UNDOCUMENTED X86 INSTRUCTIONS TO CONTROL THE CPU AT THE MICROARCHITECTURE LEVEL IN MODERN INTEL PROCESSORS

Mark Ermolov
Positive Technologies
mermolov@ptsecurity.com

Dmitry Sklyarov
Positive Technologies
dsklyarov@ptsecurity.com

Maxim Goryachy
independent researcher
m.goryachy@gmail.com

July 7, 2021

ABSTRACT

At the beginning of 2020, we discovered the Red Unlock technique that allows extracting microcode (ucode) and targeting Intel Atom CPUs. Using the technique we were able to research the internal structure of the microcode and then x86 instructions implementation. We found two undocumented x86 instructions which are intendent to control the microarhitecture for debug purposes. In this paper we are going to introduce these instructions and explain the conditions under which they can be used on public-available platforms. We believe, this is a unique opportunity for third-party researchers to better understand the x86 architecture.

Disclamer. All information is provided for educational purposes only. Follow these instructions at your own risk. Neither the authors nor their employer are responsible for any direct or consequential damage or loss arising from any person or organization acting or failing to act on the basis of information contained in this paper.

Keywords Intel · Microcode · Undocumented · x86

1 Introduction

The existence of undocumented mechanisms in the internals of modern CPUs has always been a concern for information security researchers and ordinary users. Assuming that such mechanisms do exist, the main worry is that it might be possible to bypass the implemented CPU protection mechanisms that control access to memory and peripherals containing personal user data. It is considered to be unacceptable for a processor manufacturer or any other organization to be able to get unauthorized access to data that the user regards as confidential by bypassing documented, well-known protection mechanisms and using methods that only the manufacturer knows about (whether those methods are based on the processor functionality or are intentionally designed for such purposes, or have a legitimate purpose but also allow bypassing protection due to certain architectural properties).

Such mechanisms include the following:

1. Special CPU operating modes in which standard access control rules do not work or work differently compared to what is described in the documentation (for Intel CPUs, it is the Software Developer's Manual [1], which is intended for developers of applications running on Intel processors).
2. Special CPU microarchitectural states in which certain access control rules can be violated (for example, such states can be achieved by performing a specific instruction sequence).

3. Undocumented CPU instructions that allow bypassing protection mechanisms (for example, memory protection) and grant special privileges to the calling code.

The existence of such undocumented capabilities in Intel processors has always been a subject for debates and speculations among information security specialists. For more than 40 years of the existence of the x86 architecture, a number of significant studies have been conducted on this subject [2],[3],[4]. Several undocumented features have been discovered (e.g. undocumented instructions described in [3]), including one that allows bypassing implemented protection [5], but this was found for very old Intel processors (80286 and 80386). Thereafter no apparent undocumented methods to gain unauthorized access have been found in x86 processors manufactured directly by Intel. This is leaving aside multiple recently discovered transient execution vulnerabilities and data leaks caused by side-channel attacks (such as Meltdown and Spectre), which are by no means intentional and are generally admitted to be design errors due to prioritizing higher performance over security. However, our research team managed to find at least two undocumented x86 instructions and a special processor debugging mode called Red Unlock (detailed information about the mode is given in a separate section of this paper) in actual Intel CPUs. Combined, these capabilities make it possible to completely bypass the existing memory protection and equipment control mechanisms. Moreover, the discovered instructions provide access to the processor at the microcode level and thus allow overriding the semantics of the main x86 instructions (that is, a set of the most important instructions in terms of security). This is by far more dangerous than, for example, violating memory protection (x86 privilege levels, or so-called protection rings) because it enables malware to be implemented at a level where it is impossible to detect it with modern protection tools and defend the system against it.

We should clarify in advance that, in our opinion, these instructions were initially introduced to facilitate the debugging of processors at the microarchitecture (hardware implementation) level exclusively by Intel engineers. However, due to a total absence of available documentation on these features (including description of all the possible ways to use them), we believe that these instructions are rather dangerous and that the Red Unlock service mode intended for processor debugging must not be implemented in production parts distributed by sales outlets and utilized by end users in their everyday tasks. In our view, Intel was obliged to at least inform users about the existence of these instructions and special CPU debugging mode so that users could be aware of the risks associated with the company's products. Moreover, on some platforms it is possible to enable this CPU debugging mode using hardware means that can be accessed by the Intel Converged Security and Management Engine (Intel CSME) [6]. A number of critical arbitrary code execution vulnerabilities [7], [8] recently discovered in the Intel CSME subsystem can be exploited to switch the CPU to the debugging mode, as was demonstrated by our company's experts [9]. This makes the Red Unlock mode available for virtually anyone who wishes to use it (rather than to Intel engineers only, as it was originally intended).

2 Undocumented x86 instructions

So, we have found two undocumented instructions of the x86 architecture implemented in modern Intel processors:

1. **udbgrd** - **0F 0E** - microarchitecture (uarch) debug read

   This instruction is to read microarchitectural data. It accepts the following arguments:

   rcx - is a command that defines the type (source) of data to read:

   0x00 - indicates the Control Register Bus (CRBUS) that connects all executive units within the CPU core, such as Instruction Fetch Unit (IFU), Data Cache Unit (DCU), Microcode Sequencer (MS), and execution core (CORE). Each unit has its own range of addresses for control registers accessible via the CRBUS.

   0x08 - indicates a System Agent (SA) register, which can be indirectly accessed via I/O ports 0xa0/0xa4.

   0x10 - indicates URAM (small microcode SRAM). It is a private memory of a CPU core and is not shared by other cores.

   0x18 - indicates an 8-bit I/O port accessed on behalf of the microcode. Various Uncore devices such as Memory Controller or Memory Encryption Engine are connected to the I/O bus and controlled by the microcode through I/O ports.

   0x40 - indicates the Staging Buffer (special SRAM that is much larger than URAM). Access to it is shared by two adjacent CPU cores in the same CPU module (a block of two CPUs and one L2-cache slice

connected to the System Agent via the IDI - inter-die interconnect - bus in modern Atom processors) or even perhaps by all cores.

0x48 - indicates a 16-bit I/O port.

0x50 - indicates a 32-bit I/O port.

0x58 - indicates a 64-bit I/O port.

0x80 - indicates the Staging Buffer (at the moment, it is not quite clear in what way it differs from 0x40; perhaps, it allows for accessing an area reserved for another core).

rax - is a data address (specific to the data type defined by the rcx register).

The output of this instruction is as follows:

rdx - can contain up to 64 bits of read data (if the data type specified by the rcx register implies a data size of less than 64 bits, the high-order bits are zeros).

rbx - contains high-order 32 bits of read data that duplicate the high-order DWORD from rdx. For the 0x08 command, instead of the high-order DWORD, it contains an error code (read result) of an external bus because the System Agent registers are always 32-bit.

2. **udbgwr** - **0F 0F** - microarchitecture debug write

This instruction is to write microarchitectural data. It accepts the following arguments:

rcx - is a command that defines the type of data to write. The udbgwr instruction supports the same commands as the udbgrd instruction. In addition, udbgwr supports the following commands:

0xc8 - is to perform a System Agent register write. Unlike the 0x08 command, it allows setting an operation code (opcode) for the internal IOSF-SB bus of the System Agent (the 0x08 command uses the built-in codes of the 0x05 operation for reading and of the 0x07 operation for writing).

0xd0 - is to execute a write request on the internal IOSF-SB bus intended for power supply control (connected to the PCU - Power Control Unit).

0xd8 - is to invoke a microcode procedure using an arbitrary MSROM address. Microarchitectural registers tmp0 - tmp15 are transmitted via the Staging Buffer (can be written with the 0x40 command), starting from the address 0xb800 with an interval of 0x40. Architectural registers (except rcx and rax) are transmitted directly.

rax - is a data write address specific to the data type (command) defined in rcx. For the CRBUS, it is the address of a CPU unit, for an I/O port - the address of an I/O register, for URAM - the address of a memory location, and so on.

rdx:rbx - defines the data to be written. If the command implies a data size of less than 64 bits, the rbx value is ignored (this includes accessing System Agent registers with the 0x08 command). For the 0xc8 command, rbx contains the opcode for the System Agent bus.

The output of this instruction is as follows:

rbx - for commands accessing System Agent registers (0x80, 0xc8) - contains the write result (response code) read from the internal bus. In all other cases, the value of the initial register (at the instruction input) does not change.

rdx - for an IOSF-SB write command for the PCU - contains the internal bus response code. In all other cases, it contains the initial value.

3 Execution conditions for the udbgrd and udbgwr instructions

Using debugging mechanisms available via the Local Direct Access Test (LDAT) port (a special debug port for the CPU core's executive units, which is accessible via CRBUS registers), we found that the microarchitecture decoder processes the udbgrd and udbgwr instructions in all modes (User Mode, Kernel Mode, SMM, VMX Root, VMX Non-Root, SGX, and others). Namely, the IFU identifies the length of these instructions (2 bytes), and the complex decoder calls the corresponding MSROM entry points. In MSROM, the microcode for the discovered instructions looks as follows (the microcode and addresses are given for the Atom Goldmont core):

```
...

U028a: tmp3:= MOVEFROMCREG_DSZ64(0x2e6, 32)

U028c: BTUJNB_DIRECT_NOTTAKEN(tmp3, 0x00000009, generate_#UD)
U028d: tmp3:= MOVEFROMCREG_DSZ64(CTAP_CR_DFX_CTL_STS, 32)
U028e: BTUJB_DIRECT_NOTTAKEN(tmp3, 0x0000000a, generate_#UD)
       SEQW GOTO U27c9
...

udbgwr_xlat:
U0660: tmp2:= CONCAT_DSZ32(rbx, rdx)
U0661: tmp1:= MOVE_DSZ64(0x00000001)
U0662: tmp3:= MOVEFROMCREG_DSZ64(0x38c, 32)
       SEQW GOTO U0b5a
...

udbgrd_xlat:
U0b58: tmp1:= MOVE_DSZ64(0x00000000)
U0b59: tmp3:= MOVEFROMCREG_DSZ64(0x38c, 32)
U0b5a: tmp3:= NOTAND_DSZ32(tmp3, 0xa0000000)

U0b5c: UJMPCC_DIRECT_NOTTAKEN_CONDZ(tmp3, U028d)
U0b5d: tmp3:= READURAM(0x005c, 64)
U0b5e: BTUJB_DIRECT_NOTTAKEN(tmp3, 0x00000002, U028d)
       SEQW GOTO U028a
...

U27c9: BTUJB_DIRECT_NOTTAKEN(rcx, 0x00000005, U6bce)
U27ca: BTUJB_DIRECT_NOTTAKEN(rcx, 0x00000002, U27cc)
       SEQW GOTO U27cd
-------------------------------------------------------

U27cc: SAVEUIP_REGOVR(0x01, U27cd, 0x0005)
       SEQW GOTO U32cd
U27cd: tmp2:= CONCAT_DSZ32(rbx, rdx)
U27ce: tmp4:= NOTAND_DSZ32(0x00000001, rax)

U27d0: tmp5:= AND_DSZ32(0x000000c0, rcx)
U27d1: tmp5:= SHR_DSZ32(tmp5, 0x00000001)
U27d2: tmp6:= AND_DSZ32(0x00000018, rcx)

U27d4: tmp8:= OR_DSZ32(tmp6, tmp5)
U27d5: LFNCEMARK-> BTUJNB_DIRECT_NOTTAKEN(tmp1, 0x00000000, U27d6)
       SEQW GOTO U27d9
-------------------------------------------------------
U27d6: NOP

U27d8: tmp9:= ADD_DSZ32(tmp8, 0x00004052)
       SEQW GOTO U27da
-------------------------------------------------------
```

```
U27d9: tmp9:= ADD_DSZ32(tmp8, 0x00004392)
U27da: LFNCEWAIT-> UJMP(tmp9)
...

U4052: BTUJB_DIRECT_NOTTAKEN(rax, 0x0000000d, U5902)

U4054: tmp10:= MOVEFROMCREG_DSZ64(rax)
       SEQW GOTO U4065
...

U4065: rdx:= ZEROEXT_DSZ64(tmp10)
U4066: rbx:= SHR_DSZ64(tmp10, 0x00000020)
       SEQW GOTO U43a4
...

U43a4: LFNCEMARK-> MOVETOCREG_BTS_DSZ64(0x00000001, 0x289)
       SEQW GOTO uend
```

Listing 1: Implementation of udbgrd/udbgwr instructions in microcode

The microcode sample given above shows the implementation of just one of the commands (CRBUS read) for the discovered instructions, which starts at the address U4052. This research paper does not aim at describing the semantics of the microoperations (uops) listed above (partially, this is already done in the write-up of our Goldmont microcode disassembler tool [10]). Instead, we would like to demonstrate the conditions under which the udbgrd and udbgwr instructions will execute the given commands without throwing the #UD exception.

