Automated Analysis of Undocumented Intel x86-64 Instructions

and the Case for Public CPU Auditing Tools

Catherine Easdon

June 19, 2018

This document is set in Palatino, compiled with pdfLaTeX2e and Biber.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Abstract

Undocumented instructions are instructions which are either not officially part of an instruction set architecture or not officially supported by a particular CPU model, but nevertheless can be executed in practice. Building on prior research by Domas (2017), this paper presents strategies for detecting and analysing x86-64 instruction functionality on Intel CPUs and a software tool for automating this analysis. The paper's key contributions are: a method for stable handling of UD exceptions in the kernel without modifying the IDT; the concept of determining instruction functionality using performance counters for both instructions which successfully execute and faulting instructions which are speculatively executed; and identification of unexplained decoding and exception behaviour. Additionally the potential consequences of undocumented opcodes and other CPU vulnerabilities are considered. A case is presented for the necessity of public verification and security auditing of CPUs, and in particular the urgent need for publicly-available automated tools for such auditing, in the hope of encouraging further research in this under-investigated field.

Contents

1 Motivation

This section presents a selection of Intel CPU vulnerabilities which pose a threat to system stability or security, and which are the motivation for this investigation of undocumented instructions. The x86-64 ISA is immensely complex, and this complexity greatly increases the potential for bugs and undocumented instruction behaviour. Instructions which cause system hangs are a threat to system stability and could be used to mount denial of service attacks, whilst other instructions can create security vulnerabilities in the form of side channels. The possibility exists for CPU backdoors, either intentionally created by Intel for debugging purposes or created under legal compulsion from the US government, and these could be implemented in the form of compromised opcodes or microcode. Finally, the broader case for the creation of public CPU auditing tools is presented and existing tools are discussed.

1.1 Why Investigate Instructions?

1.1.1 ISA Complexity

Potential Instruction Space

As an Intel x86_64 instruction can be up to 15 bytes long, the potential instruction space is enormous: there are 2^{120} possibilities (1.33 x 10^{36}). Of course, as there are so many possible combinations of prefixes and operands, the majority of these instructions will essentially be minor variations on a much smaller set of actual 'instructions' which have distinct behaviour (i.e. would be represented by an assembly mnemonic, if they were documented). Heule, 2016 estimated the total number of documented instructions (considering all combinations of mnemonic and operand type to be a different instruction, but without considering prefixes) to be 3,684; although this does not include all possible variants of these distinct instructions, this still leaves a vast range of undocumented instruction encodings. How many of these might the processor be able to execute, if any, and what might these undocumented instructions do?

Bugs

The success of x86 has now become its curse as attempts to maintain full backward-compatibility with prior versions (dating back to the Intel 8086 in 1978) whilst evolving its capabilities have made the ISA incredibly complex, with obsolete historic behaviour and many unusual edge cases retained solely for compatibility purposes. As the ISA becomes ever more complex the likelihood of implementation bugs increases; bugs have plagued several new features, most notably TSX, which had to be completely disabled via microcode on Haswell processors and continued to cause 'unpredictable system behaviour' on Skylake (Intel, 2017a).

Intel regularly publishes specification updates containing long lists of errata (bugs), many of which cause system hangs, machine checks, or unpredictable system behaviour, and have no planned fix. Bugs in which instruction behaviour contradicts the ISA are common; for example, the errata list for 7th generation (Kaby Lake) and certain 8th generation processors (Intel, 2017b) includes SMSW, which should be illegal within an SGX enclave and yet still executes; TZCNT, which executes even on models for which it is supposed to be interpreted as a sequence of different instructions; and all 256-bit AVX instructions, which may cause 'unpredictable behaviour' under 'complex micro-architectural conditions' (conditions which are not detailed, so no developer can attempt to avoid this situation). Perhaps even more alarming than these bugs, however, is that simply turning the display on and off may be sufficient to hang the system! Given the immense pressure Intel are currently under to rapidly develop hardware mitigations against the Meltdown and Spectre vulnerabilities, it seems likely we will see an increase in the rate of bugs over the next few years, as there will be less time available for testing and verification.

Poor Documentation and Developer Confusion

The ISA's complexity also increases the risk of misdocumentation of instruction behaviour or developer misunderstanding. For example, attempts at automatic synthesis of a formal semantics for the ISA (specifically for 1795 Haswell instructions) found 50 errors in the Intel software developer manual, which were confirmed as such by Intel (Heule et al., 2016); as these errors were found manually and only a subset of the ISA was synthesised, it is likely the manual (which is the primary official source of information on Intel's interpretation of the ISA) contains many further errors. Intel also has a long history of disclosing certain documentation only to key partners. Most infamous of these cases is Appendix H of the third volume of the developer's manual for the Pentium processor family, which was available only under a legally binding nondisclosure agreement in an attempt to keep details of Virtual Mode Extensions (VME) secret from competitors (Case, 1993). Intel have also confirmed that they add undocumented features requested by key corporate partners into their main product lines (rather than producing custom processors for these use cases) (Morgan, 2013). Regarding developer confusion, the recently disclosed POP SS/MOV SS vulnerability affecting Linux, Windows, macOS, FreeBSD, and some Xen implementations exploited developer misinterpretation of a statement in the Intel SDM regarding DB exception handling when either of these instructions is followed by INT N or SYSCALL. The exploit can be used to make the kernel use state information chosen by the original unprivileged program, leading to kernel crashes, user mode access to kernel memory or even execution of arbitrary kernel code (Peterson and Mulasmajic, 2018). Intel have already released a revised version of the SDM in response to this, but there may be many more such cases waiting to be found.

1.1.2 Halt and Catch Fire

A *Halt and catch fire* (HCF) instruction is an instruction which crashes the system when executed. The most infamous case of this was F0 0F C7 C8 (the F00F bug) on Pentium processors prior to the B2 stepping, an invalid instruction encoding which could execute from a user mode program and completely halt the procesor until it was rebooted (Collins, 1998). Other Intel examples include F4 on certain Intel DX4 processors (intended to only temporarily halt the processor, but buggy), and 0F 04 on the 80286.

On the Motorola 6800 the opcodes 9D and DD were intentionally implemented debug instructions; from the user's perspective the machine simply halts, but the processor is actually reading every single address in memory sequentially in an infinite loop. The instruction became popular with developers as such predictable behaviour was useful for identifying hardware timing and address logic problems. HCF instructions are obviously extremely inconvenient for users if encountered and could cause data loss or corruption if the processor halts during data processing operations. They could also be the basis for a denial of service attack on critical infrastructure. However, despite their nickname these instructions are typically incapable of causing actual hardware damage. In contrast, *killer poke* instructions have the potential to cause permanent hardware damage (note that some researchers do not make this distinction between HCF and killer poke, and use the terms interchangeably). For example, it was reported that on certain Commodore PET models the integrated CRT monitor could potentially be damaged by the instruction POKE 59458,62 (modifying a pin register), as the instruction caused unusual output from the video chip leading to the monitor de-syncing and being damaged if not turned off promptly (Fachat, 2013). Killer poke instructions have typically been found on older hardware and games consoles without hardware memory management, but could feasibly still exist on modern hardware.

1.1.3 Side Channels

Side channel attacks use implementation dependencies to exploit a system. For example, a data-dependent encryption algorithm can be exploited by measuring differences in timing or power consumption. The enormous risk CPU side channels pose to security was brought to the world's attention in January 2018 with the disclosure of Meltdown and Spectre, and continues to be in focus with the ongoing disclosure of the Spectre class of vulnerabilities (Lipp et al., 2018, Kocher et al., 2018). Such information leakage is clearly a security risk for anyone processing sensitive data, but side channels pose a particular threat to users of cloud computing instances. A single hypervisor can host virtual machines for many customers, each of which can execute arbitrary code within their VM and might attempt a side channel attack for cross-VM information stealing; one of the recently disclosed SpectreNG vulnerabilities alarmingly enables a VM to attack both other VMs and the hypervisor itself (Schmidt, 2018). An individual opcode alone can open a side channel, as seen with the case of CLFLUSH; this instruction for evicting data from the cache has a shorter execution time if the data are not cached compared to when the data are cached, enabling the powerful Flush+Flush cache timing attack (Gruss et al., 2016).

1.1.4 Backdoors

Given the staggering extent of the global mass surveillance and cyber warfare activities of governmental intelligence agencies revealed by Wikileaks from 2011 onwards, it seems very plausible US governmental agencies may have compelled Intel to incorporate surveillance measures ("backdoors") into their CPUs. Many have argued that this is unlikely, as it would be far easier to exploit higher-level aspects of a system such as the OS. However, lower-level exploits have the crucial advantage that they are much more persistent and much harder to detect. The NSA's and CIA's extensive malware development is known to have included many low-level firmware and hardware exploits (Spiegel, 2013, Storm, 2014); a typical attack such as IRONCHEF combines exploitation of the BIOS and SMM to provide access persistence to a system via a hardware implant (Appelbaum, 2013). Such an implant can be inserted by an agent with access to the system or by tampering of hardware whilst in transit from the manufacturer ("interdiction"), but this is time-consuming and inevitably involves the risk of detection; the process is greatly simplified if manufacturers build in backdoors to their technologies from the start. According to budget documentation obtained by Wikileaks, the NSA spends more than $250 million a year on its Sigint Enabling Project to "covertly influence and/or overtly leverage" the designs of technology products (Nicole Perlroth and Shane, 2013). There are known examples of this occurring in the US and partner intelligence nations (such as the UK), with technology companies being paid or legally compelled by governmental agencies to make surveillance modifications to the software or hardware systems they develop (Menn, 2013, Ball, Harding, and Garside, 2013). One of the biggest concerns with governmental backdoors is the potentially disastrous security consequences if such backdoors become publicly known; whilst governmental agencies may perhaps only use a backdoor in a targeted manner to investigate specific individuals, other attackers may not be so restrained.

