Dissertation Type: research

DEPARTMENT OF COMPUTER SCIENCE

Undocumented CPU Behaviour on x86 and RISC-V Microarchitectures: A Security Perspective

Catherine Easdon

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Friday 10th May, 2019

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is entirely the work of the author. The views in this document are those of the author and do not in any way represent those of the University.

Signed: Catherine Easdon, Friday 10th May, 2019

Contents

List of Figures

List of Tables

List of Listings

Executive Summary

CPU design is in crisis. Moore's law has failed: semiconductor advances can no longer be relied upon for substantial performance gains, motivating increasingly aggressive pipeline optimisation and faster design cycles as manufacturers struggle to remain competitive. Security has suffered under this approach, as highlighted by the ongoing disclosure of transient execution vulnerabilities. Mitigations are now in place for Meltdown and Spectre, but has the industry learnt anything from them? With little economic incentive for security and increasingly complex microarchitectural designs, it is likely more CPU vulnerabilities will emerge in the future. Open-source CPU auditing tools can help tackle this problem, complementing verification tooling by enabling users to investigate the behaviour of their CPUs.

This project develops one such auditing tool for investigation of undocumented instruction behaviour on the x86 and RISC-V architectures. The tool is used to test three hypotheses across five different microarchitectures: firstly, the hypothesis that undocumented instructions exist on the microarchitectures under test; secondly, that undocumented exception and/or decoding behaviour occurs; and finally, that transient execution of #UD-faulting instructions occurs on Intel Broadwell. The project is the first known investigation of undocumented instructions on RISC-V, with the surprising finding that a commercially-available RISC-V platform not only features undocumented instructions but decodes two entirely undocumented instruction formats. On x86, the project presents new strategies for searching and testing the instruction space, in particular developing a methodology for microbenchmarking of undocumented instructions via speculative execution. It confirms the finding of Canella et al. that #UD-faulting instructions are not transiently executed, demonstrating experimentally that #UD-faulting instructions exhibit uniform behaviour from the instruction decoding stage onwards.

Concretely, this project's main contributions are as follows:

- The first known investigation of undocumented instructions on RISC-V, identifying 2048 undocumented instructions on the HiFive Unleashed (at least 608 of which modify architectural state) and partially reverse-engineering their instruction format and functionality.
- A novel x86 instruction search approach conducting a timing attack on the three-byte opcode space.
- Two novel instruction testing techniques (using TSX RTM and the specpoline construct) which suppress exception generation for significantly improved stability and performance (RTM is 95.8% faster than the established instruction testing technique).
- An experimental investigation of transient execution of #UD-faulting instructions, with results supporting the findings of [1].
- The first CPU fuzzing tool for RISC-V, a version 2 release of the open-source OpcodeTester tool compatible with both RISC-V Linux and SiFive Freedom Metal and additionally incorporating the project's contributions on x86 [2].

The summaries at the beginning of Chapters 3 and 4 list the project's contributions more fully.

Dissertation Outline

Chapter 1 motivates the project, providing a high-level introduction to core architectural concepts and to central issues in CPU security. It presents the design trends and pressures in the CPU industry which led to the extreme complexity of current designs, discusses the broader economic, political, and technical context for CPU security and potential undocumented behaviour, and motivates why undocumented instruction behaviour in particular poses a security concern. Additionally, it summarises prior work regarding undocumented instruction behaviour and presents the project's specific aims and objectives.

Chapter 2 provides technical background in security and computer architecture concepts relevant to the project. It elaborates upon design concepts such as the CPU pipeline, instruction-level parallelism, privilege levels, and caching, providing a deeper understanding of why CPUs are vulnerable to the side-channel and transient execution attacks introduced in Chapter 1. It introduces the ISAs and microarchitectures under test, in particular presenting details of x86 and RISC-V instruction encoding necessary for understanding of Chapters 3 and 4.

The implementation of OpcodeTester and the project's experimental results are split by ISA, with Chapters 3 and 4 presenting the project's contributions on RISC-V and x86 respectively. Chapter 3 describes the instruction testing technique developed for x86 in a previous project, discusses the challenges encountered porting this technique to RISC-V and improving its stability, and provides minimal code examples for instruction testing on RISC-V in user mode on Linux and in machine mode with the Freedom Metal framework. It details experimental results proving hypotheses 1 and 2 correct on the HiFive Unleashed and HiFive1, and describes the reverse-engineering of the undocumented instructions found. Hypothesis 3 was not investigated as neither of the microarchitectures under test support transient execution.