The entry point for the udbgrd instruction is the MSROM address U0b58, and for the udbgwr instruction - U0660. Yet, the main steps of performing the specified commands take place starting from the address U27c9. Besides, for both read and write operations, the target address in the microcode ROM is calculated: it indicates the location of the microcode that actually executes the read or write command. For the CRBUS read command, the code starts at the address U4052: it is calculated by adding the command identifier (at the address U27d8), which is passed to the rcx register before the execution of the udbgrd instruction, to the value 0x00004052. This also determines the way in which command identifiers are incremented - namely, with an increment of 0x08. For the CRBUS read command with the identifier 0x00, the target address is U4052, and control over the UJMP microoperation at U27da is transferred to this address. For write commands, the initial address for calculating the command handlers equals U4392 (at the address U27d9). In this way, the udbgrd and udbgwr instructions support several commands.

However, the main code that checks the conditions under which these instructions can be executed is located starting from the address U028a. As mentioned earlier, the udbgrd and udbgwr instructions to be executed are accepted by the CPU frontend in any processor operating mode, and it is the microcode located at the addresses starting from U028a that determines whether the instructions can be executed or not.

We can notice that, starting from the address U028a, the microcode checks two conditions that enable the execution of the instructions. The conditions are as follows:

1. Bit#9 (value 0x200) is set in the register at the CRBUS address 0x2e6.
2. Bit#10 (value 0x400) is not set in the *CTAP_CR_DFX_CTL_STS* register at the CRBUS address 0x285.

If either of these conditions is not met, the BTUJNB_DIRECT_NOTTAKEN/BTUJB_DIRECT_NOTTAKEN microoperations pass control to the *generate_#UD* subroutine, which eventually leads to the execution of the SIGEVENT microoperation that causes a handler of the invalid opcode exception (#UD) with the interrupt vector #0x06 to be invoked. The BTUJNB_* microoperations test the bit with the index that is defined as the second operand, and if it is set (or not set, depending on the type of the microoperation - B or NB), they transfer control to the microcode whose address is set in the third argument of the microoperation; otherwise, control passes to the microoperation that comes immediately after BTUJNB_*. Let's try to find out what the bits checked for these conditions mean. As mentioned earlier, the internal CRBUS of the processor core integrates all units (executive blocks) residing inside the IP block of the processor called Core, which in effect executes x86 instructions. Such units include: Instruction Fetch Unit (IFU), x86 decoders, Data Cache Unit (DCU, contains level-one data cache), Microcode Sequencer (MS), and others. On the CRBUS, each unit has a reserved range of addresses through which the unit's control registers are accessed. However, addresses are sometimes mixed up, and registers of different units can have adjacent addresses on the CRBUS. In Intel's in-house documentation (for example, in the system software .xml files), the name of each register has a prefix indicating the unit that the register belongs to. For example, the *CTAP_CR_DFX_CTL_STS* register belongs to the Test Access Port Controller (indicated here by CTAP), which is a unit controlling the processor's JTAG debug port.

We believe that the CRBUS register at the address 0x2e6 is not related to any specific hardware unit of the processor core, but rather belongs to a subset of CRBUS registers that are used for storing data and are saved to a special SRAM within the processor core. Microcode uses such registers to store execution time data, which can be stored independently from the computational core. Such CRBUS registers are prefixed with UCODE_CR_*. We do not know all the properties of such registers (for instance, whether they are readable or writable by other units connected to the CRBUS). Yet, what we do know is that microcode maps the value of the CRBUS register with the address 0x2e6 onto the MSR with the address 0x1e6. In this paper, we will not elaborate on how we found this correspondence. Suffice it to say that we obtained a complete table of all MSRs supported by the Goldmont core, including, for example, the following: processor modes in which an MSR is available; MSR type (that is, where the MSR is mapped to: URAM, CRBUS, or devices external to the core); address specific to the MSR type; MSROM entry points where MSR read and write operations actually take place. We also found that a write to MSR 0x1e6 causes a write of the specified value to the 0x2e6 register on the CRBUS (under a certain condition that we will describe later). So, Condition 1 for executing the instructions can be controlled by means of MSR 0x1e6.

However, judging by the microcode at the entry points (labels ending with _xlat) of the analyzed instructions, Condition 1 is not always required. As we can see, the microcode at the address U0b5c and address U0b5e allows skipping the check of Condition 1 and passing control right to Condition 2. At the address U0b5c, another CRBUS register - with the address 0x38c - is checked, while at the address U0b5e, bit #2 from the URAM location with the address 0x005c is checked. We do not know exactly what the register with the address 0x38c is called and what it controls. Yet we suppose that this register is related to the IFU, which fetches x86 instructions from the input stream of instructions to execute. We make such a conclusion based on the address range that this register is within as we know for sure that the register with the address 0x3c0 is an LDAT port of the IFU: via this port, we could access internal IFU arrays and see initial x86 instructions before they are decoded. If our guess is correct, then the 0x38c register can map either the value of fuses (one-time programmable memory) for the IFU, or special architectural states of the instruction input stream (for example, the presence of a special prefix or a certain sequence of x86 instructions). The microcode at the addresses U0b59 - U0b5c checks that bits #29 and #31 are both set in the 0x38c register. If they are, the check of Condition 1 can be skipped (that is, the check of the bit set via MSR 0x1e6). Therefore, we can expect that under certain conditions of the instruction input stream (at the moment, it is not clear exactly which ones), the udbgrd and udbgwr instructions may require no preliminary activation via MSR 0x1e6. Besides, if bit #2 of a 64-bit value written in URAM (private static memory storing microcode execution time data) at the address 0x005c is set, Condition 1 is also optional. We found that in the microcode associated with the handling of the Machine Check Exception (MCE), bit #2 in URAM at the address 0x005c is set if an occurring machine check error is related to internal CPU core units (unlike, for example, external device errors). This gives reason to think that the #MC handler (interrupt vector #0x12) allows using the udbgrd and udbgwr instructions without their prior activation via MSR 0x1e6.

Yet, the main condition for executing udbgrd and udbgwr is Condition 2. There is no way to bypass its check in microcode, so it must be met for the instructions to be successfully executed. Let's try to figure out what this condition implies. Empirically, we found that the *CTAP_CR_DFX_CTL_STS* register at the CRBUS address 0x285 is a hardware register. (We assigned this name to the register ourselves, by analogy with other CRBUS registers from various sources and on the basis of our assumption that this register controls the DFX (Designed for Testability, Debuggability, and Manageability) mechanisms of the CPU core, which are used for JTAG-based hardware debugging of Intel products.) This means that some bits are not writable by microcode, but rather are set at the hardware level. We know for sure that bit #11 (0x800) of this register controls CTAP (CPU core debug port connected to JTAG): when the bit is set, privileged JTAG commands are blocked (IR/DR scan chains that are available only in Red Unlock mode will return 0). Bit #10 (0x400), which is checked as part of Condition 2, cannot be modified by microcode; it is a status bit and is set at the hardware level. We found that bit #10 of the *CTAP_CR_DFX_CTL_STS* register gets the value of 0 (reset) when the processor debug unlock procedure (Red Unlock) is performed. Later in this paper, we will take a closer look at the Red Unlock mode and the ways in which the processor can be switched to it. And at this point, we would like to note that we cannot be one hundred percent sure that using Red Unlock is the only way to reset bit #10 of the *CTAP_CR_DFX_CTL_STS* register. Besides Red Unlock, there might be some other conditions under which bit #10 gets the value of 0, making it possible to execute the udbgrd an udbgwr instructions. (For instance, we saw that, for certain tasks, microcode itself makes a write request to the *CTAP_CR_DFX_CTL_STS* register, although to bits #0 and #1.) Therefore, for now, we can only be sure that in the Red Unlocked state, Condition 2 is fulfilled, and the udbgrd and udbgwr instructions are unlocked. We made a request to Intel for a complete description of all bits of the *CTAP_CR_DFX_CTL_STS* register in order to have some degree of certainty that Red Unlock is the only way to activate the udbgrd and udbgwr instructions, but we haven't received a reply yet. It is important to note that the microcode procedure that handles writing to the MSR (x86 wrmsr instruction) at the address 0x1e6 (for Condition 1) also checks whether bit #10 of the *CTAP_CR_DFX_CTL_STS* register is reset; if the bit is set, the wrmsr instruction for this address generates a general protection exception (#GP). This gives reason to suspect that bit #10 of the *CTAP_CR_DFX_CTL_STS* register indicates the DFX locked state (has name *DfxPolicyLocked*) of the CPU core.

To sum up, the udbgrd and udbgwr instructions are activated through bit #9 (value 0x200) in MSR 0x1e6, and can be executed if the processor is in the Red Unlocked state. Besides MSR 0x1e6, there are other ways to activate the instructions for a processor in the Red Unlocked state - for example, automatic activation via the #MC handler. We cannot guarantee that Red Unlock is the only condition that allows executing the udbgrd and udbgwr instructions and there is no other way - whether software- or hardware-based - to put the processor into a state in which these instructions can be activated and executed.