Microcode

Modern x86 CPUs translate their variable-length CISC instruction set into micro-ops (denoted for simplicity in this paper as uops, without the Greek mu), the fixed-length RISC instruction set which is actually implemented in hardware. Each instruction decoder can produce up to 4 fused-domain uops at a time; more complex instructions require uops to be fetched from the microcode ROM (Intel, 2016). Complex instructions are therefore essentially short uop programs, and these programs can be modified in a microcode update, which writes to microcode patch RAM; an instruction 'program' in patch RAM takes precedence over the original definition in ROM at execution time. As this memory is volatile microcode updates must be re-applied by the BIOS or operating system at startup. This has been a feature of Intel CPUs since the P6 family (Pentium Pro) to allow for patching of critical bugs. However, such updates are also a potential exploit mechanism. A malicious microcode update might modify an opcode's behaviour to implement a privilege escalation mechanism (for example, when executed with a 'password' combination of register values the opcode switches the processor to SMM), to introduce a side channel, or to compromise cryptographic operations. Duflot, 2008 considers the consequences of CPU backdoors with regard to privilege escalation in more detail.

The potential for malicious microcode updates was demonstrated by Koppe et al., 2017, who reverse-engineered AMD microcode on the K8 and K10 microarchitectures to implement proof-of-concept microcode trojans and demonstrated how, thanks to ASM.JS, an attacker could even exploit these remotely via the browser. This was possible as AMD microcode was merely obfuscated rather than encrypted on older microarchitectures; microcode patches on newer AMD processors are encrypted and are not known to have been reverse-engineered. Likewise, Intel microcode updates are encrypted and decoded by the processor, and incorrectly encrypted updates will fail to load (Intel, 2017c). Attempts at reverse-engineering have suggested that the update is signed with 2048-bit RSA (Hawkes, 2012). This provides some protection against malicious updates, but the mechanism could still potentially be exploited, for example if Intel's private key were disclosed or discovered, or if exploitable bugs were discovered in the decoding routine; there is some evidence to suggest that it may contain a division by zero bug (Hawkes, 2012). Furthermore, the BIOS is responsible for ensuring that a newer microcode version is not overwritten by an older version (Intel, 2017c); as

BIOSes have been the target of many attacks and are a known weak point in a system's security (Matrosov, 2017), this suggests it would be feasible to revert a processor to older microcode (microcode still vulnerable to Spectre, for example) if the BIOS were already compromised.

RDRAND, an instruction which returns a random number and is recommended by Intel for cryptographic purposes, provides an excellent example of why an opcode might be targeted for compromise via microcode. There has been considerable speculation that this instruction may have been backdoored by Intel under compulsion from US intelligence agencies; it is based on CTR-DRBG from the NIST SP 800-90 standard, the same standard which featured the flawed Dual EC DRBG implementation (Intel, 2017c). A backdoor could be implemented in the design, for example with flaws in CTR-DRBG akin to Dual EC DRBG's 'skeleton key', enabling the output to pass randomness tests, but rendering the output easy to brute force for an attacker with knowledge of crucial values. Alternatively it could be implemented as a targeted microcode update for select individuals, for example to provide low entropy output when RDRAND is used directly (e.g for ASLR on Linux), or to produce actively malicious output when combined with other entropy sources. In 2013 this was identified as a vulnerability in the Linux kernel's random.c (Hornby, 2017 - now patched): other entropy sources were XORed with RDRAND *after* all entropy mixing, so a malicious RDRAND would have control over 'random' number output (for example by outputing a value which would XOR to 0 with the other entropy sources). There are many potential targets other than RDRAND; for example, there is strong evidence to suggest that all SGX instructions are implemented in microcode, so all the security claims of SGX could potentially be compromised by a malicious microcode update (Costan, Lebedev, and Devadas, 2017b).

Intel ME and SMM

Other likely candidates for an intentional backdoor include System Management Mode (SMM) and the Intel Management Engine (ME). Historically these have been exploited with complex exploit programs rather than with individual instructions, but it is possible there are undocumented instructions to access them (likely with a required register value or other more complex 'password' mechanism) or which enable undocumented functionality within them. ME poses a particular challenge as it is an entirely separate processor, so must be tested for undocumented instructions individually from the main processor.

SMM is a processor mode with higher privileges than the operating system and hypervisor which handles power and hardware management (such as deep sleep, shutdown on overheating, and legacy USB support). SMM poses a significant security risk as the operating system has no knowledge of what code is executed in this mode, and is powerless to prevent the processor switching into this mode as it cannot block the system management interrupt (SMI) used to enter it. SMM code is installed by the BIOS, and so the SMM could be compromised via one of the many BIOS exploits (see above). Numerous SMM rootkits already exist (Embleton, Sparks, and Zou, 2008, an OS-independent keylogger which sends logged keystrokes over the network to an attacker via UDP, is one of many examples), including several known to have been developed by the NSA (Schneier, 2014b, Schneier, 2014a). Domas, 2015 demonstrated a privilege escalation vulnerability which allows arbitrary code execution in the SMM from ring 0 on pre-2013 Intel processors.

Intel ME is an even more troubling target for a backdoor. It includes a separate Intel x86 processor running its own operating system, has direct network access via the Ethernet controller, and can run even whilst the system is turned off but still connected to power (Skochinsky and Corna, 2017). As the subsystem's microcontroller is integrated into the Platform Controller Hub (PCH), ME has access to all communication between the processor and peripherals. It enables Intel's Active Management Technology (AMT) administrative features, and in settings where one administrator is managing many systems (for example in schools or businesses) these features are genuinely useful: the ability to deploy updates over the network to all systems at once, even if some systems are powered off, is very convenient. Yet Intel ME is also present in processors which do not support AMT, and for these processors its existence seems harder to justify. It is also involved with functions such as hardware initialisation, Boot Guard, SGX (Skochinsky and Corna, 2017), and Quiet System Technology for fan control and hardware sensors monitoring (Intel, 2010). However, its capabilities seem too far-reaching to be justified by this functionality alone; none of these justify its independent network access or capacity to run even when the system is powered off, for example.

Intel ME is extremely difficult to disable; several tools have been developed, but the disabling process is complex as if certain conditions are not met the processor will not boot or will shutdown after 30 minutes. Notably, a crucial stage of the current disablement process involves setting the HAP bit, which Intel have confirmed was added to support the US government's High Assurance Platform program; this demonstrates the US government's awareness that ME is indeed too vulnerable to be enabled on systems where security is critical (Ermolov and Goryachy, 2017). Even if no intentional backdoor exists, any vulnerability in this ultra-privileged coprocessor is clearly a severe security threat. Troublingly, severe vulnerabilities have already been discovered. The 'Silent Bob is Silent' vulnerability allows remote administrative AMT control of all systems with AMT enabled and had allegedly been known to Intel for five years before it was publicly disclosed; it is still in the process of being patched (Ylonen, 2018). A further vulnerability has been found allowing arbitrary code execution in ME if an attacker has local access to the machine (Ermolov and Goryachy, 2018); this does not require AMT to be enabled or supported on the system.

Debugging Mechanism

Finally, there is the possibility that an undocumented debugging mechanism exists which enables undocumented behaviour or creates a security vulnerability. For example, the Intel 80286 and 80386 processors supported the LOADALL instruction for testing and debugging via in circuit emulation. It could be used to access the entire memory address space from real mode and to put the processor into illegal states not possible with other instructions; on the 80386, LOADALL was undocumented and had officially been removed, yet was still executable (Collins, 1991). More recently, debugging features have been discovered on AMD processors which are activated by loading a password value of 9C5A203A into the EDI register to enable the use of four undocumented MSRs. However, the fact that writing to these MSRs requires ring 0 privileges somewhat mitigates the security risk posed by this mechanism (Goodin, 2010).

1.2 The Case for Public CPU Auditing

The above vulnerabilities barely scratch the surface of the complex landscape of x86 CPU security. A fully comprehensive analysis is beyond the scope of this paper, but suggested further reading on other important aspects such as BIOS/UEFI as the root of trust, Boot Guard as a potential backdoor, TPM, TXT and SGX and their security advantages/vulnerabilities is Rutkowska, 2015; Costan, Lebedev, and Devadas, 2017a; and Costan, Lebedev, and Devadas, 2017b. Research has explored numerous possibilities for backdoors involving hardware modification; most alarmingly, Becker et al., 2013 demonstrated the potential for virtually undetectable hardware trojans implemented *below the gate level* by changing the dopant polarity of transistors on the chip.

Although not comprehensive, the above examples do illustrate that modern CPUs present an immense potential attack surface for compromising the security of a system. Yet this area is underresearched and the risk it poses is frequently underestimated by IT professionals. Whilst CPU manufacturers understandably wish to protect their intellectual property, the severity of both the recently exposed vulnerabilities (Meltdown, Spectre, ME, amongst others) and potential future vulnerabilities demonstrates that we cannot afford to permit manufacturers to continue relying on security by obscurity. The benefits of public peer review have been long known in cryptography and in the open source community; it is time for us to stop relying on the CPU as an opaque root of all trust in systems and instead apply public peer review to its design and security. With their history of secrecy and documentation NDAs, this information is unlikely to be forthcoming from manufacturers such as Intel, and so further research and development of public auditing tools is required to gather this knowledge. Furthermore, these tools should be as accessible as possible to allow any interested user to test their CPU for defects or compromise, as an individual CPU may have vulnerabilities (particularly malicious microcode or a hardware backdoor) even if a standard CPU of its model is secure.

1.2.1 Existing Tools

Sandsifter is a tool for detecting undocumented opcodes and was developed as part of the first known research into undocumented opcodes on modern (post-Pentium) Intel CPUs (Domas, 2017). Many of the discovered opcodes were analysed, and findings included a halt and catch fire instruction which completely locks execution from user space on an as-yet undisclosed x86 CPU (pending responsible disclosure). However, it is unclear how analysis was carried out in this work or to what extent it was automated; Sandsifter itself has no capacity for analysis. Manual analysis of instruction functionality is extremely time-consuming and given the size of the x86-64 instruction space there is a clear need to automate this analysis. Automated analysis also makes investigating undocumented instructions accessible to users with less in-depth knowledge of the ISA. This work directly builds on Sandsifter to create such an automated tool; further information about Sandsifter is provided in 3.1.4.