Chapter 4 presents the project's contributions on x86, including a novel instruction search strategy via a timing attack on the three-byte opcode space, and two novel instruction testing techniques employing Intel TSX and the specpoline mechanism. The latter enabled fine-grained investigation of instructions' microarchitectural effects and was used to confirm the findings of [1] regarding transient behaviour of #UD-faulting instructions. Primarily hypotheses 2 and 3 were investigated, with the new testing techniques and search strategy facilitating future work on hypothesis 1.

Chapter 5 critically evaluates the project, summarising its outcomes, discussing the challenges faced, and reflecting on the work's assumptions and limitations. It presents the case for future work porting instruction fuzzing to other architectures and to other hardware components beyond the CPU, discusses the potential for alternative search strategies for the instruction space, and concludes with reflection on the outlook for CPU security.

Supporting Technologies

The following hardware was used in development and testing:

- Hardware provided by my supervisor Dr. Daniel Page
  - SiFive HiFive Unleashed development board, used as the RISC-V Freedom U540 microarchitecture under test
  - SiFive HiFive1 development board, used as the RISC-V Freedom E310 microarchitecture under test
  - Toshiba NB200 laptop, used as the x86 Intel Bonnell microarchitecture under test
  - Intel Galileo Gen. 1 development boards x2, intended for use as Intel x86 microcontrollers under test but unfortunately both bricked by failed firmware updates
- Dell Latitude E7450, personal laptop used as the x86 Intel Broadwell microarchitecture under test and as my primary development environment
- Dell Inspiron 1564, personal laptop used briefly as the x86 Intel Westmere microarchitecture under test before hardware failure

As always, I am indebted to the open-source community, as this project made extensive use of open-source hardware and software. In particular:

- The Linux ecosystem (particularly Ubuntu, the GNU toolchain, Python, and the version-control software Git) was crucial to my development environment
- The RISC-V toolchain and SiFive Freedom SDK enabled development for the HiFive boards
- For x86, the Intel XED disassembler was invaluable as a 'golden reference' for documented instructions [3] and is linked into the final tool; similarly Michael J. Clark's disassembler for RISC-V [4]
- The Sandsifter CPU fuzzer inspired the first version of my OpcodeTester tool, and its tunneling algorithm was used extensively in my investigation of undocumented behaviour on x86 [5]

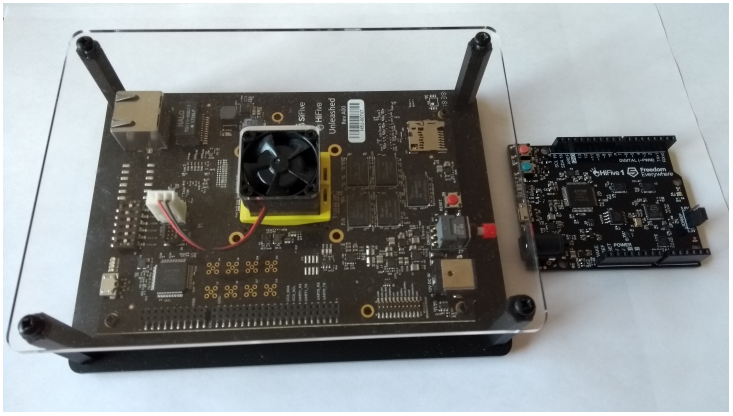


Figure 1: The SiFive HiFive Unleashed (left) and HiFive 1 (right) development boards.

Notation and Acronyms

#GP	:	General Protection Exception
#UD	:	Illegal Opcode Exception
AMD	:	Advanced Micro Devices
BIOS	:	Basic Input/Output System
BPU	:	Branch Prediction Unit
BTB	:	Branch Target Buffer
CPU	:	Central Processing Unit
CISC	:	Complex Instruction Set Computer
(D)RAM	:	(Dynamic) Random-Access Memory
HCF	:	'Halt and Catch Fire' instruction (colloquial)
IC	:	Integrated Circuit
IDQ	:	Instruction Decode Queue
IP	:	Intellectual Property
(I)TLB	:	(Instruction) Translation Lookaside Buffer
ISA	:	Instruction Set Architecture
LLC	:	Last Level Cache
LSD	:	Loop Stream Detector
ME	:	Management Engine
MIPS	:	Microprocessor without Interlocked Pipelined Stages (RISC ISA)
MSR	:	Model-Specific Register
MSROM	:	Microcode Sequencer Read-Only Memory
NDA	:	Non-Disclosure Agreement
NSA	:	National Security Agency
NX	:	No Execute (bit)
OEM	:	Original Equipment Manufacturer
OS	:	Operating System
PSP	:	Platform Security Processor
PRNG	:	(Pseudo-) Random Number Generator
RISC	:	Reduced Instruction Set Computer
RISC-V	:	RISC ISA developed by the RISC-V Foundation
ROB	:	Reorder Buffer
RS	:	Reservation Station
RSB	:	Return Stack Buffer
RTM	:	Restricted Transactional Memory
SDK	:	Software Development Kit
SMM	:	System Management Mode
SoC	:	System on Chip
TSX	:	Transactional Synchronization Extensions
VM	:	Virtual Machine
x86	:	CISC ISA developed by Intel and AMD (includes x86-64/AMD64)