4 How we found the udbgrd and udbgwr instructions and for which processor models

In mid-2020, our team managed to extract microcode for modern Atom processors that are based on the Goldmont microarchitecture. It became possible to do this on Atom Goldmont systems-on-chip (SoCs) due to an arbitrary code execution vulnerability in Intel CSME (Intel-SA-00086). The vulnerability allowed us to enable the Red Unlock mode on such SoCs not only for the chipset (PCH), but also for the CPU: on SoCs, hardware components that control the platform's unlocked states are shared between IP blocks of the system logic and CPU units, which include the x86 execution core. As soon as we realized that we had achieved Red Unlock for the CPU on the Intel Apollo Lake platform, we managed to restore a number of internal JTAG registers whose IR codes (IR, or Instruction Register is a term used in the JTAG standard [11]) are absent from publicly available versions of Intel DAL and OpenIPC software packages for Intel microchip hardware debugging. With these IR codes, it is possible to access the Control Register Bus (CRBUS) of the CPU core. Once we were able to access the CRBUS, we found registers of the internal unit called Microcode Sequencer (MS). By using MS registers to access the MS LDAT (Local Direct Access Test) port, we managed to extract all five internal arrays (of the MS internal memory). One of them was MSROM - read-only memory that stores microcode of all x86 instructions implemented via microcode assist (some instructions are decoded directly, bypassing MSROM) as well as CPU initialization code. Then we spent a lot of time on reverse-engineering the extracted microcode to understand the format of microoperations and their purpose. We also started developing our own microcode disassembler for the Atom Goldmont core in order to study the microarchitecture of modern Intel processors and x86 instructions implementation. To date, we have already published our Atom microcode disassembler tool [12], so that the infosec community can get access to the x86 implementation in modern Intel processors and also check our arguments independently. When analyzing the microcode listing obtained by our disassembler, we noticed an interesting fragment. In microcode, it looks as follows:

```
U30fd: tmp6:= LDSTGBUF_DSZ64_ASZ16_SC1(0x0000b980)
U30fe: tmp7:= LDSTGBUF_DSZ64_ASZ16_SC1(0x0000b9c0)

U3100: tmp8:= LDSTGBUF_DSZ64_ASZ16_SC1(0x0000ba00)
U3101: tmp9:= LDSTGBUF_DSZ64_ASZ16_SC1(0x0000ba40)
U3102: tmp10:= LDSTGBUF_DSZ64_ASZ16_SC1(0x0000ba80)

U3104: tmp11:= LDSTGBUF_DSZ64_ASZ16_SC1(0x0000bac0)
U3105: tmp12:= LDSTGBUF_DSZ64_ASZ16_SC1(0x0000bb00)
U3106: tmp13:= LDSTGBUF_DSZ64_ASZ16_SC1(0x0000bb40)

U3108: tmp14:= LDSTGBUF_DSZ64_ASZ16_SC1(0x0000bb80)
U3109: tmp15:= LDSTGBUF_DSZ64_ASZ16_SC1(0x0000bbc0)
U310a: LFNCEMARK-> SAVEUIP(0x00, U21fe)

U310c: SAVEUIP(0x01, U21fe)
U310d: NOP
U310e: LFNCEWAIT-> UJMP(rax)
```

Listing 2: Microcode fragment calling arbitrary routine in microcode

It seemed rather strange to us that the UJMP(rax) microoperation (in binary format: 0x015d00000800) was there, as this meant that a certain microcode fragment allowed transferring control to an arbitrary address in MSROM - an address determined by an **architectural** register. We had no doubts that we identified the opcode for the UJMP microoperation (0x15d) correctly because this was proven empirically: using the match/patch mechanisms, we could execute arbitrary microcode when calling any x86 instruction implemented by means of MSROM-assist. Also, we were certain that the argument of the microoperation was in the rax register (the 0x20 selector in the *src1* field). All that pointed to the existence of an x86 instruction that allows calling an arbitrary address in MSROM! It is noteworthy that, before control passes to the specified address in microcode, the microarchitectural state (tmpx registers) is read from the staging buffer (static memory shared by all CPU cores). This is because the tmpx registers cannot be set on the x86 architecture level, but are actively used by microcode. So, we were even more convinced that we had found a mechanism to call an

arbitrary subroutine in MSROM. When looking for references to this microcode fragment throughout MSROM, we found the beginning of the fragment, but there were no direct references to it in microcode (in either conditional or unconditional operations to pass control to an address in MSROM, in which the target address is specified directly in the microoperation). Here it is:

```
U440a: tmp0:= LDSTGBUF_DSZ64_ASZ16_SC1(0x0000b800)

U440c: tmp1:= LDSTGBUF_DSZ64_ASZ16_SC1(0x0000b840)
U440d: tmp2:= LDSTGBUF_DSZ64_ASZ16_SC1(0x0000b880)
U440e: tmp3:= LDSTGBUF_DSZ64_ASZ16_SC1(0x0000b8c0)

U4410: tmp4:= LDSTGBUF_DSZ64_ASZ16_SC1(0x0000b900)
U4411: tmp5:= LDSTGBUF_DSZ64_ASZ16_SC1(0x0000b940)
       SEQW GOTO U30fd
```

Listing 3: Beginning of fragment for calling arbitrary microcode routine

As we can see, the complete set of microarchitectural registers is read from the staging buffer before control is transferred. We were certain that, although there were no direct references to this microcode fragment, its very presence suggested that control should pass to it in one way or another. Otherwise, we wouldn't be able to logically explain its existence. We theorized that there is an MSROM area that indirectly transfers control to this fragment, possibly using arithmetic operations to calculate the target address in microcode. To prove this, we began looking for all 16-bit constants used in arithmetic addition or subtraction microoperations that could result in the value 0x0000440a. Eventually, we found the following set of microoperations in microcode:

```
...
U27d9: tmp9:= ADD_DSZ32(tmp8, 0x00004392)
U27da: LFNCEWAIT-> UJMP(tmp9)
```

Listing 4: Indirect reference to fragment at U440a

When analyzing the microcode located at the addresses preceding U27d9, we found that the tmp8 register can get the value of 0x78 (0x0000440a – 0x00004392), which is necessary for transferring control to the required address. We also examined the logic of the tmp8 register value generation and found that, apart from the ability to call an arbitrary MSROM address, **there is a whole set of subroutines allowing both read and write access to various microarchitectural data**. In particular, we paid attention to CRBUS access: via this bus, it is possible to control the match/patch registers and Patch RAM in Microcode Sequencer. It is precisely these mechanisms that allow modifying arbitrary MSROM areas with microcode patch data. The whole microcode block that enables the calling of these subroutines is provided in the report section that describes conditions required for executing the undocumented udbgrd and udbgwr instructions. And here we'd like to highlight the following: when analyzing this block for calling various subroutines in microcode, we found two MSROM entry points for x86 instructions that involve this mechanism. At that point, we already had an understanding of how to tell the difference between a microcode entry point for an x86 instruction and the microcode associated with, for example, CPU initialization. Entry points for x86 instructions are located in MSROM at low-order addresses, ranging from U0000 to U1000; each address must be a multiple of 8 and there must be no references to it from other microcode areas in MSROM. Thus, we found two entry points for unknown x86 instructions (as described earlier, one of them is for writing data and the other is for reading data):

```
udbgwr_xlat:
U0660: tmp2:= CONCAT_DSZ32(rbx, rdx)
U0661: tmp1:= MOVE_DSZ64(0x00000001)
U0662: tmp3:= MOVEFROMCREG_DSZ64(0x38c, 32)
       SEQW GOTO U0b5a
...
udbgrd_xlat:
U0b58: tmp1:= MOVE_DSZ64(0x00000000)
U0b59: tmp3:= MOVEFROMCREG_DSZ64(0x38c, 32)
U0b5a: tmp3:= NOTAND_DSZ32(tmp3, 0xa0000000)

U0b5c: UJMPCC_DIRECT_NOTTAKEN_CONDZ(tmp3, U028d)
U0b5d: tmp3:= READURAM(0x005c, 64)
U0b5e: BTUJB_DIRECT_NOTTAKEN(tmp3, 0x00000002, U028d)
       SEQW GOTO U028a
```

Listing 5: Entry points in microcode for udbgrd/udbgwr instructions

Our disassembler supports microcode labels so that an arbitrary address can be assigned a text label to make its representation in the microcode listing more convenient. We decided to assign names to all labels with the *_xlat* ending related to x86 instruction entry points in order to differentiate them from other labels that describe various subroutines in microcode. In the microarchitecture of modern processors, XLAT (short for "Translate") denotes a mechanism that ensures static conversion of an x86 instruction opcode to an MSROM address at which the entry point for the instruction implementation is located in microcode. This name reflects the tabular nature of the mechanism, which is used by the complex decoder to handle x86 instructions implemented by means of MSROM-assist (that is, most of x86 instructions).

Unfortunately, we haven't found XLAT tables among various microarchitectural arrays that allow read/write access via JTAG/CRBUS for the Atom Goldmont core. This data is crucial because it will enable us to establish one-to-one correspondence between MSROM entry points and x86 instructions (their opcodes). But we are yet to obtain it. The only solution that we could think of in order to find x86 instructions for the necessary MSROM entry points was fuzzing of all x86 opcodes (in other words, their enumeration).

Searching for undocumented instructions by means of x86 fuzzing is an extremely difficult task [13]. Besides, the classic fuzzing technique has a weakness: it does not help to differentiate between a non-existent x86 opcode (that causes a #UD exception) and, for example, an invalid format of the instruction or incorrect processor execution mode in which the instruction could be performed. Yet we had an advantage - reliable feedback from the microarchitecture. So what we did was to configure the match/patch mechanism to intercept the necessary MSROM entry points for unknown x86 instructions, initiate the enumeration of all possible x86 opcodes, and then wait for the mechanism to be triggered. This allowed us to significantly improve the enumeration of x86 opcodes: we could send up to 256 instructions to be executed at a time, and in case of a #UD exception, we would just re-send the remaining instructions from the set. And luck was on our side: when analyzing instructions with two-byte opcodes, our mechanism was triggered for the 0x0f 0x0e and 0x0f 0x0f instructions: the former for the read operation entry point, and the latter for the write operation entry point. We found undocumented x86 instructions! Of course, it then took us some time to understand the execution conditions of these instructions and write the actual proof-of-concept (PoC) code that would work without physically connecting a hardware debugger, but all that was of secondary importance. We integrated our PoC code for Red Unlock on the Apollo Lake platform [9] and a PoC code for read/write access to the CRBUS using udbgrd/udbgwr [14], and managed to obtain local access to the CRBUS without connecting to the platform via JTAG. This makes the Red Unlock mode extremely important in terms of platform security because previously it could pose a threat only in case of physical connection via XDP/DCI.

Of course, one of the first concerns that we had was whether these instructions exist only in the x86 subset supported by modern Atom CPUs, or they are implemented in all present-day Intel CPUs. Since the Red Unlock mode itself has been present in all Intel processors for quite some time (at least, starting from Ivy Bridge processors and newer, which we can prove by referring to .xml files from the Intel DAL hardware debugger library), we suspect that the instructions that we found are also supported on Intel desktop processors. After posting the initial results of our work on Twitter, we managed to obtain indirect evidence that these undocumented instructions can be found on other, non-Atom, CPUs [15], [16]. All in all, in terms of supported x86 instructions, modern Atom CPUs are not much different from the latest CPUs of Intel's main product line (they support all instruction sets except AVX extensions, such as VMX, SGX, and MPX). Based on that, we can make a bold assumption that the instructions we found are an undocumented part of the x86 architecture as such, meaning that they are supported by all present-day Intel processors.

At the moment, we can only prove that the instructions are present on the Apollo Lake and Gemini Lake platforms (based on the Atom Goldmont and Goldmont Plus processors) - we have the PoC to perform Red Unlock on these platforms. However, we are strongly determined to soon achieve Red Unlock on one of the most recent desktop platforms as we have all the prerequisites for that.