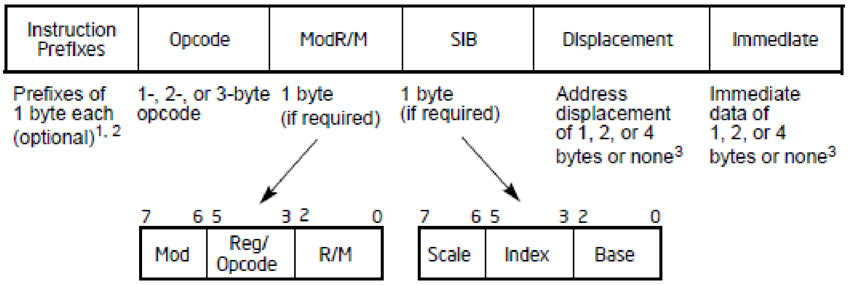
Chipsec is a tool for platform security assessment which can be used to detect configuration vulnerabilities in the BIOS/UEFI, Secure Boot, and SMM. It also provides a framework for further firmware testing and analysis beyond the scope of the provided modules. Whilst it is not an independent tool (it is developed by Intel and so will obviously not identify intentional backdoors or vulnerabilities in non-public functionality) it is excellent for identifying OEM misconfiguration. Correct configuration of firmware can prevent several of the vulnerabilities identified above.

2 Background

2.1 x86 Instruction Format

x86 instructions range from one to fifteen bytes in length. The most crucial element is the opcode; the other elements can be optional depending on the given instruction, but an instruction without an opcode is always invalid. Opcodes can be one, two, or three bytes long; there can also be a 3-bit opcode field in the ModR/M byte. Documented two and three byte instructions always begin with either 0x0f (known as the escape opcode byte) or a mandatory prefix (0x66, 0xf2, or 0xf3) followed by 0x0f.

The other elements of an instruction are the prefixes before the opcode, the ModR/M byte, SIB byte and addressing displacement byte(s) which encode the form of addressing, and the immediate operand(s). Whether these elements are required, optional or invalid varies depending on the opcode. Certain (documented) encodings of ModR/M and SIB have no effect and so can be used to create an alternate representation of an instruction (Johnson, 2011).



1. The REX prefix is optional, but if used must be immediately before the opcode; see Section 2.2.1, "REX Prefixes" for additional information.
2. For VEX encoding information, see Section 2.3, "Intel® Advanced Vector Extensions (Intel® AVX)".
3. Some rare instructions can take an 8B immediate or 8B displacement.

Figure 2.1: Intel instruction format (Intel, 2017c)

2.1.1 Prefixes

The possible prefixes are the legacy prefixes (0xf0, 0xf2, 0xf3, 0x2e, 0x36, 0x3e, 0x26, 0x64, 0x65, 0x66, and 0x67) and the REX prefixes from 0x40 to 0x4f (only in 64-bit mode; these are INC or DEC opcodes in other modes). Prefixes appear to be mostly ignored when added to instructions which they officially do not support. However, Intel's documentation says that such prefix usage may cause "unpredictable behaviour" (which in practice often means that the behaviour is entirely predictable, but Intel do not wish to document it) and they have been observed to sometimes have an effect. For example, the 0x67 prefix is meant to be used with a modR/M byte, but Johnson (2011) demonstrated it can also modify the displacement value of MOV instructions.

Johnson also demonstrated that up to 14 prefixes can be added to a one byte opcode (the 15 byte instruction limit cannot be exceeded, and the instruction must include an opcode), which contradicts the oft-reported limit of four legacy prefix bytes (Intel's own documentation does not state this limit, instead saying that it is "only useful" to include one prefix from each of the four groups).

2.2 Signals, Interrupts and Exceptions

Note: for brevity, this section exclusively covers the behaviour of Intel processors, and all references to 'operating system' refer to Linux distributions running kernel 4.17. See Chapter 6 of the Intel Developer Manual Intel (2017c), and traps.c and mce.c of the Linux kernel for further details.

Interrupts and exceptions indicate events which require the processor's immediate attention. **Interrupts** are signals from external hardware (or from software with the INT n instruction), whereas **exceptions** indicate an error condition in the processor. The processor's behaviour when an interrupt or exception occurs is not determined by default; the processor requires the operating system to define 'handlers', which are short sections of code for dealing with each interrupt/exception. The operating system defines these at boot time by initialising the Interrupt Descriptor Table (IDT) with gate descriptors (entries). A gate descriptor tells the processor at which address it can find the code for a given handler.

Exceptions are divided into three classes: traps, aborts, and faults.

- A **trap** (debug, breakpoint, or rarely the overflow exception) occurs immediately after an instruction when the processor detects a debug condition. Traps are used by debuggers to step through a program or break on a given condition (note that certain types of debug exception can also be faults). The return address for the handler (defined by the saved values of the CS and EIP/RIP registers, which the processor pushes to the handler's stack) is the instruction following the trapped instruction. There is a program state change, as a trap occurs after instruction execution.
- An **abort** exception (double fault or machine check) indicates a serious error in the CPU and may cause a kernel panic (the machine check handler does attempt to recover if possible). In the case of a double fault, the return address is undefined - there is nowhere to return to, so there is little the handler can do to try and recover.
- A **fault** is an exception which can normally be corrected to resume execution without faulting, and so the return address points to the instruction which caused the fault. For example, when a page fault occurs the kernel allocates the required page to the process and then the instruction can be executed again successfully. There is normally no program state change as the entire machine state is restored to its state before the faulting instruction, although there are some exceptions to this such as the floating-point error exception and when an exception occurs during a task switch.

An invalid opcode exception (UD) is a fault, but in this case execution cannot simply be resumed at the same instruction, because it will throw the same UD exception again, and keep repeating this indefinitely unless the handler changes the return address. Operating systems normally handle UD by killing the program which ran the instruction. However, this is a problem for investigating undocumented instructions, because the majority of instructions in the 'gaps' between the documented instructions throw UD (general protection fault, GP and page fault, PF, also sometimes occur). If the testing program is killed after each instruction it is very difficult to efficiently test millions of instructions! Fortunately, a user mode program can easily register its own signal handler to handle exceptions on its own. **Signals** are how interrupts and exceptions are passed to user mode programs by the Linux kernel. For example, a program will receive a SIG_ILL signal if it runs an instruction which throws UD, and SIG_SEGV if it throws GP. Custom exception handling in a kernel driver is more challenging, but still possible; see 3.2.2 for further details.

If several exceptions or interrupts occur simultaneously, the processor runs the handler for the exception or interrupt with the highest priority. Lower priority interrupts are held pending, whilst lower priority exceptions are discarded and then regenerated when the handler returns execution to the point in the program where the exceptions occurred. Note that in some cases exceptions cannot be handled serially: depending on the types of the first and second exceptions, a double fault exception may be thrown (see Intel, 2017c, Table 6-5 for details). A double fault cannot be caused by multiple UD exceptions but can be caused by multiple simultaneous GP or PF exceptions.

2.3 Execution Ports

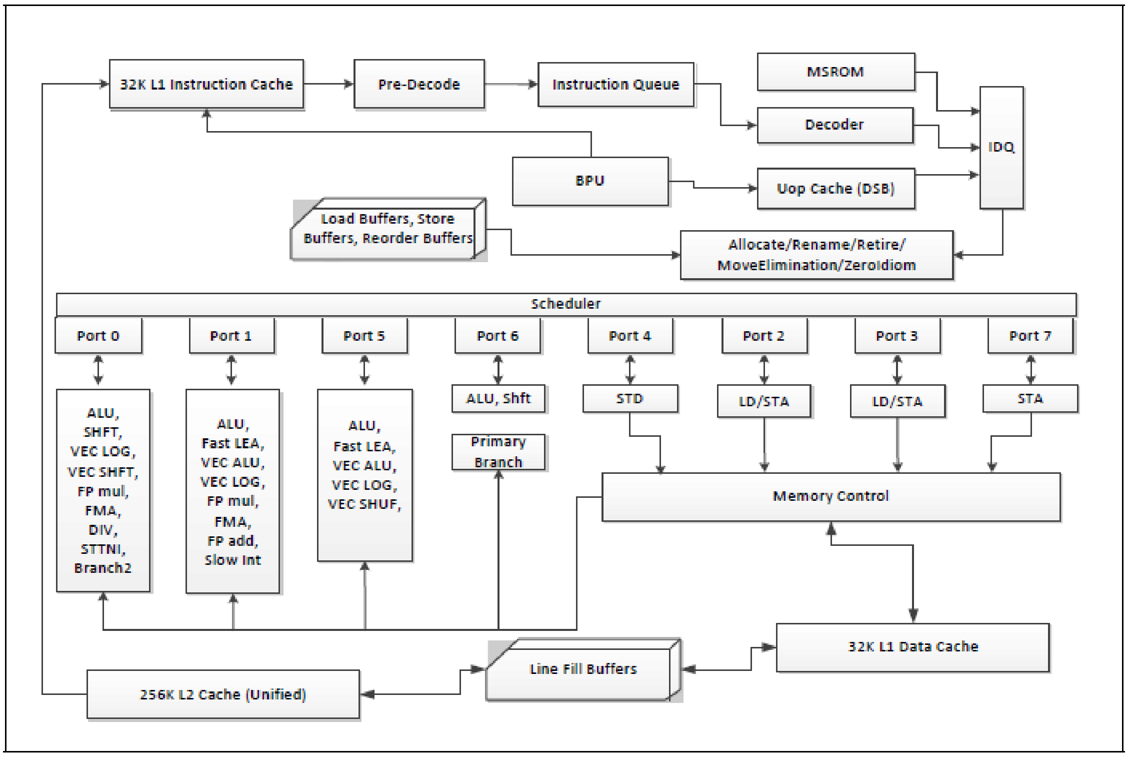

32K L1 Instruction Cache
Pre-Decode
Instruction Queue
MSROM
Decoder
IDQ
BPU
Uop Cache (DSB)
Load Buffers, Store Buffers, Reorder Buffers
Allocate/Rename/Retire/ MoveElimination/ZeroIdiom
Scheduler
Port 0
Port 1
Port 5
Port 6
Port 4
Port 2
Port 3
Port 7
ALU, SHFT, VEC LOG, VEC SHFT, FP mul, FMA, DIV, STTNI, Branch2
ALU, Fast LEA, VEC ALU, VEC LOG, FP mul, FMA, FP add, Slow Int
ALU, Fast LEA, VEC ALU, VEC LOG, VEC SHUF,
ALU, Shft
Primary Branch
STD
LD/STA
LD/STA
STA
Memory Control
32K L1 Data Cache
Line Fill Buffers
256K L2 Cache (Unified)