Acknowledgements

It has been a long journey to writing this dissertation and thanks are due to more people than I can possibly name here, but in particular I would like to thank:

- My parents for all their support, and their patience with the fact that for (at least!) the last four years their daughter has been replaced by a thundercloud of stress and chaos;
- My supervisor Dr. Daniel Page, for his encouragement and advice throughout the project, many entertaining discussions about hardware security, and in particular for suggesting I investigate RISC-V in addition to x86. I found the architectural contrast very valuable for my understanding of ISA design and security;
- Ben Marshall, for his advice on verification practices in industry and open-source verification tooling, areas I had very little prior knowledge of;
- My advisor for my previous research project at TU Graz, Dr. Daniel Gruss, for taking a chance on a complete beginner to CPU security and encouraging me to consider further research;
- And my friends in the University of Bristol Expeditions Society (UBES), for being a second family to me throughout my time at university.

Finally, this dissertation is dedicated to RuneScape, wooden swords, and a virtual quest for redberry pie, which improbably set me on the path to this degree back in 2006.

Chapter 1

Motivation

> "The only truly secure system is one that is powered off, cast in a block of concrete and sealed in a lead-lined room with armed guards - and even then I have my doubts."
>
> Gene Spafford [6]

1.1 The Complexity Explosion

Programming early computers such as the ENIAC was a demanding task. In contrast to the ease with which we switch apps on our laptops and phones today, changing the software on the ENIAC required physically adjusting cable connections and switches, taking anywhere from half an hour to an entire day [7]! With the first implementation of the stored-program paradigm in 1948 [8], however, the Central Processing Unit (CPU) was born. Effectively acting as the computer's brain, the CPU fetches instructions from memory, decodes them, either executes them or delegates them to other processing components, and stores the results. Crucially, it is general-purpose: it can execute instructions with a wide range of different functionality, and the instructions it loads can be easily modified without physically modifying the hardware. The instructions the CPU understands are known as its instruction set architecture (ISA), which specifies a contract between the software developer and the hardware designer: software created for a given ISA will run on any CPU which supports that ISA, even though the physical designs (the microarchitecture) of each of those CPUs may be very different.

Whilst early CPUs were formed of many discrete hardware components, CPUs as we know them today were born with the advent of the microprocessor in the 1970s, when CPUs were produced for the first time on a single integrated circuit (IC). One such microprocessor was the Intel 8086, released in 1978. The first CPU for the x86 ISA now dominant on desktop and server computers, it featured just 29,000 transistors at a scale of 3,000 nm each [10]. In the four decades since, there has been an overwhelming explosion of complexity in CPU design: the A12 CPU released in 2018 for the iPhone comprises an astounding *6.9 billion* transistors, each just 7 nm in size[1] [12]! This immense complexity enables the A12 to be approximately 50,000x faster than the 8086[2], but makes it fiendishly complex to verify the CPU's correctness or prove its security.

How did this complexity explosion occur? For many years, advances in semiconductor technology and increasing transistor counts enabled rapid performance improvements. Dennard scaling enabled transistor density to be increased whilst keeping power density constant, and Moore's law - Gordon Moore's famous prediction that CPU transistor counts would *double* every two years [7] - became a self-fulfilling prophecy as the software

[1] Note that these sizes are exploited for marketing purposes, and so what a quoted size actually measures varies [11].