Here is another argument that backs up our assumption that the instructions are supported on processors of Intel's main product line: Intel aims at manufacturing hybrid processor architectures so that some cores on a chip belong to the Intel Atom processor family while others belong to "big" CPU cores of the Intel Core family (for example, Alder Lake SoCs).

The next section discusses the Intel processor debug mode, Red Unlock: we will look at possible ways to enable it and the debugging capabilities that it provides to Intel engineers (and, as it turns out, not only to them).

5 Red Unlock (Intel Unlock) mode

The JTAG (IEEE 1149.1 [11]) standard has been implemented in almost all modern chips produced by various manufacturers. These days, it's hard to find a microcontroller, FPGA chip, or any other programmable logic microchip

that doesn't support JTAG to that or another extent. Originally designed to test connections of printed circuit boards, JTAG quickly became used for the integrated logic (microchips) due to its simplicity, flexibility, and scalability. Although the standard itself describes only commands (IR codes) to test the internal logic using external connection signals (of contacts on the chip), manufacturers started extending it (thanks to its potential to do so) by adding a great number of their own IR codes for a whole range of different tasks. Intel was not an exception to that, and today all Intel processors and system logic microchips support JTAG and its extensions to provide various management, debugging, and verification capabilities, both at the chip level and at the level of external elements. Based on JTAG, Intel has created a whole infrastructure for post-silicon verification of its chips (as opposed to the pre-silicon verification, which is implemented in software emulators). Beginning with Pentium processors, the presence of such infrastructure for microchip testing was considered by Intel to be one of its key achievements that would define the company's success in the near future (see [17]). It won't be an exaggeration to say that nowadays almost all Intel IP blocks (Intelligent Property blocks are hardware components that make up the final product on a chip, be it a processor, SoC, or system logic microchip) can operate under JTAG control, which allows reading and writing the majority of the most significant RTL logic registers. This JTAG-based testing and debugging infrastructure, also known as DFX (Designed for Testability, Debuggability, and Manageability), covers almost all Intel silicon products. At the core of DFX is a system of hierarchical integration of IP blocks (TAPC), which is a JTAG extension (the standard itself describes only consecutive integration of components). With TAPC, it is possible to connect to any IP block within the chip (that is, to get access to TMS, TDI, and TDO signals), regardless of how deep the block is in terms of the internal interaction hierarchy.

To appreciate the level of debugging capabilities provided by DFX, let us treat the core of modern Intel processors that support x86 instructions as a single IP block connected to the JTAG chain by means of TAPC. Then, it will be suffice to say that DFX allows comparing the internal microarchitectural state (of the so-called arrays) - while the core is performing common tasks on the chip - with software emulation models, in order to identify logic errors implemented in the register transfer level (RTL) logic. (At the RTL level, hardware is designed using a hardware definition language (HDL), such as Verilog.) Apart from post-silicon validation, DFX also implements hardware debugging of the system software, including various types of firmware run on all sorts of microcontrollers and microprocessors on chips manufactured by Intel.

Considering the depth of penetration to the hardware internals, DFX must be protected from unauthorized use. For instance, it is possible to leverage DFX to get read or write access to the Real Register File (RRF - for details, see the section on speculative execution of udbgrd/udbgwr) during the CPU core execution, and then change the final result of executing any dependency chain of x86 instructions. This allows bypassing any protection mechanisms used in x86 (for example, memory protection). However, the JTAG standard does not provide for any built-in protection. For this reason, manufacturers take the initiative by implementing various mechanisms to prevent unauthorized use of JTAG by third parties. For the most part, such mechanisms completely lock JTAG in end products that have passed the validation and testing stages of production. However, Intel decided to do it differently: instead of locking DFX in end products, it implemented a hierarchy of DFX access and unlock levels in its final products to be shipped to customers. In this way, Intel engineers could debug equipment whenever it would fail and be returned after purchase. This undoubtedly has improved the quality of the manufactured chips. Yet, the reverse side of this approach is reduced security. If something can be exploited, it doesn't take long before someone attempts to do it for illegitimate purposes. So, Intel decided to choose a protection strategy known as "security through obscurity." It means that there is almost no publicly available information about how DFX is implemented (except for a few rather basic patents) and what risks it entails for end users. Moreover, up to a certain moment, the very fact of the existence of such a debugging mechanism was only known to system engineers who were obliged to sign a non-disclosure agreement (NDA) with Intel.

Now, let's look at how the DFX protection is implemented in modern Intel processors and chipsets. DFX defines several levels of access to the debugging capabilities it is based on. The levels are intended for:

1. Intel engineers (Intel Unlock or Red Unlock)
2. Original equipment manufacturers (OEM Unlock or Orange Unlock)
3. Other users (Green Locked)

Clearly, the Green Locked mode provides minimum debugging capabilities, while the Red Unlock mode ensures full access to all DFX mechanisms. The Green Locked mode allows only standard debugging of x86 code (Run Control), namely, performing a core halt and changing the architectural state. It won't allow "deeper" access (for example, to the core's microarchitectural state), and any attempts to execute IR codes of privileged commands will be rejected. This report will not consider all the DFX debugging capabilities (as they are numerous) or their availability in different unlock modes. The main goal of the report is to show the ways to activate the highest-level mode, Red Unlock, because we know for sure that it is this mode (rather than Orange Unlock) that allows executing the undocumented instructions

to access the microarchitecture.

We know that there are four methods to activate Red Unlock on modern Intel processors:

- Hardware strap (only for pre-production chips)
- Special fuse
- JTAG password
- Software-based method via the *PERSONALITY* register of the DFX AGGREGATOR

Let's take a closer look at each method.

5.1 Hardware strap (only for pre-production chips)

Without doubt, Intel chips have a great variety of configuration parameters, such as external interface settings, operation modes of integrated devices, power supply settings, and much more. Within one generation, all models of system logic microchips are essentially the same chips but with different configurations set at the final production stages. Some configuration parameters are hard-coded via one-time programmable (OTP) memory by Intel, while others are to be set by end-product manufacturers (OEMs or ODMs). In turn, OEM parameters are divided into three groups: OEM OTP, software straps, and hardware straps. For software straps, there is a special SPI image region called Flash Descriptor. Hardware straps are set using hardware signals (sent to microchip entries), which join either the common signal pathway (Ground) or positive power supply voltage via pull-up or pull-down resistors. Intel chip specifications (datasheets) provide data on several hardware straps: their purpose, microchip contact, and electric type (pull-up or pull-down activation). However, official datasheets do not list all the configuration parameters set at the hardware level. Manufacturers of equipment based on Intel chips (for example, motherboards or laptops) release their own documentation for repair service centers in the form of service manuals. Such manuals describe the device circuit design. They also contain connection schemes for Intel chips and can even describe hardware straps that are absent from Intel datasheets. When looking through a service manual of this kind, we found a peculiar description of two hardware configuration parameters. Here it is:

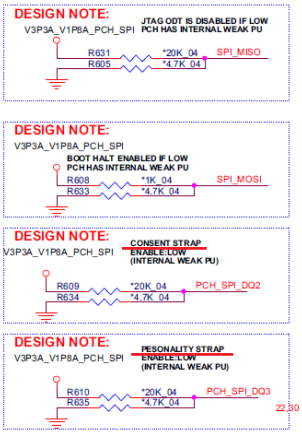

Figure 1: Undocumented hardware straps for PCH from Service Manual

The names *CONSENT* and *PERSONALITY* (misspelled in the diagram) immediately drew our attention because by that time we already knew that these are the names of two registers of a special device called DFX AGGREGATOR (DFX AGG). Through these registers, software-based DFX unlocking is performed (described later in this section). It is used by the Intel CSME firmware (the code is integrated into the CSME ROM). However, we hadn't found the Intel-SA-00086 vulnerability by then, so we couldn't perform arbitrary writes to these registers. Therefore, any opportunity to affect the registers, even through hardware straps, seemed to be quite attractive.

We enabled the specified straps via pull-down resistors (by means of ground connection) and found that the *CONSENT* register of DFX AGG changed its value to 1. (This register controls writing to *PERSONALITY*. If its value is 1, writing is allowed. The default value is 0.) However, the *PERSONALITY* register retained its zero value, and our actions had no

effect in terms of DFX unlocking. We found a solution a bit later, when we learnt about the VISA hardware signal tracing technology and looked at visa.xml files, which describe a subset of all existing hardware signals to be analyzed with VISA. More specifically, we found this description of a signal group:

```
<group clk="../dfx_agg/i_dfx_agg_dlcs/side_clk" name="../dfx_visa_unit_mux_2" security="0">
     <mux_path clk_sel="0" lane_sel="2" mux_name="../dfx_visa_unit_mux" />
     <mux_path clk_sel="14" lane_sel="15" mux_name="../dfx_mas_clm/visa_clm" />
     <signal bit="0" name="../dfx_agg/i_dfx_agg_dlcs/visa_sb_a0_debug_strap" />
     <signal bit="1" name="../dfx_agg/i_dfx_agg_dlcs/visa_IP_Request" />
     <signal bit="2" name="../dfx_agg/i_dfx_agg_dlcs/visa_fuse_pull_done" />
     <signal bit="3" name="../dfx_agg/i_dfx_agg_dlcs/visa_fuse_download_done" />
     <signal bit="4" name="../dfx_agg/i_dfx_agg_dlcs/visa_decode_start" />
     <signal bit="5" name="../dfx_agg/i_dfx_agg_dlcs/visa_decode_done" />
     <signal bit="6" name="../dfx_agg/i_dfx_agg_dlcs/visa_decode_error" />
     <signal bit="7" name="../dfx_agg/i_dfx_agg_dlcs/visa_fuse_sense_error" />
</group>
```

Listing 6: Hardware signals of DFX AGG describing OTP processing

These signals belong to the DFX AGG device (see dfx_agg in the signal paths), and the prefix a0 in the name of the *visa_sb_a0_debug_strap* signal clearly suggests that the hardware strap that we found for *PERSONALITY* is interpreted only for engineering sample (pre-production) processors with the A0 stepping. Apparently, there is a special OTP bit that is always set for chips with a stepping other than A0 so that configuration intended only for the A0 versions is ignored.