Figure 2.2: Haswell execution ports (Intel, 2016)

Waiting uops are scheduled out-of-order as their dependencies are satisfied (e.g. if a uop takes the output of a second uop as its input, the second uop must finish beforehand) and dispatched to an execution port. The Haswell (and Broadwell) microarchitecture has eight execution ports (also known as execution units) per core. These can execute in

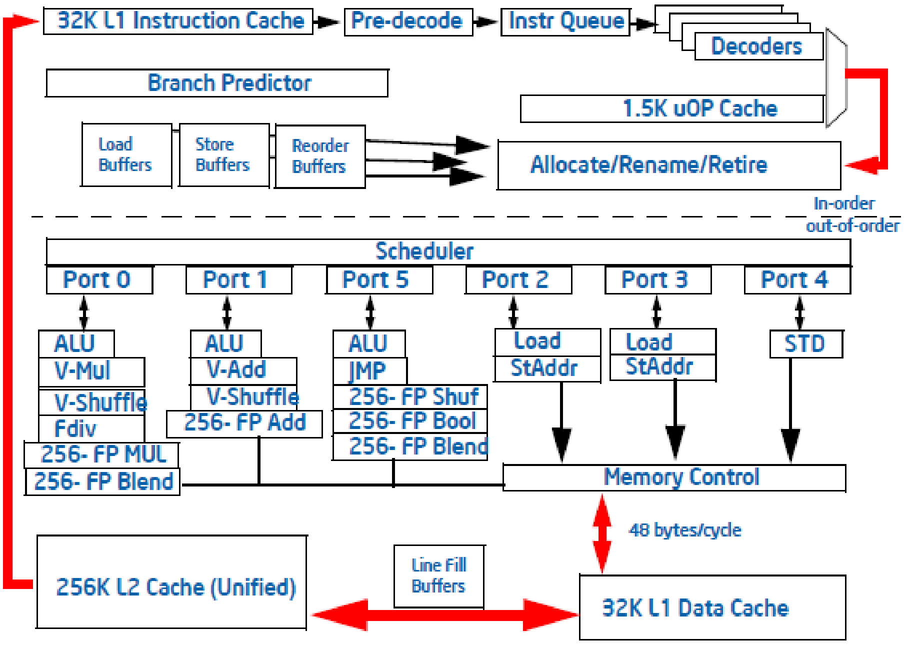


Figure 2.3: Sandy Bridge execution ports (Intel, 2016)

parallel, enabling up to eight uops to be executed per cycle. However, whether this rate is actually reached depends on the types of the uops, as ports have distinct functionality. Ports 2+3, 4, and 7 are dedicated to memory uops (loading/storing an address, storing data, or storing an address respectively), whilst ports 0, 1, 5, and 6 handle all other uops; all four can run ALU uops, but each port also has specialities (such as floating-point addition on port 1). According to an Intel patent (Intel, 1997), counters are used to distribute work evenly between ports when a uop could be run by several ports. The Skylake microarchitecture has very similar execution port functionality to Haswell. Sandy Bridge and Nehalem only have five execution ports and functionality is divided differently (see Figure 2.3). 3.3.2 describes how this distinction of execution ports by functionality category is useful for analysis of undocumented instructions.

3 Automated Instruction Analysis

The goal of this project was to create an automated tool for the analysis of undocumented instructions. This section describes the challenges involved in detecting, executing, and analysing undocumented instructions, and the development of Opcode-Tester, a tool for analysing undocumented instructions on Linux in both user mode and in the kernel.

3.1 Detecting Undocumented Instructions

3.1.1 Instruction Search Space

As described in 1.1.1, with a maximum instruction length of 15 bytes (120 bits) the search space consists of a staggering 2^{120} possible instructions (1.33 x 10^{36}). Even optimistically assuming a testing rate of 1 billion instructions per second, a brute-force search would require 4.21 x 10^{19} years of constant testing, or approximately 98x the estimated age of the universe! A complete brute-force search either to find undocumented instructions or to count the number of documented instructions is therefore completely infeasible, and so strategies are required to reduce the size of the search space. Three such strategies are described below.

It may sound odd that the number of documented instructions is unknown, but in fact no official count or formal semantics of the instruction set exists (Heule et al., 2016). The most authoritative public source of information on Intel's implementation of the ISA is XED (Intel, 2018), an instruction encoding and decoding library; it is developed by Intel and can correctly identify many unusual valid instruction encodings which are poorly documented and considered invalid by other disassembly tools. It recognises 1569 instruction classes (iclasses), which roughly correspond to assembly mnemonics. However, iclasses can have many possible encodings (MOV only has over 30), with each encoding taking a range of different values. There are 6290 recognised instruction forms (iforms) which also attempt to distinguish between different encodings (mostly different addressing modes), but even these do not distinguish between every possible instruction variation; for example, many iforms do not take prefixes or scalable operand widths into account. As iforms do not take into account every possible bit-level instruction variation, a comparison of the number of iforms with the size of the instruction space is unfair, but this remains the closest available figure for the number of documented instructions.

3.1.2 Approach 1 - Manual Targeting

The most obvious strategy is to manually determine target instruction ranges for investigation. This might be prompted by noticing unusual behaviour after a programmer error in assembly or machine code, or by noticing prominent gaps in the documented opcode maps. Historically almost all investigations of undocumented instructions and undocumented instruction behaviour have used this strategy. An example of a tool using this approach is the Corkami Standard Test, which identifies selected undocumented opcode behaviour on Windows (Albertini, 2011). Unfortunately this strategy is exceptionally time-consuming and requires a deep understanding of the ISA to effectively identify targets; clearly only a tiny subset of the instruction space can be searched and the identification of new targets cannot be automated.

3.1.3 Approach 2 - Opcode Search

A second approach is to do a brute-force search of just the 1-3 byte opcode range, reducing the search space to a manageable 16,777,216 (2^{24}) instructions. Although the opcode may form only a small part of a long instruction, it is by far the most important component, as an instruction without an opcode is always invalid and will not execute. Not all opcodes have unique functionality: for example, opcodes often include a register offset (POP is 0x58 + a register index). However, many opcodes do provide unique functionality and even opcodes which use offsets have a small number of possible offsets, so the search does not need to continue far to find the next unique opcode. Given this, it seems reasonable to reduce the search space to this range, although this does of course prevent the detection of undocumented instructions which have unique functionality due to other elements (such as prefix combinations). Note also that such a 3-byte search does not entirely cover the opcode range, as a 3-bit opcode field can also be encoded in the ModR/M byte.

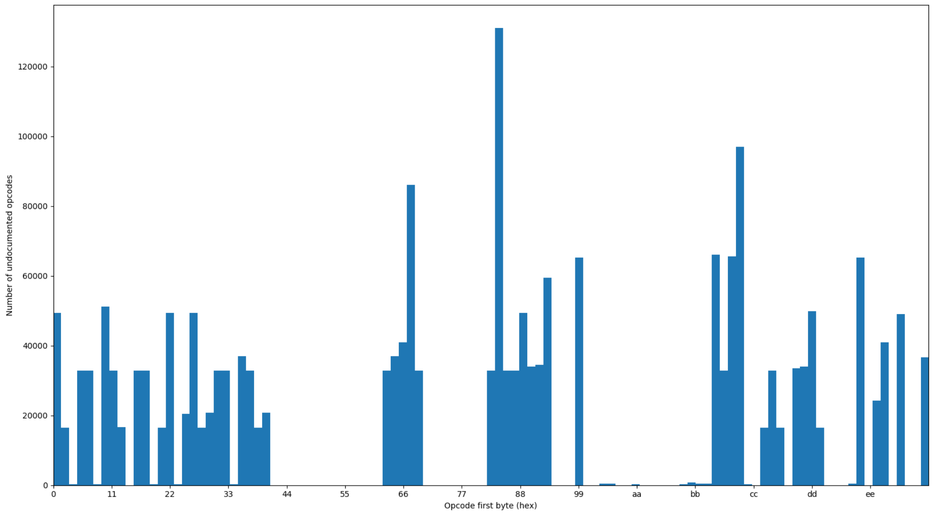


Figure 3.1: A visualisation of the distribution of the first bytes of undocumented opcodes in the 1-3 byte opcode search space

This approach found 5,884,209 potential undocumented instructions on Broadwell in 64-bit mode; this reduced to 1,991,821 when all microarchitectures and all machine modes were included. 3.1.3 shows the distribution of this latter set of instructions by first byte, providing a potential indication of where a brute-force search of full 15-byte instructions could be most effective. These results are particularly interesting because the one-byte opcode range is often considered to be fully explored, yet there were over 160 potentially undocumented opcodes found in this space. Of course, it is likely most of these will not be valid and will fail to execute, but this can only be verified through testing. Unfortunately, testing the opcodes generated by this search is problematic, as many are not full valid instructions: they require additional elements such as addressing modes and operands. The approach to user-mode exception handling described in 3.2.1 fails frequently as the opcodes trigger chains of segmentation faults or stack smashing; these exception chains currently cannot be recovered from and the only solution is to kill the program. Even after extensive manual blacklisting of opcode ranges this occurs sufficiently often that testing the full opcode space is infeasibly slow. This approach could still potentially be incorporated into OpcodeTester, but requires further investigation to determine how to handle or prevent these errors.

3.1.4 Approach 3 - Tunneling

The tunneling algorithm was developed by Domas as part of the Sandsifter tool, the only known existing tool which focuses on discovering undocumented instructions rather than simply undocumented behaviour of existing instructions (Domas, 2017).

