUs. In *OffensiveCon 22* (2022).

[33] Ermolov, M., Sklyarov, D., and Goryachy, M. MicrocodeDecryptor, https://github.com/chip-red-pill/MicrocodeDecryptor 2022.

[34] Ermolov, M., Sklyarov, D., and Goryachy, M. uCodeDisasm, https://github.com/chip-red-pill/uCodeDisasm 2022.

[35] Ermolov, M., Sklyarov, D., and Goryachy, M. Undocumented x86 Instructions to Control the CPU at the Microarchitecture Level in Modern Intel Processors. *Journal of Computer Virology and Hacking Techniques* (2022).

[36] Falk, Brandon. Mesos, https://github.com/gamozolabs/mesos 2020.

[37] Fioraldi, A., Maier, D., Eissfeldt, H., and Heuse, M. AFL++: Combining Incremental Steps of Fuzzing Research. In *WOOT* (2020).

[38] Gan, S., Zhang, C., Qin, X., Tu, X., Li, K., Pei, Z., and Chen, Z. CollAFL: Path Sensitive Fuzzing. In *IEEE S&P* (2018).

[39] GNU. GDB: The GNU Project Debugger, https://www.sourceware.org/gdb 2022.

[40] Hawkes, B. Notes on Intel Microcode Updates, https://inertiawar.com/microcode 2012.

[41] Hennessy, J. L., and Patterson, D. A. *Computer Architecture, Fifth Edition: A Quantitative Approach.* Morgan Kaufmann, 2011.

[42] Intel. Application Note AP-526: Optimizations for Intel's 32-Bit Processors, 1995.

[43] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide, 2019.

[44] INTEL. Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations. *Developer Zone - Secure Coding* (2020).

[45] INTEL. Intel Debug Technology, https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/intel-debug-technology.html 2021.

[46] INTEL. Data Operand Independent Timing Instruction Set Architecture (ISA) Guidance, https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html 2022.

[47] INTEL. Intel® 64 and IA-32 Architectures Optimization Reference Manual, 2022.

[48] KAIVOLA, R., GHUGHAL, R., NARASIMHAN, N., TELFER, A., WHITTEMORE, J., PANDAV, S., SLOBODOVÁ, A., TAYLOR, C., FROLOV, V., REEBER, E., AND NAIK, A. Replacing Testing with Formal Verification in Intel® Core™ i7 Processor Execution Engine Validation. In *CAV* (2009).

[49] KLEIN, A. Attacks on the RC4 stream cipher. *Designs, Codes and Cryptography 48*, 3 (2008).

[50] KOCHER, P., HORN, J., FOGH, A., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre Attacks: Exploiting Speculative Execution. In *S&P* (2019).

[51] KOGLER, A., WEBER, D., HAUBENWALLNER, M., LIPP, M., GRUSS, D., AND SCHWARZ, M. Finding and Exploiting CPU Features using MSR Templating. In *IEEE S&P* (2022).

[52] KOLLENDA, B., KOPPE, P., FYRBIAK, M., KISON, C., PAAR, C., AND HOLZ, T. An Exploratory Analysis of Microcode as a Building Block for System Defenses. In *ACM CCS* (2018).

[53] KOPPE, P., KOLLENDA, B., FYRBIAK, M., KISON, C., GAWLIK, R., PAAR, C., AND HOLZ, T. Reverse Engineering x86 Processor Microcode. In *USENIX Security* (2017).

[54] KWAN, D., SHTOYK, K., SEREBRYANY, K., LIFANTSEV, M. L., AND HOCHSCHILD, P. SiliFuzz: Fuzzing CPUs by proxy. Google Research, 2021.

[55] LI, W. System-on-chip devices and methods for testing system-on-chip devices. Patent WO2017164872A1, 2017.

[56] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., FOGH, A., HORN, J., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security* (2018).

[57] LLVM PROJECT. libFuzzer – a library for coverage-guided fuzz testing., https://llvm.org/docs/LibFuzzer.html 2018.

[58] LUO, S. trapfuzzer: Coverage-guided Binary Fuzzing with Breakpoints. In *HITB SecConf* (2021).

[59] MAVROPOULOS, P. CPUMicrocodes: Intel, AMD, VIA & Freescale CPU Microcode Repositories, https://github.com/platomav/CPUMicrocodes 2022.

[60] MISARSKY, J. F. How (not) to Design RSA Signature Schemes. In *Public Key Cryptography* (1998).

[61] NAGY, S., AND HICKS, M. Full-speed Fuzzing: Reducing Fuzzing Overhead through Coverage-guided Tracing. In *IEEE S&P* (2019).

[62] PAYER, M. The Fuzzing Hype-Train: How Random Testing Triggers Thousands of Crashes. *IEEE Security and Privacy* (2019).

[63] QUALCOMM TECHNOLOGIES INC. Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions, https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf 2017.

[64] RANE, A., LIN, C., AND TIWARI, M. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *USENIX Security* (2015).

[65] RAVICHANDRAN, J., NA, W. T., LANG, J., AND YAN, M. PACMAN: Attacking ARM Pointer Authentication with Speculative Execution. In *ISCA* (2022).

[66] SCHWARZ, M., LIPP, M., MOGHIMI, D., VAN BULCK, J., STECKLINA, J., PRESCHER, T., AND GRUSS, D. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *ACM CCS* (2019).

[67] SOARES, L., AND PEREIRA, F. M. Q. Memory-Safe Elimination of Side Channels. In *CGO* (2021).

[68] SUTHERLAND, J., COULL, N., AND MACLEOD, A. CPU covert channel accessible from JavaScript. *CyberForensics* (2014).

[69] SUTTON, J. A. Microcode Patch Authentication. Patent US20030196096A1, 2003.

[70] VAN BULCK, J., MINKIN, M., WEISSE, O., GENKIN, D., KASIKCI, B., PIESSENS, F., SILBERSTEIN, M., WENISCH, T. F., YAROM, Y., AND STRACKX, R. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security* (2018).

[71] VAN SCHAIK, S., MILBURN, A., ÖSTERLUND, S., FRIGO, P., MAISURADZE, G., RAZAVI, K., BOS, H., AND GIUFFRIDA, C. RIDL: Rogue In-flight Data Load. In *IEEE S&P* (2019).

[72] VOSS, NATHAN. afl-unicorn: Fuzzing Arbitrary Binary Code, https://medium.com/hackernoon/afl-unicorn-fuzzing-arbitrary-binary-code-563ca28936bf 2017.

[73] WU, M., GUO, S., SCHAUMONT, P., AND WANG, C. Eliminating Timing Side-Channel Leaks Using Program Repair. In *ISSTA* (2018).

[74] YAROM, Y., AND FALKNER, K. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security* (2014).

[75] ZALEWSKI, M. American Fuzzy Lop, https://github.com/Google/AFL 2021.

[76] ZHOU, C., WANG, M., LIANG, J., LIU, Z., AND JIANG, Y. Zeror: Speed Up Fuzzing with Coverage-sensitive Tracing and Scheduling. In *ASE* (2020).