5.2 Special fuse

We suppose that for modern Intel processors starting with the Cannon Lake generation, Red Unlock can be implemented on a per-processor basis at the production stage - namely, during the programming of the chip's OTP configuration by "blowing" the appropriate fuses. (EFUSE, or Electronic Fuse, is a method of implementing OTP memory used for modern Intel processors. It is based on fusing of conductors at the moment of writing.) Such a bold assumption can be backed up by a full description of the DFX status register that we found in .xml configuration files of the DAL hardware debugger for Cannon Lake processors. Here it is:

```
<State _name="STATUS_CSR_UPPER_MAP(31:0)">
         <Field _map="13:13" _name="PULLER_ERROR" />
         <Field _map="12:10" _name="PULLER_TYPE" />
         <Field _map="9:9" _name="DECODER_DONE" />
         <Field _map="8:8" _name="DECODER_ERROR" />
         <Field _map="7:7" _name="ENABLE_DECODER" />
         <Field _map="6:6" _name="FUSE_SENSE_ERROR" />
         <Field _map="5:5" _name="ORANGE_UNLOCK" />
         <Field _map="4:4" _name="RED_OR_METAL_UNLOCK" />
         <Field _map="3:3" _name="RED_FUSE_ENABLE" />
         <Field _map="2:2" _name="LEGACY_FUSE_DISABLE" />
         <Field _map="1:1" _name="ORANGE_FUSE_ENABLE" />
         <Field _map="0:0" _name="FUSE_DOWNLOAD_DONE" />
</State>
```

Listing 7: High-order bits of DFX AGG STATUS register

The status register belongs to the DFX AGG device. This IP block is present on all modern processors, chipsets, and SoCs manufactured by Intel. It is DFX AGG that controls the state of DFX unlocking on these chips. It ensures authorized execution of DFX unlocking (by Intel engineers or by OEMs) and secure transmission of the unlock status to other IP blocks on the chip. During a hardware reset, DFX AGG accesses a special device called FUSE CONTROLLER via the IOSF-SB bus and reads the OTP configuration using special FUSE PULL requests. The description of the DFX status register bits suggests that one of the bits in the OTP configuration is *RED_OR_METAL_UNLOCK*. This bit permanently enables Red Unlock. We believe that it is actually a fuse, rather than the unlock status, because the unlock status is displayed in other fields of the DFX status register (in the low-order DWORD), while the high-order DWORD shown in the picture above consolidates data on the OTP configuration and its processing by DFX AGG. Presumably, this fuse can be enabled only for engineering samples (pre-production parts). However, we are convinced that pre-production chips (with the A0 stepping) of both processors and chipsets differ from production versions only in their OTP configurations (leaving aside differences at the hardware logic level, which result from fixing errors found

during post-silicon testing of samples with a stepping other than A0). Moreover, it is not a single fuse, but rather a number of fuses that block various features of engineering samples. For example, Intel CSME has a set of fuses that are independent from DFX AGG and block mechanisms available in pre-production versions (debug encryption keys, ROM bypass, and other). This gives reason to suspect that by using fuses, it is technically feasible to enable Red Unlock for production chips too, provided that the corresponding bit is set in the chip's factory-programmed OTP configuration (because it can be processed independently from other bits in DFX AGG).

5.3 JTAG password

Besides OTP configuration, DFX AGG allows enabling Red Unlock via JTAG using special IR commands. As it turned out, .xml files of publicly available DAL versions that describe JTAG registers for DFX AGG contain no information about the IR codes and JTAG registers that can be used for DFX unlocking. However, we also looked through visa.xml files for various platforms: they describe hardware signals of IP blocks whose state can be traced in real time using the VISA hardware tracing system. In these files, we found the following internal signals belonging to the IP block of DFX AGG:

```
<group clk="../i_dfx_agg_dlcs/side_clk" name="../dfx_visa_unit_mux_1" security="0">
     <mux_path clk_sel="0" lane_sel="1" mux_name="../dfx_visa_unit_mux" />
     <mux_path clk_sel="14" lane_sel="15" mux_name="../dfx_mas_clm/visa_clm" />
     <signal bit="0" name="../dfx_agg/i_dfx_agg_dlcs/visa_Fuse_valid" />
     <signal bit="1" name="../dfx_agg/i_dfx_agg_dlcs/visa_DLCS_value[0]" />
     <signal bit="2" name="../dfx_agg/i_dfx_agg_dlcs/visa_DLCS_value[1]" />
     <signal bit="3" name="../dfx_agg/i_dfx_agg_dlcs/visa_RedPasswdEnable" />
     <signal bit="4" name="../dfx_agg/i_dfx_agg_dlcs/visa_OrangePasswdEnable" />
     <signal bit="5" name="../dfx_agg/i_dfx_agg_dlcs/visa_MetalPasswdDisable" />
     <signal bit="6" name="../dfx_agg/i_dfx_agg_dlcs/visa_DecommissionEnable" />
     <signal bit="7" name="../dfx_agg/i_dfx_agg_dlcs/visa_Mfg_Exit" />
</group>
```

Listing 8: Hardware signals of DFX AGG describing OTP configuration

If we compare this description of several DFX AGG hardware signals with the description of DFX status register bits, it will be clear that both describe the same data (cf. *visa_RedPasswdEnable* and *RED_FUSE_ENABLE*, *visa_MetalPasswdDisable* and *LEGACY_FUSE_DISABLE*). Moreover, we can identify another way to perform DFX unlocking of all three types - namely, use of passwords for RED, ORANGE, and METAL (or LEGACY) unlock. Intuitively we understood that those should be hardware passwords to be entered to the chip's logic via JTAG. However, we still needed IR codes for the registers that forward these passwords to DFX AGG, and the .xml files describing DFX AGG registers for JTAG didn't provide that information. So, we looked into visa.xml files once again and found the necessary information - namely, this description of a signal group:

```
<group clk="../dfx_agg/ftap_tck" name="../dfx_visa_unit_mux_52" security="0">
    <mux_path clk_sel="2" lane_sel="52" mux_name="../dfx_visa_unit_mux" />
    <mux_path clk_sel="14" lane_sel="15" mux_name="../dfx_mas_clm/visa_clm" />
    <signal bit="0" name="../i_dfx_agg_phtmcrk_tapreg/visa_instr_RedUnlock" />
    <signal bit="1" name="../i_dfx_agg_phtmcrk_tapreg/visa_instr_OrangeUnlock" />
    <signal bit="2" name="../i_dfx_agg_phtmcrk_tapreg/visa_instr_MetalUnlock" />
    <signal bit="3" name="../i_dfx_agg_phtmcrk_tapreg/visa_instr_UniqueID" />
    <signal bit="4" name="../i_dfx_agg_phtmcrk_tapreg/visa_UnlockCaptureDr" />
    <signal bit="5" name="../i_dfx_agg_phtmcrk_tapreg/visa_UnlockShiftDr" />
    <signal bit="6" name="../i_dfx_agg_phtmcrk_tapreg/visa_UnlockUpdateDr" />
    <signal bit="7" name="../i_dfx_agg_phtmcrk_tapreg/visa_PC_write_once" />
</group>
```

Listing 9: Hardware signals of JTAG commands for DFX unlocking

This data proved our assumption that the unlock passwords are entered to special JTAG registers of DFX AGG. But most importantly, it contained the JTAG *UniqueID* register that we already saw in a DAL .xml file. Here it is:

```
<TapCommand Ir="0x40" Register="READ0" TapRegister="PHTM_CRK_UNIQUE_ID"... />
```

Listing 10: JTAG UniqueID register description

We theorized that the password registers in question are located starting from the IR code 0x41 because the .xml file with the description contained an interrupt between IR codes from 0x41 to 0x44. By tracing the above-mentioned VISA

signals in real time and enumerating IR codes in IR/DR scan chains via JTAG, we managed to restore all the three registers for entering unlock passwords (spotting the right IR was indicated by a corresponding VISA signal):

```
<TapCommand Ir="0x41" Register="WRITE" TapRegister="ORANGE_UNLOCK_PASSWD".../>
<TapCommand Ir="0x42" Register="WRITE" TapRegister="RED_UNLOCK_PASSWD".../>
<TapCommand Ir="0x43" Register="WRITE" TapRegister="METAL_UNLOCK_PASSWD" .../>
```

Listing 11: JTAG registers for DFX unlocking

The next vital step was to get the binary format of these registers. Without going into detail about how we obtained it, here is the format of the DFX unlock registers:

```
<TapRegister _indices="63:0" _map="" _name="ORANGE_UNLOCK_PASSWD">
    <Field _map="63:0" _name="DR_PASSWORD" />
</TapRegister>
<TapRegister _indices="63:0" _map="" _name="RED_UNLOCK_PASSWD">
    <Field _map="63:0" _name="DR_PASSWORD" />
</TapRegister>
<TapRegister _indices="31:0" _map="" _name="METAL_UNLOCK_PASSWD">
    <Field _map="31:0" _name="DR_PASSWORD" />
</TapRegister>
```

Listing 12: DFX unlocking JTAG registers bit layout

Anticipating questions with regard to Metal Unlock, we'd like to clarify that, originally, Intel chips seem to have had only one JTAG unlock password to access the most important functions. This password was not unique for each specific processor, chipset, or SoC, but rather was shared across all chips of the same model. In some versions of the Intel System Studio software package, configuration files still contain such passwords for a number of legacy platforms. For newer platforms, the initial password values for both Red Unlock and Orange Unlock - along with the *UniqueID* value - are all stored in DFX AGG fuses and are unique for each product sample. At the final production stage, unique fuses are coded for every single chip. (Intel maintains a database with passwords of all manufactured chips.) However, for some reason, each chip also has an inherited, non-unique password (at the die level). Its use can be prohibited by a certain OTP bit of DFX AGG (it was prohibited on all platforms that we've analyzed). In fact, Metal Unlock presupposes the use of a legacy password, and this type of unlock is a functional equivalent of Red Unlock. Yet the Metal Unlock password is half as long (32 bits) as the Red Unlock and Orange Unlock passwords. We suspect that Metal Unlock is used for initial chip testing (before the chip's OTP configuration is programmed) and that the fixed password is blocked during that production OTP programming.

When using the above-mentioned VISA signals tracing to search for the IR codes of the JTAG registers for unlock passwords, we discovered an important thing: the JTAG DR scan chain did not pass the UpdateDr stage (see *visa_UnlockUpdateDr* in the group of signals shown in 9). This could only mean that a password, although put in the DR register as a result of IR/DR scanning, was not really interpreted by the equipment. Further analysis revealed that a JTAG password can be entered only at a certain stage of platform loading. This stage is called Early Boot Debug Window. To bring the platform to this stage, you need to install the so-called Boot Halt mode using hardware debugging tools. If you turn on the platform in Boot Halt mode activated by means of JTAG tools, normal loading will be blocked and DFX AGG will wait for a hardware password to be entered. It is even possible to proceed with loading without entering a password, which can be done using JTAG registers of DFX AGG. What is more important, you can also tell the platform that a password will be entered later, and, if so, the platform will start loading in Delayed Authentication Mode (DAM). This paper will not elaborate on DAM; note, however, that when DAM is enabled, hardware passwords for both Orange and Red Unlock can be applied at any time.