The algorithm carries out a depth-first search, reducing the search space by assuming instruction length changes signal a change in instruction functionality (and thus signal an area worthy of further investigation). It begins with a 15-byte zero buffer and then proceeds as follows: the last byte is incremented, and the instruction length is observed; if the instruction length changes, incrementation continues with the new last byte. When the last byte reaches FF, it is rolled over to 00 and the second-to-last byte is then incremented. According to Domas focusing on instruction length changes ensures the algorithm focuses on only "meaningful" instruction bytes such as prefixes, opcodes and operand selection bytes.

The true instruction length is determined by placing the instruction at a page boundary. Initially, the first byte is placed on the last byte of an executable page; if the decoder tries to fetch two bytes or more (i.e. the instruction length is two bytes or more), a page fault occurs; the next byte of the instruction is moved onto the first page and the process is repeated until the instruction fetch generates no more page faults. Instruction fetch page faults are distinguished from other page faults (for example a successful executing instruction causing a page fault) by checking the address in the CR2 register. Domas notes that this technique is particularly interesting because it can be used to determine the "length" of instructions which do not exist (with their length being simply the point at which the decoder stopped attempting to decode the invalid instruction); this provides hints about the operation of the decoder. Unexplained decoder behaviour is discussed further in 3.2.2.

Domas observed this approach reduced the search space to around 100,000,000 instructions, although in testing for this project figures closer to 800,000,000 were observed; as the algorithm is guided by the instruction decoder, the number of instructions searched will of course vary by microarchitecture. Nevertheless this is a substantial improvement, making a brute-force search feasible in 8-10 hours. A crucial advantage of tunneling is that the produced instructions are much less likely to cause catastrophic errors than the instructions produced by opcode search. This is due to the fact Sandsifter only increments one byte at a time (keeping as many bytes as possible at 00 reduces the risk of corrupting the program state through a memory modification, for example via modifying the .data segment) and is guided by the instruction decoder (so instructions are much more likely to be of the correct length including any required addressing and operand bytes).

Sandsifter

Sandsifter is a very useful tool and, as the first automated tool for discovering undocumented instructions, set an important precedent for the development of public CPU auditing tools. As the tunneling algorithm is currently the best known approach for searching the instruction space and the tool handles exceptions very effectively, Sandsifter was chosen as the detection mechanism for OpcodeTester (simply taking Sandsifter log files - or similarly formatted instruction lists - as input for subsequent analysis). However, both the tool and the algorithm are not without their flaws.

The greatest issue is the rate of false positives. Sandsifter depends on the Capstone disassembler to identify whether an instruction is documented, and Capstone fails to recognise many instructions which XED recognises as valid. In testing the vast majority of detected "undocumented" instructions were found to be known to XED and reported as valid for the given processor in 64-bit mode. Furthermore, the tunneling algorithm assumes that the CPU will always stop decoding after a single instruction, but some "undocumented" instructions reported by Sandsifter are decoded by XED as two or more instructions. Notably, 00 00 can not only form part of the addressing mode or be an immediate operand; depending on the opcode of the main instruction it can also be decoded as ADD, so it cannot be assumed to be harmless "padding" of the 15-byte instruction buffer with no effect on execution. Note however that XED can decode for all machine modes (and does so by default in the command-line tool) and so an instruction which is genuinely undocumented in 64-bit mode may be incorrectly reported as documented because it is in another mode. This is commonly seen with 0x40 to 0x4f: in 64-bit mode these are REX prefixes (when in a prefix position), but XED decodes these as separate INC and DEC instructions (they only have this functionality in 32-bit mode). This does not occur in OpcodeTester as 64-bit mode is specified for decoding.

Secondly, the tool assumes that any instruction which throws SIG_ILL in user mode (i.e. causes a UD exception) cannot be a valid undocumented instruction. This is incorrect as there are a range of reasons why a valid instruction may throw UD; for example, this can occur if an instruction is run in the wrong privilege level or with the wrong control register values. This incorrect assumption invalidates Sandsifter's claim that it can be used to detect undocumented instructions in other protection rings from user mode (it assumes that such privileged instructions would always cause SIG_SEGV). Fortunately this issue can be corrected by amending one line of the tool's code (line 299 of sifter.py, to remove the condition that the signum is not equal to SIG_ILL). Note that this modification causes Sandsifter to identify a much larger number of undocumented instructions and so may lead to excessive RAM consumption and crashes; using the –low-mem command line option for Sandsifter helps to prevent this. Of course, an "ideal" undocumented instruction would successfully execute in user mode, but once false positives are removed these occur very rarely in a Sandsifter search (for example, 64 out of 800,000,000 tested instructions on Intel i7-5600U).

3.2 Executing Undocumented Instructions

The OpcodeTester tool is composed of two main elements, a user mode program (for testing instructions in ring 3) and a Linux kernel driver (for testing instructions in ring 0). For simplicity the user mode program will be referred to simply as 'the program' in this section.

3.2.1 Ring 3 Program

Firstly, how can undocumented instructions be executed when they have no assembly mnemonic? There are well-known strategies for executing machine code in user mode from shellcode exploits. Listing 5 shows the strategy used by OpcodeTester, which allows machine code to integrate well with larger programs in C rather than requiring a dedicated hex file. Hex is entered into an unsigned char array (other entry formats will also work, for example entering the corresponding integer values, but it is important the array is unsigned) and the page containing the char array is made executable. Note that the pointer passed to mprotect must be aligned to a page boundary or the protection change will fail. The char array can then be executed by creating and calling a function pointer to it. Note the function prelude '0x55' and prologue '0x5d 0xc3' surrounding the opcode (to push the frame pointer onto the stack at the beginning, then pop it from the stack at the end and return from the function); this is the bare minimum to avoid a segmentation fault, as GCC expects the code to behave like a regular C function.

This example code will successfully execute valid instructions. The opcode used in the example code, 0f0d04, is a relatively unknown PREFETCH instruction (falsely flagged by Sandsifter as undocumented) so should succeed, but any instruction which throws an exception will result in the program being killed before it reaches the printf statement. To prevent this, a signal handler is also required; this can be used to catch exceptions, log the exception type, and restore the program to a valid state. The most crucial aspect of this is changing the instruction pointer to point to a valid instruction, as otherwise the return address will still be the address of the faulting function, and the program will fault in a loop. OpcodeTester uses the approach used by Sandsifter: a global label is created in assembly immediately after the location in the program where the instruction is tested. In the exception handler the instruction pointer can simply be changed to the address of the label.

Note that care must be taken when working with opcodes in hex. The biggest issue is that functions which return upon finding the null terminator in an array (e.g. strlen) will terminate upon encountering a 00 hex byte, potentially skipping part of the instruction. It is therefore necessary to record the length of the instruction when it is first input or read in from a file and maintain this record throughout the lifetime of the instruction.

3.2.2 Ring 0 - Kernel Driver

The kernel driver is a character driver designed to work in tandem with the program. The program writes an instruction to the kernel driver, which then attempts to execute the instruction in ring 0 whilst determining as accurately as possible the number of clock cycles taken by the instruction, and finally the program reads the recorded statistics back from the driver.

Undocumented instructions can be executed in the kernel with the same technique as in user mode. Memory must be made executable with vmalloc rather than mprotect, but this makes the initialisation easier than in user mode as it is unnecessary to calculate the address of the page boundary. See Listing 5 for a minimal example of this.

The most challenging aspect is exception handling, as this differs significantly from user mode. Executing unknown code with kernel privileges is of course higher risk than in user mode, and a great deal of development time was spent ensuring the kernel driver could reliably handle exceptions without crashing the system or causing any data loss.

With Linux's default exception handling, when an exception is thrown by an instruction executed in the driver the user program is then killed (because the exception is triggered by the program's interaction with the driver). This is problematic, as such exceptions occur frequently; manually restarting the program for each instruction is not an option for testing hundreds of thousands or millions of instructions, and automated restarts are still slow in comparison with "unkilled" execution and increase the likelihood of program state corruption and crashes. Unlike signal handling in user mode programs, custom exception handling in kernel drivers is very poorly documented as it is (quite reasonably) assumed that a driver will not intentionally throw exceptions. Several options were investigated for modifying the exception handling, including using Systemtap to change the return values of exception handling functions and modifying the IDT. However these proved very challenging to implement stably and have the major disadvantage that they change the behaviour of all exceptions handled by the hijacked handler (or the behaviour of all exceptions, in the case of the Systemtap hook). As the faulting instructions are also tested in ring 3 in the user program, there will certainly be other exceptions thrown and these should not be handled with the custom logic for the kernel driver.

After much investigation, a solution was developed using a die notifier. Die notifiers can be considered roughly equivalent to signal handlers in user mode; their capabilities are less flexible, as there are a range of exception handling actions the kernel takes before it signals notifiers which cannot be bypassed. The code shown in 3.2.2 is the core Linux kernel exception handling function for UD and most other exceptions (found in arch/x86/kernel/traps.c). The notify_die function notifies all die notifiers in the notification chain, and if the chain returns NOTFIY_STOP, the code in the if statement including do_trap (which prints a kernel oops and kills the user program) is skipped. However, if the die notifier takes no other action the system will freeze in an infinite loop, as the instruction pointer is still at the address of the faulting instruction. This means the handler returns to the faulting instruction, faults again, returns to the faulting instruction again, faults again, and so forth. It is therefore crucial that the die notifier modifies the instruction pointer to point to a valid instruction. The notifier must also re-enable local interrupts and should implement a limit on the number of times it will handle exceptions caused by the same instruction; this prevents the system hanging in an infinite loop if an exception which cannot yet be handled correctly is encountered. Notifiers also appear to be informed of *all* die events, not just those affecting their process, so the notifier must check that the exception is occurring at an expected time (e.g. immediately after executing an instruction - this can be easily implemented with a boolean value which is modified in the lines before and after the instruction is tested) and that it is of an exception and/or event type which the notifier is designed to handle.

s

UD Exception Handling

The global assembly label technique used to modify the instruction pointer in user mode fails in the kernel, and so the instruction pointer must be incremented manually past the faulting instruction. This revealed some very interesting behaviour: for instructions 4 bytes in length or longer, the return address does not point to the first byte of the instruction as would be expected, but to the third byte. (Note that behaviour is as expected for instructions 1-3 bytes in length.) This contradicts Intel's documentation

Listing 3.1: Traps.c exception handling code

```
static void do_error_trap(struct pt_regs *regs, long error_code, char *str,
                          unsigned long trapnr, int signr)
{
        siginfo_t info;

        RCU_LOCKDEP_WARN(!rcu_is_watching(), "entry code didn't wake RCU");

        /*
         * WARN*()s end up here; fix them up before we call the
         * notifier chain.
         */
        if (!user_mode(regs) && fixup_bug(regs, trapnr))
                return;

        if (notify_die(DIE_TRAP, str, regs, error_code, trapnr, signr) !=
                        NOTIFY_STOP) {
                cond_local_irq_enable(regs);
                clear_siginfo(&info);
                do_trap(trapnr, signr, str, regs, error_code,
                        fill_trap_info(regs, signr, trapnr, &info));
        }
}
```

that all trap and fault exceptions are guaranteed to be reported on an instruction boundary. The exact behaviour of the decoder in such situations is unknown, but it would be logical for instruction fetching to abort as soon as an encoding which cannot be valid is detected, in which case this behaviour would be expected for any instruction with an invalid initial two bytes. However this behaviour is remarkably consistent across all tested instructions, many of which had initial two byte sequences which could begin valid instructions.