[2] [13] found the average performance of the 8086 to be 0.4 relative to the VAX-11/780 across four benchmarks. This result was used to convert the performance metric relative to the VAX-11/780 in Figure 1.1 of [7] to a metric relative to the 8086, resulting in 4-core Intel Core i7 Extreme 3.2 Ghz performance of over 25,000 relative to the 8086. Finally, the result of [12] that the Apple A12 outperforms a moderately-clocked Skylake CPU in single-threaded performance was used to relate the A12 to the relative performance of the i7. These repeated comparisons inevitably add imprecision to the result and it must be stressed that this figure is for illustrative purposes only.

Figure 1.1: An Intel Core i7-7820X CPU (14nm node), showing the (colourised) die - the IC itself - removed from its heatsink and external casing. The die has been etched to the polysilicon layer and photographed with a reflected light microscope: for reference, it has approximately the same surface area as a one penny coin. Image reproduced from [9].

industry began to rely on this rate of performance improvement to increase the capabilities (and corresponding complexity and inefficiency [14]) of software; in turn, the CPU industry became dependent on this demand for continued sales. Powerful instruction-level parallelism techniques such as pipelining, out-of-order execution, and speculative execution helped compensate for the performance bottlenecks imposed by other hardware components improving at slower rates, such as memory (DRAM) [7].

Over time, however, transistor advances began to face acute physical constraints, and in stark contrast to the 52% per year performance increases from 1986-2003, performance improved just 3.5% per year from 2016-2018 [7]. Dennard scaling reached its limit in 2004, prompting the rise of multicore CPUs composed of multiple interconnected cores optimised for different tasks or used together to further exploit parallelism. However, these too soon encountered constraints. Amdahl's law limits the number of useful cores for a task: if a given percentage of a computation is inherently serial, then no number of cores can achieve a performance improvement factor greater than this value. Heat dissipation, meanwhile, became increasingly challenging with higher transistor counts now (without Dennard scaling) also increasing power density. This produced the phenomenon of *dark silicon*: CPUs could no longer use all their transistors simultaneously. At the very smallest transistor sizes (so-called nodes) today such as 8nm, CPUs can only operate an estimated 50% of their transistors at full frequency at once [15].

With increased transistor counts becoming burdensome rather than beneficial, the pressure to maintain Moore's law has been desperate in recent years, and - combined with intense market competition - has incentivised manufacturers to minimise testing and verification and ship the "most unreliable CPU that can not be detected as unreliable" [16]. It has also motivated the increasingly aggressive use of instruction-level parallelism techniques, which - whilst ingenious - are fiendishly complex to implement in hardware, verify, or formally prove secure. It is easy to envisage the degree to which hardware complexity growth has outstripped the capabilities of verification and test tools; verifying 20,000 transistors is considerably more feasible than verifying 7 billion transistors subject to quantum effects [17]! The ISA itself can pose further verification challenges. x86 in particular has evolved into a monstrously complex beast due to a variety of factors, including maintaining backwards compatibility back to 1978, compensating for weak performance gains via new ISA features [18], and fierce competition between the key manufacturers Intel and AMD [19]. With its 15-byte instructions alone (see Section 2.3) it has 2^{120} different encodings: even testing 1 billion instructions per second, testing every single one of these to ensure they behave as expected would take over 3 billion times longer than the current age of the universe[3].

Throughout this complexity explosion, security has been seriously neglected. Whilst certain hardware 'security' features such as hardware roots of trust and secure enclaves have been added, these have known vulnerabilities and violate the fundamental security principles of least privilege and separation of privilege (see Section 1.2.4). With ever-increasing complexity, ineffective verification, and few economic incentives for manufacturers to prioritise security [16], CPU vulnerabilities are now a serious threat. This was dramatically

[3]Calculated using $4.3 * 10^{17}$ as the approximate current age of the universe in seconds (source: WolframAlpha).

illustrated by the Meltdown and Spectre vulnerabilities in 2018 (with disclosure of related transient execution vulnerabilities still ongoing) [20] [21], which exploited instruction-level parallelism techniques to bypass hardware-enforced security isolation between abstraction layers, such as between the unprivileged user and the privileged operating system (see Section 1.2.4). Performance optimisations traded security for speed, and led, ultimately, to an attacker being able to steal data from your web browser and even read arbitrary memory on your computer remotely via a web page [22] [23]. Yet the potential risks of these optimisations had been known for more than 20 years [24]. How can we fix the complexity explosion and broken economic incentives which led to these vulnerabilities being ignored for so long? And are there yet more serious CPU vulnerabilities lying in wait?

1.2 Untrusting the CPU

To complement the prior section's discussion of trends in CPU complexity, this section introduces the broader economic, political, and technical context for CPU security, discussing the security impacts of global supply chains, governmental surveillance, IP and licensing, and 'leaky' technical abstractions[4].