The following example demonstrates an attempt to enter a hardware password in the command-line interface (CLI) of the Intel DAL software package in order to activate Red Unlock for a Cannon Lake processor (a recent CPU platform). Since the password is incorrect, the unlock attempt fails:

```
>>> itp.holdhook(0, 3, True) #Enable Boot Halt
>>> DCI: Target connection has been fully established
       Note: Configuration on all debug ports has started
       Target Configuration: CNL_CNP_OpenDCI_CCA_[DbC]_ReferenceSettings
       Warn: No CPU power detected on pod 0, skip detecting tap devices on this pod!
       Note: Configuration on all debug ports has finished
       Note: Power Loss occurred
```

```
>>> itp.chipsets["CNP_SECAGG0"].state.tap.STATUS_CSR
STATUS_CSR(63:0) = 0x0000020D0000011A
  UPPER_MAP_(31:0) = 0x0000020D
  DFX_SECURE_POLICY(15:0) = 0x0000
  PERSONALITY_VALUE(4:0) = 0x00
  DLCS_VALUE(2:0) = 0x2
  EARLY_BOOT_DONE = 0x0
  PERSONALITY_LOCK = 0x0
  BOOT_HALT_STATUS = 0x1
  POLICY_UPDATE = 0x1
  POLICY_IS_FINAL = 0x0
  EARLY_BOOT_DEBUG_WINDOW = 0x1
  DELAY_AUTH_MODE_STATUS = 0x0
>>> itp.chipsets["CNP_SECAGG0"].state.tap.BOOTCTRL_CSR.DAM_ENABLE=1
>>> itp.chipsets["CNP_SECAGG0"].state.tap.BOOTCTRL_CSR.RESUME_BIT0=1
>>> itp.chipsets["CNP_SECAGG0"].state.tap.BOOTCTRL_CSR.RESUME_BIT1=1
>>> itp.chipsets["CNP_SECAGG0"].state.tap.STATUS_CSR
STATUS_CSR(63:0) = 0x0000020D00030159
  UPPER_MAP_(31:0) = 0x0000020D
  DFX_SECURE_POLICY(15:0) = 0x0003
  PERSONALITY_VALUE(4:0) = 0x00
  DLCS_VALUE(2:0) = 0x2
  EARLY_BOOT_DONE = 0x1
  PERSONALITY_LOCK = 0x0
  BOOT_HALT_STATUS = 0x1
  POLICY_UPDATE = 0x1
  POLICY_IS_FINAL = 0x0
  EARLY_BOOT_DEBUG_WINDOW = 0x0
  DELAY_AUTH_MODE_STATUS = 0x1
>>> Note: Configuration on all debug ports has started
       Target Configuration: CNL_CNP_OpenDCI_CCA_[DbC]_ReferenceSettings
       Note: Power Restore occurred
       Note: Target reset has occurred
       Note: The 'coregroupsactive' control variable has been set to 'GPC'
       Note: Configuration on all debug ports has finished
       Warn: Device trees have changed. Device nodes may have been added or removed.
>>>
>>? itp.boxes["CNL_MDU_DFX_AGG0"].state.tap.STATUS_CSR
STATUS_CSR(63:0) = 0x0000020D00030159
  UPPER_MAP_(31:0) = 0x0000020D
  DFX_SECURE_POLICY(15:0) = 0x0003
  PERSONALITY_VALUE(4:0) = 0x00
  DLCS_VALUE(2:0) = 0x2
  EARLY_BOOT_DONE = 0x1
  PERSONALITY_LOCK = 0x0
  BOOT_HALT_STATUS = 0x1
  POLICY_UPDATE = 0x1
  POLICY_IS_FINAL = 0x0
  EARLY_BOOT_DEBUG_WINDOW = 0x0
  DELAY_AUTH_MODE_STATUS = 0x1
>>? itp.irdrscan(itp.boxes["CNL_MDU_DFX_AGG0"], 0x42, 64, None, BitData(64, 0xDEADBEEFDEADBEEF))
[64b] 0x0000000000000000
>>? itp.boxes["CNL_MDU_DFX_AGG0"].state.tap.STATUS_CSR
STATUS_CSR(63:0) = 0x0000020D0003014D
  UPPER_MAP_(31:0) = 0x0000020D
  DFX_SECURE_POLICY(15:0) = 0x0003
  PERSONALITY_VALUE(4:0) = 0x00
  DLCS_VALUE(2:0) = 0x2
  EARLY_BOOT_DONE = 0x1
  PERSONALITY_LOCK = 0x0
  BOOT_HALT_STATUS = 0x0
  POLICY_UPDATE = 0x1
  POLICY_IS_FINAL = 0x1
  EARLY_BOOT_DEBUG_WINDOW = 0x0
  DELAY_AUTH_MODE_STATUS = 0x1
```

```
>>?
```

Listing 13: DAL CLI output of attempt to perform Red Unlock for CNL CPU via JTAG password

It is necessary to highlight that there is only one attempt to enter an unlock password. If it fails, DFX AGG will block the DFX Policy (the *POLICY_IS_FINAL* field of the status register), and all subsequent attempts to enter any of the unlock passwords (Red, Orange, or Metal) will be ignored. To enter a password again, you have to disable the keep-alive voltage on the platform. This is a protection mechanism against unlock password bruteforcing.

Note that DAM can be enabled not only via the appropriate DFX AGG register using the JTAG hardware debugger, but also using software/firmware, namely, Intel CSME (this option is provided in the FIT settings of the Intel Flash Image Tool for generating an SPI image of the platform).

Restricted-access versions (for Intel engineers) of Intel DAL and Intel OpenIPC software packages that allow for JTAG debugging contain a special **unlock** command [18]. The command application is based on hardware unlock passwords: when executed, the command reads *UniqueID* for DFX and sends it in encrypted form to one of the Intel's web resources. After authorization, the internal database of all chips released by Intel is searched for the appropriate JTAG unlock password based on the specified *UniqueID*. The password for the chip in question is then sent back to the client (DAL or OpenIPC) in encrypted form. The unlock command decrypts the password and writes it to the corresponding JTAG unlock register. This enables the engineer to activate the Red or Orange Unlock mode and proceed with debugging.

5.4 Software-based method via the *PERSONALITY* register of the DFX AGGREGATOR

Software-based DFX unlocking is performed via a special DFX AGG register called *PERSONALITY*. The DFX AGG device provides access to control registers via the IOSF-SB bus, and it is not connected to the backbone bus IOSF Primary (it is not registered among PCI devices and does not have PCI configuration space). A distinctive feature of the IOSF-SB bus is device addressing based on a unique one-byte identifier called Port ID. On each type of chips, the integrated DFX AGG device has its own IOSF-SB Port ID. We know its value for Atom SoCs (0x84), for the PCH of all desktop and server platforms (0xb7), and for modern processors starting with Cannon Lake (0x4c). SoCs (with a CPU and PCH integrated on a single chip), which are currently represented by various platforms based on the Atom CPU, contain only one DFX AGG IP block, which is shared by all platform components. As for desktop and server platforms (starting with Cannon Lake processors), they integrate two instances of DFX AGG - one in the microchip of the PCH system logic, and the other one in the processor. The DFX AGG instance in the PCH is the main one, and the DFX AGG instance in the processor is secondary. Processors that are based on the legacy architecture of the System Agent and Uncore devices (Skylake and earlier) do not have the DFX AGG device and, therefore, do not allow for software-based unlocking (to perform hardware-based unlocking as described in items 1 - 3, a designated IP block in Uncore is used). For systems with two DFX AGG devices, hardware-based DFX unlocking is performed separately for each of them (for example, a JTAG unlock password must be entered on the corresponding device). This research paper considers only the latest Intel chips that contain DFX AGG and does not cover processors of the Skylake generation and earlier. The latter allow for hardware-based unlocking via a JTAG password [19], but not for software-based unlocking, which is of greater interest.

DFX AGG controls write access to the *PERSONALITY* register using a special device identifier (in addition to Port ID) called Security Attribute of Initiator (SAI). This identifier is assigned to all devices connected to both IOSF-Primary and IOSF-SB. DFX AGG residing on a SoC or PCH allows write access to its registers for Intel CSME. DFX AGG of a processor can be written (for instance, to perform DFX unlocking) by another IP block that executes special firmware - namely, Power Control Unit (PCU) or PUNIT, the processor's power management microcontroller. Whether Intel CSME has write access to the DFX AGG device residing in the CPU is an open question, which we will address in our future research.

For now, we can say that it is possible to perform software-based DFX unlocking for the CPU core via Intel CSME only in relation to Atom SoCs. Meanwhile, the presence of DFX AGG in modern desktop processors means that it is not definitely impossible to perform software-based DFX unlocking of such processors (for example, via PUNIT).

The *PERSONALITY* register (at the address 0x000) of DFX AGG has the following format:

```
<State _name="PERSREG_MAP(31:0)">
    <Field _map="26:17" _name="PERSONALITY_MASK" />
    <Field _map="10:3" _name="USER_N_AUTH" />
    <Field _map="2:2" _name="OEM_AUTH" />
    <Field _map="1:1" _name="INTEL_AUTH" />
    <Field _map="0:0" _name="LOCK" />
```

```
</State>
```

Listing 14: DFX AGG *PERSONALITY* register bit layout

The *LOCK* bit allows blocking a subsequent write to the *PERSONALITY* register. The *INTEL_AUTH* and *OEM_AUTH* bits control Red Unlock and Orange Unlock, respectively. *USER_N_AUTH [3:10]* is used for specific types of DFX unlocking of certain functions (which are available in Red Unlock mode). At the moment, we know only about one specific type of DFX unlocking - namely, access to VISA signals of the Red security level, without other functions that are available in Red Unlock mode. This type of unlocking, known as VISA Signals Security Colors Override, is performed via bit #3. The *PERSONALITY_MASK* field allows applying a logical mask (NOTAND) to bits 1 - 10 of the written value.

Besides access control via SAI and the *LOCK* bit, writing to the *PERSONALITY* register is controlled by another DFX AGG register called *CONSENT* (address 0x04). It has the following format:

```
<State _name="CONSENT_CSR_MAP(31:0)">
    <Field _map="31:31" _name="DEBUG_NOTIFCATION" />
    <Field _map="30:30" _name="LOCK_PRIVACY_OPT" />
    <Field _map="0:0" _name="PRIVACY_OPT" />
</State>
```

Listing 15: DFX AGG *CONSENT* register bit layout

If set, the *PRIVACY_OPT* bit allows writing to the *PERSONALITY* register at the global level. The *LOCK_PRIVACY_OPT* field allows blocking writes to the *CONSENT* register, and the *DEBUG_NOTIFICATION* status bit indicates that connection with the platform via JTAG (DCI) is established.

Thus, the different mechanisms for controlling access to the *PERSONALITY* register have the following priority in excluding order:

- *LOCK* field of the *PERSONALITY* register
- *PRIVACY_OPT* field of the *CONSENT* register
- Access control via SAI in the DFX AGG device

If any of these mechanisms blocks writing to the *PERSONALITY* register for a device connected to the IOSF-SB bus (IOSF-SB master), neither writing to the *PERSONALITY* register nor DFX unlocking will be possible. We have discussed software-based DFX unlocking at the level of DFX AGG registers. Now let's outline how this mechanism works in the Intel CSME firmware for SoCs and PCHs.

At the hardware level, Intel CSME has access to the IOSF-SB bus as well as write access to the *PERSONALITY* register of DFX AGG. However, the implementation of the DFX unlocking mechanism in the Intel CSME firmware is much more complex than just writing to the two DFX AGG registers.

Firstly, on the side of the CSME hardware components, an additional mechanism known as CSE SRAM zeroing (or simply CSE zeroing) is implemented. When activated, this mechanism ensures zeroing of the whole CSME RAM. In this way, when the Red Unlock mode is enabled for an PCH (or SoC), it is impossible to get access to encryption keys used in Green Locked mode. (When performing Red Unlock, root encryption keys become unavailable in hardware and special debug keys are used instead [20].) So, during the DFX unlocking process, the Intel CSME firmware zeros (requests zeroing from the SRAM controller) its internal memory (SRAM of about 2 MB) and then reboots the CSME processor. Initialization code that is hard-coded in the CSME ROM checks whether an active request to perform DFX unlocking exists (the same hardware register that controls CSE zeroing is used). If such a request has been made prior to the reboot, the code writes corresponding values (based on the requested Unlock type - Red or Orange) to the *CONSENT* and *PERSONALITY* registers of DFX AGG. In this scenario, unlocking can be considered to be safe because an engineer accessing the PCH/SoC debugging tools cannot read the contents of the CSME memory and thus access encryption keys and other secrets.