This behaviour could be investigated further by either replacing Linux's IDT entry for UD or by testing UD in a purpose-built minimal operating system, as it is possible that it is Linux's exception handling rather than the processor that is responsible for this incorrect return address. No indication of this has been found so far in the kernel code involved in handling UD, although entry_64.S still needs more thorough investigation.

Using the die notifier technique OpcodeTester can reliably handle UD exceptions on a large scale; 500,000 faulting instructions have been tested consecutively with no loss of system stability. (Note that, unfortunately, this is the usual scenario when testing undocumented instructions: the vast majority do not successfully execute in ring 3 or ring 0.) One remaining issue is specific to the documented UD faulting instructions UD0, UD1 and UD2. The Linux kernel uses these to indicate bugs and prints a kernel bug message to the kernel log when these are executed; unfortunately this occurs before notify_die and so cannot be skipped using the die notifier method. These bug messages are harmless spam when the instructions are only executed a small number of times. However, by default in Ubuntu there is no limit on log file size, so if these instructions are executed a very large number of times the logs may expand to use up to 100% of the disk; this will cause the system to crash and subsequently prevent it from booting until disk space is cleared from the recovery console. Documented methods of setting limits on log file size appear to be ignored by the operating system, so the only solution is to avoid large-scale testing of UD0, UD1 or UD2 (or any other action which prints a large number of log messages, for example a printk with statistics after each instruction execution) in the kernel, or to regularly delete log files during testing in such situations.

It is important to note that there is no documented error code for UD, so on Linux the si_code field of the siginfo_t struct is always filled with ILL_ILLOPN (illegal operand - see traps.c). This is an unusual default case and if unknown may cause time to be wasted investigating alternative operands for an opcode which is actually entirely invalid; it would seem far more logical to return ILL_ILLOPC (illegal opcode) instead, and indeed this is the error code which is returned by default on many other architectures. The opcode may also *not* be invalid, and might instead report one of the alternative error codes designed for other architectures if this were supported by x86 processors (illegal addressing mode, privileged opcode, privileged register etc). Such ambiguity is frustrating but unfortunately cannot currently be resolved. It remains a possibility however that there may be an undocumented method of determining the UD type, as this would be a very useful debugging feature for Intel's internal verification of the decoder. Also note that not all exceptions use do_error_trap: for example, do_general_protection is used for GP. This also calls notify_die so the same NOTIFY_STOP strategy can be used, but the code before and the skipped code after is different and this must be taken into account when defining how the die notifier handles the exception. In testing, UD, GP and PF were the only observed exceptions resulting from testing undocumented instructions.

3.3 Analysing Instruction Functionality

Performance monitoring provides a wealth of information about the behaviour of executed instructions, and this can be used to infer the functionality of undocumented instructions provided that they do successfully execute. The OpcodeTester tool measures clock cycles and uses execution port profiling to make a best guess about an instruction's functionality category.

3.3.1 Clock Cycles

Clock cycles are notoriously difficult to measure accurately, especially at the extremely fine-grained resolution of a few bytes of machine code. OpcodeTester follows an adapted version of the kernel driver measurement approach suggested in an Intel white paper (Paoloni, 2010). A combination of RDTSC, RDTSCP and other serialising instructions is used to ensure that the instruction under test and the two reads of the time stamp counter are not executed out-of-order. Measuring in the kernel means that preemption and local interrupts can be disabled to reduce the possibility of the measurement being skewed (for example by the kernel driver being descheduled). However, the resolution is still too large to accurately measure a single instruction execution, and so the instruction under test is executed multiple times per measurement. After repeated testing (currently 1000 repetitions per instruction), the overhead of the measurement (calculated by runtime profiling) is removed from the results before dividing by the number of executions to approximate the actual runtime of the instruction, and then the results are averaged. The kernel driver reports the minimum time, the mean time and the mean of the smallest times, as it was observed in testing with known instructions that some timings are extremely large and skew the mean away from the expected value; these results may be caused by frequency scaling or entry into SMM. The measurement allows clear distinguishing of NOP instructions, simple instructions, and complex instructions. Currently it is not reliable enough to be compared with the actual throughputs of known instructions to guess the possible functionality of an instruction, but such a comparison could be made after runtime profiling of known instructions using the same measurement. A target for further investigation would be to compare the accuracy of RDTSCP with the clock cycle performance counters; some of these counters account for processor nuances such as frequency scaling.

3.3.2 Execution Port Profiling

As described in 2.3, execution ports handle distinct functionality categories of uops. For example, on Haswell/Broadwell port 6 handles ALU, shift and branch uops. The number of uops executed on each port can be accurately measured with performance counters; after accounting for the noise created by the current system load, these counts quite closely match the expected behaviour (as documented by Fog, 2018) when known instructions are tested. Using microarchitecture-specific heuristics a “best guess” of an instruction’s functionality can then be determined based on its uop counts per port. The port counts can be combined with other performance counters (for example the floating-point instruction counters, branch counters, or memory operation counters) to improve the accuracy of this guess. These heuristics assume that the instruction under test is being tested at a sufficient rate to saturate the execution ports, so that every port which is capable of executing uops from that instruction is active, and such analysis is of course only possible if an instruction successfully executes. However, it may be possible to use other performance counters to learn about the possible functionality of faulting instructions; 4.0.3 discusses this in more detail.

4 Further Research

4.0.1 Further Development of Auditing Tools

Much further work is needed to expand the capabilities of OpcodeTester. For example, undocumented instructions in SGX, SMM, other machine modes, and non-Intel processors remain unexplored; as an entirely separate coprocessor, Intel ME is much harder to test but is also in need of investigation. One severe limitation of this research is that it was entirely conducted on a single processor model (Intel i7-5600U 2.6Ghz running various versions of Ubuntu) due to a lack of available hardware. Further investigation is therefore necessary on other microarchitectures and on other operating systems (for example, Windows has an entirely different approach to exception handling).

Instruction analysis could be developed further: there are a wide variety of other performance counters available, and profiling could be tailored to the behaviour of known instructions on the system under test (beyond simply accounting for noise) rather than on the overall microarchitecture. Additionally, instruction detection could be improved by implementing stable error handling for the opcode search strategy and developing new algorithms to reduce the search space. Investigating the behaviour of the instruction decoder could also improve detection (for example, by determining exactly what triggers an instruction fetch to end, so that false positives which are actually multiple documented instructions could be avoided). Detecting undocumented instruction behaviour which is triggered only by certain machine states (such as by register 'password' values) remains an unsolved problem, as the search space over all instructions and all possible states is vastly larger than the regular instruction search space. One possible approach would be to speculatively execute instructions (see below) and examine them for unexpected speculative behaviour (which would presumably occur before any checks of the required 'password' values had completed).

Finally, there are many other aspects of CPU behaviour beyond undocumented instructions which remain relatively unexplored and for which no public auditing tools exist. As presented in 1.2, there is an urgent need for the development of such tools so that undocumented behaviour can be brought to light and any serious bugs or security vulnerabilities can be identified and mitigated.

4.0.2 Unexplained Decoder and Exception Behaviour

As described in 3.2.2, it is still unclear why the return address for an UD exception appears to point to the third byte of the invalid instruction for instructions of four bytes or more. Given Intel's claim that instructions faulting with UD are never executed, the expected behaviour would be a return address pointing to the first byte, yet this observed behaviour was consistent across hundreds of thousands of instructions. This suggest the actual behaviour of the instruction decoder differs from the documented behaviour.

Instructions were also occasionally observed to throw GP (reported in user space as SEGV_ACCERR) rather than the UD exception normally seen when executing the same instructions. According to Intel's documentation, it should not be possible for an UD instruction to instead throw GP, as UD is a higher priority exception and should always occur before GP as it occurs when decoding an instruction rather than during instruction execution. This behaviour is interesting as it suggests the possibility of a hardware bug (there are errata on certain processor models in which an incorrect exception is thrown under certain conditions, but none of these concern UD) or that the tested instructions are valid (but privileged, so throw GP) under certain conditions which may occur spontaneously during testing (for example a specific register state). Unfortunately this behaviour is difficult to reproduce so has not yet been investigated further.

4.0.3 Speculative Execution of UD Instructions

The developer manual states unequivocally that an "invalid instruction is not executed" because the exception occurs during decoding. However, intriguingly it also states that **"decoding and speculatively attempting to execute an invalid opcode does not generate this exception"** because the UD exception is not generated until the processor attempts to retire the instruction (Intel, 2017c).

It is not clear whether an 'attempt' at execution of a valid but faulting instruction (one valid in some other mode/state, e.g. a privileged instruction) might involve full speculative execution of the instruction or whether the attempt would fail at some early stage in the pipeline. But if it were to involve full speculative execution, it might be possible to detemine an instruction's functionality from examining the microarchitectural traces left behind after the state rollback (as seen in the ongoing disclosure of the Spectre class of vulnerabilities) or by using performance counters. There are a range of performance counters which distinguish between the values for all (retiring and non-retiring) instructions and for only instructions which retired, and from these statistics for instructions which did not retire (i.e. were incorrectly speculated) can be calculated. If faulting privileged instructions with access to privileged memory could be analysed in this way this would be a security vulnerability, the severity of which would depend on the exact functionality (and the level of privilege) of the undocumented instruction(s).

Presumably an attempt to execute an entirely invalid instruction would do nothing, as its function is undefined. However, depending on the hardware logic across the entire pipeline (e.g. the logic of the decoder, the scheduler, the relevant execution port, etc.) it is plausible that an entirely invalid instruction could be interpreted by the hardware logic to represent one or more uops and then be speculatively executed, or begin to be executed and then fail at some (detectable) intermediate stage in the pipeline. This would be a fascinating target for further research because it could not only provide evidence of new undocumented instructions but also provide information about the hardware logic of the decoding and execution pipeline.

5 Conclusion

This paper presented OpcodeTester, a tool for the automated analysis of undocumented instructions. One of the tool's key features is stable testing of faulting instructions in the kernel (in testing, over 500,000 were tested consecutively with no loss of system stability or exececssive output to system logs). Although a small subset of instructions still cause errors (those leading to chains of GP or PF exceptions), such errors simply cause the user mode program to be killed and do not crash the system. This means that for the first time it is feasible to test undocumented instructions in the kernel on a large scale in order to identify privileged undocumented instructions which require execution in ring 0. It is also the first known tool to attempt functionality analysis of undocumented instructions. Using performance counters for execution ports and other processor statistics, undocumented instructions which successfully execute can be grouped into functionality categories. This enables much more rapid identification of instructions worthy of further investigation than if analysis is done manually. Given the tremendous scale of the instruction search space, such large-scale automation of testing and analysis is crucial for effective investigation.

This paper also presented a detailed consideration of the risks of undocumented instructions and the case more generally for the development of public CPU auditing tools. Several targets for further research were presented, such as identifying methods for defending against unknown undocumented behaviour, developing improved algorithms for reducing the size of the instruction search space, and investigating unexplained decoder and exception behaviour discovered during development of the tool. Most notably, it was identified that speculative execution of faulting instructions may allow the functionality of these instructions to be determined using performance counters and/or microarchitectural traces left behind after a state rollback.

Bibliography

Albertini, Ange (Apr. 2011). *CoST (Corkami Standard Test)*. URL: https://code.google.com/archive/p/corkami/wikis/StandardTest.wiki (visited on 05/26/2018) (cit. on p. 14).

Appelbaum, Jacob (Dec. 2013). *To Protect And Infect: The Militarization of the Internet*. URL: http://cryptome.org/2013/12/appelbaum-30c3.pdf (visited on 05/19/2018) (cit. on p. 4).

Ball, James, Luke Harding, and Juliette Garside (Aug. 2013). *BT and Vodafone among telecoms companies passing details to GCHQ*. URL: https://www.theguardian.com/business/2013/aug/02/telecoms-bt-vodafone-cables-gchq (visited on 05/19/2018) (cit. on p. 4).

Becker, Georg T. et al. (2013). "Stealthy Dopant-level Hardware Trojans." In: *Proceedings of the 15th International Conference on Cryptographic Hardware and Embedded Systems*. CHES'13. Santa Barbara, CA: Springer-Verlag, pp. 197–214. URL: https://sharps.org/wp-content/uploads/BECKER-CHES.pdf (cit. on p. 7).

Case, Brian (Mar. 1993). "Intel Reveals Pentium Implementation Details." In: *Microprocessor Report*. Vol. 7, No. 4. MicroDesign Resources. URL: http://datasheets.chipdb.org/Intel/x86/Pentium/070402.pdf (cit. on p. 2).

Collins, Robert (Oct. 1991). *The LOADALL instruction*. URL: http://www.rcollins.org/articles/loadall/tspec_a3_doc.html (visited on 05/19/2018) (cit. on p. 6).

Collins, Robert (May 1998). *The Pentium FooF Bug*. URL: http://www.rcollins.org/ddj/May98/F00FBug.html (visited on 05/19/2018) (cit. on p. 2).

Costan, Victor, Ilia A. Lebedev, and Srinivas Devadas (2017a). "Secure Processors Part I: Background, Taxonomy for Secure Enclaves and Intel SGX Architecture." In: *Foundations and Trends in Electronic Design Automation* 11, pp. 1–248. URL: https://pdfs.semanticscholar.org/eec4/e1a5275fa7d70557bfcd281f8bba25380836.pdf (cit. on p. 7).

Costan, Victor, Ilia A. Lebedev, and Srinivas Devadas (2017b). "Secure Processors Part II: Intel SGX Security Analysis and MIT Sanctum Architecture." In: *Foundations and Trends in Electronic Design Automation* 11, pp. 249–361. URL: https://pdfs.semanticscholar.org/ccfc/6c76716e0a48cd5ccb5202b48017cb951604.pdf (cit. on pp. 5, 7).

Domas, Christopher (2015). "The Memory Sinkhole: An Architectural Privilege Escalation Vulnerability." In: *Blackhat Conference USA 2015*. Las Vegas, NV, USA. URL: https://github.com/xoreaxeaxeax/sinkhole/blob/master/us-15-Domas-TheMemorySinkhole-wp.pdf (cit. on p. 5).

Domas, Christopher (2017). *Breaking the x86 ISA*. Paper. Battelle Memorial Institute. URL: https://www.blackhat.com/docs/us-17/thursday/us-17-Domas-Breaking-The-x86-Instruction-Set-wp.pdf (cit. on pp. iii, 7, 15).

Duflot, Loic (2008). "CPU Bugs, CPU Backdoors and Consequences on Security." In: *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*. ESORICS '08. Berlin, Heidelberg: Springer-Verlag, pp. 580–599. URL: http://dx.doi.org/10.1007/978-3-540-88313-5_37 (cit. on p. 4).

Embleton, Shawn, Sherri Sparks, and Cliff Zou (2008). "SMM Rootkits: A New Breed of OS Independent Malware." In: *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks*. SecureComm '08. Istanbul, Turkey: ACM, 11:1–11:12. URL: http://doi.acm.org/10.1145/1460877.1460892 (cit. on p. 5).

Ermolov, Mark and Maxim Goryachy (Aug. 2017). *Disabling Intel ME 11 via undocumented mode*. URL: http://blog.ptsecurity.com/2017/08/disabling-intel-me.html (visited on 05/19/2018) (cit. on p. 6).

Ermolov, Mark and Maxim Goryachy (Jan. 2018). *How to Hack a Turned-off Computer, or Running Unsigned Code in Intel ME*. URL: http://blog.ptsecurity.com/2018/01/running-unsigned-code-in-intel-me.html (visited on 05/19/2018) (cit. on p. 6).

Fachat, André (2013). *PET index - Killer Poke*. URL: http://www.6502.org/users/andre/petindex/poke/index.html (visited on 05/19/2018) (cit. on p. 3).

Fog, Agner (Apr. 2018). *Instruction Tables for Intel, AMD and Via CPUs*. URL: http://www.agner.org/optimize/instruction_tables.pdf (visited on 05/26/2018) (cit. on p. 21).

Goodin, Dan (Nov. 2010). *'Super-secret' debugger discovered in AMD CPUs*. URL: https://www.theregister.co.uk/2010/11/15/amd_secret_debugger/ (visited on 05/19/2018) (cit. on p. 6).

Gruss, Daniel et al. (2016). "Flush+Flush: A Fast and Stealthy Cache Attack." In: *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*. DIMVA 2016. San Sebastian, Spain: Springer-Verlag, pp. 279–299. URL: https://doi.org/10.1007/978-3-319-40667-1_14 (cit. on p. 3).

Hawkes, Ben (Dec. 2012). *Notes on Intel Microcode Updates*. URL: http://inertiawar.com/microcode/ (visited on 05/19/2018) (cit. on p. 4).

Heule, Stefan (Mar. 2016). *How Many x86-64 Instructions Are There Anyway?* URL: https://stefanheule.com/blog/how-many-x86-64-instructions-are-there-anyway/ (visited on 05/19/2018) (cit. on p. 1).

Heule, Stefan et al. (June 2016). "Stratified Synthesis: Automatically Learning the x86-64 Instruction Set." In: *Programming Language Design and Implementation (PLDI)*. ACM. DOI: 10.1145/2908080.2908121. URL: http://dx.doi.org/10.1145/2908080.2908121 (cit. on pp. 2, 13).

Hornby, Taylor (July 2017). *@DefuseSec analysis of RDRAND's code*. Unfortunately Hornby has not written about his work on RDRAND, so this text and its mirrors are the only known sources. URL: https://gist.github.com/mimoo/5957603f5aa5f0cded33e55f930644cb (visited on 05/19/2018) (cit. on p. 5).

Intel (Nov. 1997). *Method and apparatus for binding instructions to dispatch ports of a reservation station*. URL: https://patents.google.com/patent/US5689674 (visited on 05/26/2018) (cit. on p. 12).

Intel (Feb. 2010). *Intel® Quiet System Technology 2.0 Programmer's Reference Manual*. Revision -001. URL: https://software.intel.com/sites/default/files/af/73/Intel_QST_Programmers_Reference_Manual.pdf (cit. on p. 6).

Intel (June 2016). *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 248966-033. URL: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf (cit. on pp. 4, 11, 12).

Intel (Nov. 2017a). *6th Generation Intel® Processor Family Specification Update*. 332689-012. URL: https://www.intel.com/content/dam/www/public/us/en/documents/

specification-updates/desktop-6th-gen-core-family-spec-update.pdf (cit. on p. 1).

Intel (Aug. 2017b). *7th Generation Intel® Processor Family and 8th Generation Intel® Processor Family for U Quad Core Platforms Specification Update*. 334663-009. URL: https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/7th-gen-core-family-spec-update.pdf (cit. on p. 2).

Intel (Dec. 2017c). *Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*. 325462-065US. URL: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf (cit. on pp. 4, 5, 9–11, 23).