Secondly, an unlock request is executed only after a special data block - Unlock Token (or Debug Token) - is analyzed. It is a small data block that can be integrated into the Intel CSME firmware using Intel System Tools (designed for generating an SPI image and configuring OEM parameters for platforms) or can be transferred to CSME from the CPU via Host Embedded Control Interface (HECI). The Unlock Token must have an Intel or OEM digital signature (depending on the type of unlocking requested) and contain a set of control commands (such as commands to perform Red or Orange Unlock). The CSME firmware checks the digital signature of the Unlock Token. If it is authentic, the firmware initiates the appropriate DFX unlocking process with a request for CSE zeroing.

However, the security architecture of the Intel CSME firmware (versions 11.0.x) used to have a critical design flaw: the BUP (BringUP) firmware module responsible for analyzing the Unlock Token and sending a CSE zeroing request could directly access DFX AGG and did not need the CSME ROM to perform unlocking after the CSE zeroing phase. TXE versions 3.1.x running on Atom SoCs had a similar architectural flaw, allowing BUP to access a special ATT-SB device that functions as a bridge between IOSF-SB and CSE address space and thus map DFX AGG. After finding an arbitrary code execution vulnerability in the BUP module (described in Intel-SA-00086), we were able to activate Red Unlock without CSE zeroing (by writing to *CONSENT* and *PERSONALITY* right from BUP). As a result, we gained access to firmware production secrets. Unfortunately, Intel has not made an official statement that would recognize this architectural flaw (in addition to running arbitrary code, we managed to impair the implemented protection mechanism and obtain access to some encryption keys) as a separate vulnerability. At the same time, Intel sent private letters to OEMs to explain the loss of encryption keys, and performed a re-key process by incrementing the Secure Version Number (SVN). However, in new versions of the Intel CSME firmware, this architectural flaw (direct access to DFX AGG or ATT-SB from BUP) was fixed, without a CVE assigned to it. The following is a fragment of a BUP module metadata file for Intel CSME 11.0.x firmware that describes devices available before the flaw was fixed:

```
. Ext#8 MmioRanges[43]:
 ...
 sel=107, base:F00B1004, size:00000004, flags:00000003 :: GASKET_GEN_PCIP
 sel=10F, base:F5010000, size:00001000, flags:00000003 ::  DFX_AGGREGATOR_SBS
 sel=117, base:E00C0000, size:00001000, flags:00000003 :: HDAU_PCIP
```

Listing 16: Fragment of BUP module metadata describing access to DFX AGG

The flag 0x00000003 indicates both read and write access. The following code fragment retrieved by the disassembler from the CSME ROM performs DFX unlocking after CSE zeroing:

```
if ( gasket_gen_mmio_cse_zeroing & 1 )
{
  // Do PCH Intel Unlock
  while ( !(gasket_gen_mmio_cse_zeroing & 8) );
  gasket_gen_mmio_cse_zeroing &= 0xFFFFFFFE;
  att_sb_mmio_slots[1].phys_addr = (int)&dfx_agg_csr_personality;
  att_sb_mmio_slots[1].mmio_size = 0x8000;
  att_sb_mmio_slots[1].reserved2 = 0;
  att_sb_mmio_slots[1].reserved3 = 0;
  att_sb_mmio_slots[1].sb_msg_params = 0x100706B7;
  att_sb_mmio_slots[1].sb_msg_extend_params = 0;
  att_sb_mmio_slots[1].enable = 1;
  dfx_agg_csr_consent |= 1u;
  spin_count = 0x30;
  do
    --spin_count;
  while ( spin_count );
  if ( misa_cfg_a0 & 0x42 ) // Rogue DMA Completion
    goto hang;
  dfx_agg_csr_personality |= 3u;
}
```

Listing 17: Disassembly fragment of CSME ROM code performing DFX unlocking

6 Speculative execution of microoperations implementing udbgrd and udbgwr

Earlier in this paper, we discussed the conditions required for the udbgrd and udbgwr instructions to be performed, paying particular attention to the Red Unlock mode in which the instructions execute the specified commands without throwing the #UD exception. However, there is another potential issue with these undocumented instructions - namely, speculative execution of microoperations that implement them. This requires thorough examination because the issue can occur even when calling udbgrd/udbgwr in User Mode without prior activation and not in Red Unlock mode. The problem of speculative execution of microoperations implementing udbgrd/udbgwr can be demonstrated by the state of two microarchitectural arrays that is partially given below. The state was obtained by intercepting ucode execution flow at the moment of calling the *generate_#UD* microcode subroutine from the address U028c, while also performing an inactivated udbgrd instruction with arguments **rcx = 8** and rax = 0 (SA register read).

Array 1 - Instruction Decode Queue (IDQ):

```
>>> idq_disassemble()
00: rax:= UJMP(0x00007d5d)
01: NOP
02: NOP
03: NOP
04: tmp3:= NOTAND_DSZ32(tmp3, 0xa0000000)
05: rax:= UJMPCC_DIRECT_NOTTAKEN_CONDZ(tmp3, 0x0000028d)
06: tmp3:= READURAM(0x0000005c)
07: rax:= BTUJB_DIRECT_NOTTAKEN(tmp3, 0x0000028d)
08: tmp3:= MOVEFROMCREG_DSZ64(0x000022e6)
09: rax:= BTUJNB_DIRECT_NOTTAKEN(tmp3, 0x00002769)
0a: tmp3:= MOVEFROMCREG_DSZ64(0x00002285)
0b: rax:= BTUJB_DIRECT_NOTTAKEN(tmp3, 0x00002769)
0c: rax:= BTUJB_DIRECT_NOTTAKEN(rcx, 0x00006bce)
0d: rax:= BTUJB_DIRECT_NOTTAKEN(rcx, 0x000027cc)
0e: tmp2:= CONCAT_DSZ32(rbx, rdx)
0f: tmp4:= NOTAND_DSZ32(0x00000001, rax)
10: tmp5:= AND_DSZ32(0x000000c0, rcx)
11: tmp5:= SHR_DSZ32(tmp5, 0x00000001)
12: tmp6:= AND_DSZ32(0x00000018, rcx)
13: tmp8:= OR_DSZ32(tmp6, tmp5)
14: rax:= BTUJNB_DIRECT_NOTTAKEN(tmp1, 0x000027d6)
15: tmp9:= ADD_DSZ32(tmp8, 0x00004392)
16: rax:= UJMP(0x00000000, tmp9)
17: NOP
```

Listing 18: IDQ disassembly for calling inactivated udbgrd instruction

Our IDQ disassembler has a number of limitations. In particular, it cannot tell the difference between microoperations with an unspecified destination (the MSROM microcode disassembler does it by analyzing the *dst* field of a microoperation) and does not support the second numeric constant for microoperations (for instance, in conditional jump microoperations, it determines either the second comparison argument or the index of the bit to test). But despite all that, by comparing the listings of the IDQ disassembly and udbgrd implementation in MSROM, we can notice something important: the IDQ contains microoperations that shouldn't be performed if the udbgrd/udbgrd instructions are not activated. (When intercepting control, we block the execution pipeline using an unconditional jump microoperation that jumps to its own address and is located in the IDQ slot with index #0x00; starting with index #0x04, udbgrd microoperations reside.)

The *generate_#UD* subroutine to which control passes if the instructions in question cannot be executed has the address U2769. The IDQ slot with index #0x09 contains a microoperation that must transfer control to *generate_#UD* (since we didn't activate the instructions in our experiment). However, as we can see, the IDQ slots starting with #0x0a contain microoperations that should only be performed if the instructions are activated! Then what is going on here and why does the execution queue contain microoperations from a conditional execution branch that must be skipped? This can be explained by speculative microoperations execution, which can be applied not only to microoperations of simple instructions (decoded by simple decoders) but also to the implementation of complex instructions in MSROM. The CRBUS read microoperation (MOVEFROMCREG_DSZ64) is time-consuming, so in order to prevent the pipeline from being stalled, Microcode Sequencer continues populating the IDQ while traversing through taken execution branches. (The term TAKEN is used to describe the behavior of a special CPU core unit known as Branch Predictor, and refers to a conditional execution branch that can be automatically predicted during speculative execution.) It is important to note that all conditional execution microoperations for the Atom Goldmont core are not taken (NOTTAKEN) - that is, during speculative execution, MS interprets the condition they are checked for as not true.

Inevitably, a question arises: are microoperations that are behind conditional jumps in the IDQ actually performed by the CPU core execution units (ALU, AGU, and so on), or are they placed in the IDQ in advance to be performed once the condition is resolved? We can easily answer this question by accessing a special microarchitectural array known as Real Register File (RRF). It contains the values of microarchitectural registers after their rename (register renaming is a distinct execution stage in the CPU core pipeline). Using the LDAT debug port, we can read the contents of RRF at any point in time. The following RRF fragment is related to the *generate_#UD* interception scenario described earlier:

Array 2 - Real Register File (RRF):

```
>>> ldat_array_dump(0x450, 1, 1, 0, 0x44)
```

```
array 01: dword 00: bank 00
0000: 0000000000000000 0000000000000000 000000007afe5998 00000000000048bd
0004: 000000007afe5991 f0000fffa09b0018 00000000a0000000 0000000000000000
0008: 0000000000000000 0000000000000000 0000000000000000 0000000000000000
000c: 0000000000000006 0000000000000107 0000000000000000 000000000000ffff
0010: 000000007afe5976 0000000000000000 0000000000000000 0000000000000000
0014: 0000000000000000 0000000000000000 0000000000000001 0000000000000000
0018: 00af9b0000000000 000000000000221e 000200065001a3a0 0000000000000000
001c: 0000280000000080 00000000004b0090 000000006b01e144 00000000ffffffff
0020: 000000000000037f 0000000000000000 0000000000000000 0000000000000002
0024: 0000000000000000 0000000000001441 000000007afe59a0 000000007afe5998
0028: 0000000000000008 0000000000000004 0000000000000000 000000006b01e18c
002c: 00000000ffff0001 00000000f0000fff 0000000000000018 0000000000000008
0030: 000000007afec7b0 00af9b0000000000 0000000000000000 00000000004b0090
0034: 0000000000000004 0000000000000000 00000000f0001000 0000000000000000
0038: 0000000000000000 0000000000000000 0000000000000000 0000000000000000
003c: 0000000000000000 0000000000000000 0000000000000000 000000006b01e144
0040: 000000000000439a 0000000000000008 0000000000000000 0000000000000000
```

Listing 19: Part of RRF for calling inactivated udbgrd instruction

During speculative execution of the IDQ slot with index #15, RRF must contain the value 0x00004392 + 0x00000008 (rcx argument when calling udbgrd). As we can see, the RRF slot with index #40 indeed contains the value 0x0000439a, which proves that speculative execution of the microoperations from the IDQ is performed and that, even when calling udbgrd/udbgwr without prior activation and not in Red Unlock mode, a part of their implementation in MSROM is still speculatively executed.

So the most important question now is whether the entire implementation of a specific command (such as a CRBUS read or write) is speculatively executed, or, rather, speculative execution is blocked at a certain stage. As we can see, the last speculative microoperation in the IDQ is UJMP to jump to the calculated address of the microcode implementing the command (the command index is passed in the rcx register). Note that in our case, the TAKEN path for the udbgrd instruction leads to speculative calculation of the target address for commands sent to udbgwr. It is demonstrated by the microcode that starts at the address U27d5, where the instruction type (read or write) is checked. The type is defined in tmp1: 0 means read, and 1 means write. However, judging by the IDQ, the execution doesn't go beyond the UJMP microoperation. We can find the explanation for this in the disassembler listing at the address of the UJMP microoperation - U27da. As we can see, UJMP is preceded by the LFNCEWAIT prefix. Let's try to find out what it means.

The MSROM microcode is defined by two microarchitectural data arrays (we partially discuss it in the write-up of our Atom Goldmont microcode disassembler tool [21], which we have recently published). The first array is a set of microoperations arranged in triads (every fourth microoperation is zero). The second array is a set of the so-called sequence words (SEQW): each such word consists of four control bytes that determine properties of the execution flow such as the address in microcode of the next triad of microoperations, certain commands related to the calling of subroutines, and commands for ending the sequencing of macroinstructions. The high-order SEQW byte contains bits that allow synchronizing with load microoperations as well as bits for some kind of synchronization barriers. Among SEQW bits, we found three commands for synchronization with load microoperations. We called them LFNCEMARK, LFNCEWAIT, and LFNCEWTMRK. The main reason of the need for these SEQW commands is the ability to synchronize between microoperations pertaining to different dependency chains. (A dependency chain is a group of data-dependent microoperations in which every microoperation, except for the first one, operates with the result of the previous one in the group.) Without the synchronization commands, independent chains of microoperations are executed out of order and in parallel even for the speculative scenarios. So, if dependency chain 1 (of microoperations) must be executed strictly after dependency chain 2, the need for the special synchronization mechanisms arises. The LFNCEWAIT command of SEQW allows the execution of microoperation chains to be postponed until all previous chains originating from the load microoperation are completed. Meanwhile, the LFNCEMARK command marks the last location (microcode address) in the microoperation execution flow up to which the load chains must be considered. Without LFNCEMARK, the considered load chains start from the beginning of the macroinstruction whereas two consecutive LFNCEMARKs allow limiting the considered set to those dependency chains that are between them. LFNCEWTMRK combines LFNCEWAIT and two LFNCEMARKs: it waits for previously marked dependency chains and then starts a new point to consider the next set of chains. We found out that the lfence x86 instruction is implemented in a similar way to the LFNCEWAIT microarchitectural command but there is one important difference: lfence instructions order only load instructions whereas LFNCEWAIT allows ordering any microoperations with

previous loads. The considered load microoperations for the synchronization include: loads from the CRBUS, reads of URAM, loads from the staging buffer and physical and linear **architectural** memory.

All this explains why UJMP at the address U27da is blocked (by the LFNCEWAIT command in the sequence word of its triad) until the CRBUS registers responsible for Red Unlock and the activation of the udbgrd/udbgwr instructions are read (see the LFNCEMARK command at U27d5). Once these CRBUS registers are read (in microoperation chains independent from UJMP), the CPU core pipeline realizes its own error and discards the speculatively executed microoperations. But what would speculative execution be like if UJMP were not affected by the LFNCEWAIT command? It would then go on, reaching the calculated address sent to UJMP, and we would be able to perform commands using the udbgrd and udbgwr instructions **without their activation and without Red Unlock**! While it is not the case with the Atom Goldmont microarchitecture, how can we be certain that there is no such problem with other modern Intel processors that, we think, also support udbgrd/udbgwr (especially considering that LFNCE* commands cannot be set automatically by some microcode programming assist tool because they imply a logic external to a data dependency)? It is certainly hard to imagine the possibility of speculative writing of microarchitectural data (i.e. URAM locations), yet we think that it is technically feasible: for individual microoperations of the same macroinstruction, independent retirement (the final execution stage in the CPU core pipeline intended to order the execution of x86 instructions and pass their data to the architectural state) is pointless, while the moment of transferring write buffers to microarchitectural arrays (or to internal buses) is not defined. We assume that perhaps the speculatively written microarchitectural data cannot be reverted at all by the pipeline recovery mechanism which is why these very important synchronization commands exist at the microcode level. Speculative reading of microarchitectural data with the udbgrd instruction seems to be even more likely: we suspect that the read results can be obtained by using the CPU cache as a side channel. (Our hypothesis is that speculative execution beyond the udbgrd macroinstruction is possible in this case because a command to end sequencing of its implementation in MSROM, which is also specified via SEQW, allows speculative execution of subsequent macroinstructions).

7 Conclusion

As described in this paper, we discovered two undocumented x86 instructions that allow reading and writing microarchitectural data and called them udbgrd and udbgwr, respectively. One of the commands supported for the instructions is the CRBUS read/write command, which can be used to modify microcode in MSROM via Patch RAM and match/patch registers. We can prove the existence of these undocumented instructions in Atom Goldmont and Atom Goldmont Plus processors - see our PoC published together with the paper [14]. We found that the instructions must be activated through MSR 0x1e6 and that the processor executes them only after performing the DFX unlock procedure called Red Unlock. In our opinion, the instructions are intended for CPU microarchitecture debugging by Intel engineers; however, the existence of such instructions poses a security threat since there is publicly available PoC code [9] that can activate the Red Unlock mode on one of the modern Intel platforms. Moreover, we demonstrated that speculative execution of microoperations (rather than macroinstructions that they implement) is possible for udbgrd/udbgwr, which needs to be thoroughly examined with regard to all present-day Intel processors. For this purpose, we created a special tool [22] that, when run on a processor, checks whether the udbgrd and udbgwr instructions can be executed on it and, if not, checks whether the processor allows speculative writing and reading of microarchitectural data when calling the instructions. This tool executes the udbgwr instruction to write a specific URAM address (Time Stamp Counter (TSC) multiplier used by the rdtsc x86 instruction), which is known for Big Cores as well as for Atom Goldmont, and then attempts to read the written data using architectural mechanisms available in User Mode. The tool also tries to speculatively read the TSC multiplier in URAM with the udbgrd instruction by using CPU cache as a mechanism to retrieve the read data. On top of that, we were the first to publicly provide a list of all (as far as we can tell) possible ways to activate the Red Unlock mode for CPU debugging and to demonstrate that some of them are rather dangerous (for example, software-based unlocking via Intel CSME and PUNIT firmware and processor-specific OTP configuration).

References

[1] Intel Corp. Intel® 64 and IA-32 Architectures Software Developer's Manual. https://software.intel.com/content/dam/develop/external/us/en/documents-tps/325462-sdm-vol-1-2abcd-3abcd.pdf, 2021.

[2] C. Easdon. Undocumented CPU Behavior: Analyzing Undocumented Opcodes on Intel x86-64. https://www.cattius.com/images/undocumented-cpu-behavior.pdf, 2020.

[3] R. Collins. Undocumented OpCodes. http://www.rcollins.org/secrets/OpCodes.html, 1995.

[4] C. Domas. Breaking the x86 ISA. https://www.blackhat.com/docs/us-17/thursday/us-17-Domas-Breaking-The-x86-Instruction-Set-wp.pdf, Jul. 2017.

[5] Wikipedia. LOADALL. https://en.wikipedia.org/wiki/LOADALL.

[6] Intel Corp. Intel® Converged Security and Management Engine (Intel® CSME) [Security White Paper]. https://www.intel.com/content/dam/www/public/us/en/security-advisory/documents/intel-csme-security-white-paper.pdf, Nov. 2020.

[7] M. Ermolov M. Goryachy. How to Hack a Turned-Off Computer, or Running Unsigned Code in Intel Management Engine. https://www.blackhat.com/docs/eu-17/materials/eu-17-Goryachy-How-To-Hack-A-Turned-Off-Computer-Or-Running-Unsigned-Code-In-Intel-Management-Engi pdf, Dec. 2017.

[8] Intel Corp. The Intel® Converged Security and Management Engine (CSME) Delayed Authentication Mode (DAM) vulnerability - CVE-2018-3659 and CVE-2018-3643. https://www.intel.com/content/dam/www/public/us/en/security-advisory/documents/the-intel-csme-dam-vulnerability-cve-2018-3659-and-cve-2018-3643-whitepaper.pdf, Jun. 2020.

[9] Positive Research. IntelTXE-PoC. https://github.com/ptresearch/IntelTXE-PoC, 2020.

[10] Ucode Research Team. Ucode Disassembler. Description of Some Important Microoperations. https://github.com/chip-red-pill/uCodeDisasm#description-of-some-important-microoperations, 2021.

[11] IEEE 1149.1 Working Group Official Webpage. https://grouper.ieee.org/groups/1149/1, 2021.

[12] Ucode Research Team. Ucode Disassembler. https://github.com/chip-red-pill/uCodeDisasm, 2021.

[13] C. Easdon. Undocumented CPU Behavior: Analyzing Undocumented Opcodes on Intel x86-64. https://www.cattius.com/images/undocumented-cpu-behavior.pdf#page=17, 2020.

[14] Ucode Research Team. Microarchitecture Debug PoC. https://github.com/chip-red-pill/udbgInstr/tree/main/udebug, 2021.

[15] C. Patulea [@eigma]. Evidence of this instruction as far back as KNC (a P6 variant) [Tweet]. https://twitter.com/eigma/status/1373155650432290819.

[16] C. Bölük. 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, Mar. 2021.

[17] Intel Corp. An Overview of Advanced Failure Analysis Techniques for Pentium® and Pentium® Pro Microprocessors. https://www.intel.com/content/dam/www/public/us/en/documents/research/1998-vol02-iss-2-intel-technology-journal.pdf#page=2, 1998.

[18] M. Ermolov M. Goryachy. Intel VISA: Through the Rabbit Hole. https://i.blackhat.com/asia-19/Thu-March-28/bh-asia-Goryachy-Ermolov-Intel-Visa-Through-the-Rabbit-Hole.pdf#page=60, 2019.

[19] M. Ermolov M. Goryachy. Intel VISA: Through the Rabbit Hole. https://i.blackhat.com/asia-19/Thu-March-28/bh-asia-Goryachy-Ermolov-Intel-Visa-Through-the-Rabbit-Hole.pdf#page=57, 2019.

[20] Intel Corp. The Intel® Converged Security and Management Engine (CSME) Delayed Authentication Mode (DAM) vulnerability - CVE-2018-3659 and CVE-2018-3643. https://www.intel.com/content/dam/www/public/us/en/security-advisory/documents/the-intel-csme-dam-vulnerability-cve-2018-3659-and-cve-2018-3643-whitepaper.pdf#page=7, Jun. 2020.

[21] Ucode Research Team. Ucode Disassembler. The Structure and the Binary Format of Intel Atom Goldmont Microcode. https://github.com/chip-red-pill/uCodeDisasm#the-structure-and-the-binary-format-of-intel-atom-goldmont-microcode, 2021.

[22] Ucode Research Team. Microarchitecture Debug Check Tool. https://github.com/chip-red-pill/udbgInstr/tree/main/udbg_test, 2021.