Intel (Apr. 2018). *Intel X86 Encoder Decoder (Intel XED)*. URL: https://github.com/intelxed/xed (visited on 05/26/2018) (cit. on p. 13).

Johnson, Nephi (Apr. 2011). *Interesting Behaviors in x86 Instructions*. URL: https://d0cs4vage.blogspot.com/2011/04/interesting-behaviors-in-x86.html (visited on 05/26/2018) (cit. on p. 9).

Kocher, Paul et al. (Jan. 2018). "Spectre Attacks: Exploiting Speculative Execution." In: *ArXiv e-prints*. arXiv: 1801.01203. URL: https://spectreattack.com/spectre.pdf (cit. on p. 3).

Koppe, Philipp et al. (2017). "Reverse Engineering x86 Processor Microcode." In: *Proceedings of the 26th USENIX Security Symposium*. Vancouver, Canada: USENIX Association, pp. 1163–1180. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions (cit. on p. 4).

Lipp, Moritz et al. (Jan. 2018). "Meltdown." In: *ArXiv e-prints*. arXiv: 1801.01207. URL: https://meltdownattack.com/meltdown.pdf (cit. on p. 3).

Matrosov, Alex (2017). "Betraying the BIOS: Where the Guardians of the BIOS are Failing." In: *Blackhat Conference USA 2017*. Las Vegas, NV, USA. URL: https://www.blackhat.com/docs/us-17/wednesday/us-17-Matrosov-Betraying-The-BIOS-Where-The-Guardians-Of-The-BIOS-Are-Failing.pdf (cit. on p. 5).

Menn, Joseph (Dec. 2013). *Exclusive: Secret contract tied NSA and security industry pioneer*. URL: https://www.reuters.com/article/us-usa-security-rsa/exclusive-secret-contract-tied-nsa-and-security-industry-pioneer-idUSBRE9BJ1C220131220 (visited on 05/19/2018) (cit. on p. 4).

Morgan, Timothy Prickett (May 2013). *Intel's answer to ARM: Customisable x86 chips with Hidden Powers*. URL: https://www.theregister.co.uk/2013/05/20/intel_chip_customization/ (visited on 05/19/2018) (cit. on p. 2).

Nicole Perlroth, Jeff Larson and Scott Shane (Sept. 2013). *N.S.A. Able to Foil Basic Safeguards of Privacy on Web*. URL: https://www.nytimes.com/2013/09/06/us/nsa-foils-much-internet-encryption.html (visited on 05/19/2018) (cit. on p. 4).

Paoloni, Gabriele (Sept. 2010). *How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures*. URL: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf (visited on 05/26/2018) (cit. on p. 20).

Peterson, Nick and Nemanja Mulasmajic (2018). *POP SS/MOV SS Vulnerability*. Paper. Everdox Tech and Triplefault. URL: http://everdox.net/popss.pdf (cit. on p. 2).

Rutkowska, Joanna (2015). *x86 Considered Harmful*. Paper. Invisible Things Lab. URL: https://blog.invisiblethings.org/papers/2015/x86_harmful.pdf (cit. on p. 7).

Schmidt, Jürgen (May 2018). *Exclusive: Spectre-NG - Multiple new Intel CPU flaws revealed, several serious*. URL: https://www.heise.de/ct/artikel/Exclusive-Spectre-NG-Multiple-new-Intel-CPU-flaws-revealed-several-serious-4040648.html (visited on 05/19/2018) (cit. on p. 3).

Schneier, Bruce (Jan. 2014a). *DEITYBOUNCE: NSA Exploit of the Day*. URL: https://www.schneier.com/blog/archives/2014/01/nsa_exploit_of.html (visited on 05/19/2018) (cit. on p. 5).

Schneier, Bruce (Jan. 2014b). *IRONCHEF: NSA Exploit of the Day*. URL: https://www.schneier.com/blog/archives/2014/01/nsa_exploit_of_1.html (visited on 05/19/2018) (cit. on p. 5).

Skochinsky, Igor and Nicola Corna (Dec. 2017). *Intel ME: Myths and Reality*. URL: https://events.ccc.de/congress/2017/Fahrplan/system/event_attachments/attachments/000/003/391/original/Intel_ME_myths_and_reality.pdf (visited on 05/19/2018) (cit. on p. 6).

Spiegel, Der (Dec. 2013). *Inside TAO: Documents Reveal Top NSA Hacking Unit*. URL: http://www.spiegel.de/international/world/the-nsa-uses-powerful-toolbox-in-effort-to-spy-on-global-networks-a-940969.html (visited on 05/19/2018) (cit. on p. 3).

Storm, Darlene (Jan. 2014). *17 exploits the NSA uses to hack PCs, routers and servers for surveillance*. URL: https://www.computerworld.com/article/2474275/cybercrime-hacking/17-exploits-the-nsa-uses-to-hack-pcs--routers-and-servers-for-surveillance.html (visited on 05/19/2018) (cit. on p. 3).

Ylonen, Tatu (Feb. 2018). *Intel AMT Vulnerability Tracking Page*. URL: https://www.ssh.com/vulnerability/intel-amt/ (visited on 05/19/2018) (cit. on p. 6).

Appendix: Code Listings

Listing 5.1: Minimal code for testing undocumented instructions in user mode (page 1)

```
#define _GNU_SOURCE
#include <sys/mman.h>
#include <unistd.h>
#include <stdint.h>
#include <stdio.h>
#include <signal.h>
extern char resume;
int instructionFailed = 0;

void signalHandler(int sig, siginfo_t* siginfo, void* context){
        /*abort if couldn't restore context - instructionFailed increasing
        means we are stuck in a loop */
        if(instrCurrentlyExecuting){
                if(instructionFailed > 3) exit(1);
                else{
                //switch(sig){ /*check for signal type here */ }
                   instructionFailed++;
                //get execution context, skip faulting instr, reset sign flag
                mcontext_t* mcontext = &((ucontext_t*)context)->uc_mcontext;
                mcontext->gregs[IP]=(uintptr_t)&resume;
                mcontext->gregs[REG_EFL]&=~0x100;
                }
        }
        else{   //unexpected signal, restore default handlers
                signal(SIGILL, SIG_DFL); signal(SIGFPE, SIG_DFL);
                signal(SIGSEGV, SIG_DFL); signal(SIGBUS, SIG_DFL);
                signal(SIGTRAP, SIG_DFL); signal(SIGABRT, SIG_DFL);
        }
}
```

Listing 5.2: Minimal code for testing undocumented instructions in user mode (page 2)

```
int main(int argc, char *argv){
        //register signal handler for all likely signals
        struct sigaction handler;
        memset(&handler, 0, sizeof(handler));
        handler.sa_sigaction = signalHandler;
        handler.sa_flags = SA_SIGINFO;
        if (sigaction(SIGILL, &handler, NULL) < 0    || \
                sigaction(SIGINT, &handler, NULL) < 0    || \
                sigaction(SIGFPE, &handler, NULL) < 0    || \
                sigaction(SIGSEGV, &handler, NULL) < 0   || \
                sigaction(SIGBUS, &handler, NULL) < 0    || \
                sigaction(SIGTRAP, &handler, NULL) < 0   || )
        {
                perror("sigaction");
                return 1;
        }
        unsigned char code[] = "\x55\x0f\x0d\x04\x5d\xc3";
        uintptr_t codePointer = (uintptr_t) &code;
        size_t pageSize = sysconf(_SC_PAGESIZE);
        uintptr_t pagePointer = codePointer & -pageSize;
        int rights =  PROT_READ|PROT_WRITE|PROT_EXEC;
        if( mprotect((void*) pagestart, pageSize, rights) ){
                perror("mprotect");
                return 1;
        }
        instrCurrentlyExecuting = true;
        ((void(*)())code)();                //run the machine code
        __asm__ __volatile__ ("\
                .global resume    \n\
                resume:           \n\
                "
                );
        ;
        instrCurrentlyExecuting = false;
        printf("Instruction executed successfully.\n");
        return 0;
}
```

Listing 5.3: Minimal code for testing undocumented instructions in a kernel driver

```
//returning NOTIFY_STOP skips the call to die() in do_error_trap()
static int opcodeTesterKernel_die_event_handler (struct notifier_block *self,
unsigned long event, void *data){
    struct die_args* args = (struct die_args *)data;
    if (prevExceptionsHandled && opcodeIsExecuting && (args->trapnr == 6) ) {
          //trapnr 6 is UD, 13 is GP, 14 is PF
          opcodeExecutedSuccessfully = 0;
          prevExceptionsHandled++;
          //TODO ADD VM CODE, THIS IS NOT COMPLETE
          if(opcodeByteCount > 2) args->regs->ip += (opcodeByteCount - 2);
          else return 0;
          //switch(args->signr){ /*check for signal type here */ }
         //equiv of unexported cond_local_irq_enable(args->regs);
         if(args->regs->flags & X86_EFLAGS_IF){
                        local_irq_enable();
         }
        return NOTIFY_STOP;
  }
  else return 0;
  //if too many exceptions or an event we cannot handle, play it safe
  //and let the kernel handle the event and kill the program
}

static __read_mostly struct notifier_block opcodeTesterKernel_die_notifier = {
  .notifier_call = opcodeTesterKernel_die_event_handler,
  .next = NULL,
  .priority = 0
};

//code snippet for testing function
opcode = __vmalloc(15, GFP_KERNEL, PAGE_KERNEL_EXEC);
memset(opcode, 0, 15);
opcode[0] = 0x55; opcode[1] = 0x0f;
opcode[2] = 0x0d; opcode[3] = 0x04;
opcode[4] = 0x5d; opcode[5] = 0xc3;
opcodeIsExecuting = true;
prevExceptionsHandled = 0;
((void(*)(void))opcode)();
opcodeIsExecuting = false;
vfree(opcode);

//code snippet for driver init function
register_die_notifier (&opcodeTesterKernel_die_notifier);

//code snippet for driver exit function
//unregistering is crucial to avoid crash when reloading driver
unregister_die_notifier(&opcodeTesterKernel_die_notifier);
```