1.2.1 Global Supply Chains

"Nearly every conceivable component within [the US Department of Defense] is networked...built on inherently insecure architectures that are composed of, and increasingly using, foreign parts...Solving this problem is analogous to complex national security and military strategy challenges of the past, such as the counter U-Boat strategy in WWII and nuclear deterrence in the Cold War." [26]

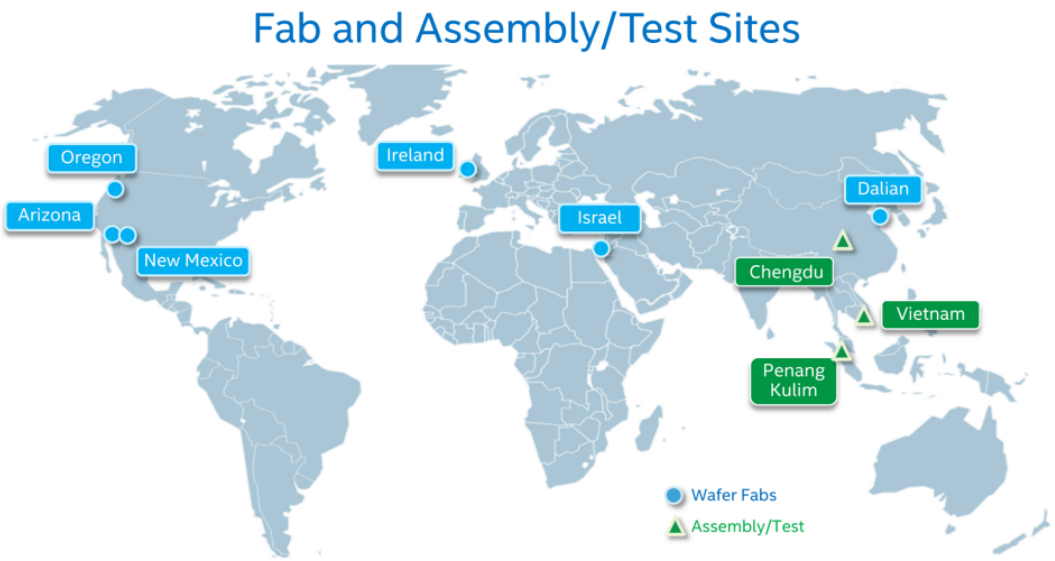


Figure 1.2: A map of Intel's ten main manufacturing sites. Image reproduced from [27].

Historically, the microelectronics industry (including a wide range of IC products, not only CPUs) has had exceptionally high financial barriers to entry. Hardware development cycles - and therefore times to market - are much slower than in software development, and manufacturing at the nanometre scale requires extremely expensive specialised equipment. A proposed new Intel fab in Israel will cost US $11 billion if built [28], and Taiwan Semiconductor Manufacturing Co (TSMC) estimate their fab dedicated to the future 3nm node will cost over US $20 billion [29]. Whilst some integrated device manufacturers (IDMs) continue to both design and manufacture their hardware, many 'fabless' companies such as AMD, Arm, Qualcomm and Broadcomm [30] operate without manufacturing capabilities altogether. They instead produce designs for CPUs, SoCs (Systems on Chips, which integrate all computing components of a system into a single IC) or subcomponents

[4] Credit is due to [25] for inspiring this section's title.

(IP blocks) and then outsource manufacturing to dedicated foundry companies such as TSMC, or license the designs to other companies to adapt, manufacture and distribute. Even IDMs such as Intel have complex global supply chains, as illustrated by Figure 1.2, and this is without even considering their far larger global networks of offices conducting hardware design, firmware and software development, and other business tasks. Unfortunately, global supply chains and the fabless business model bring with them a substantial security risk: lack of control over the supply chain. This introduces the potential for hardware tampering at many different stages.

The CPU begins life as a high-level design specification. This design is implemented in code at the register-transfer level (RTL) in a language such as Verilog, before being synthesised into a gate-level netlist and then transformed into GDSII format using EDA (electronic design automation) tools [31]. At each of these stages third-party intellectual property (IP) blocks (such as a secure coprocessor) may be added. The foundry fabricates the GDSII layout into a wafer, which is then cut into individual dies at the assembly facility, with each die being subsequently processed further and encapsulated. The finished CPUs are shipped to distributors and system manufacturers (referred to here as OEMs), before being integrated into systems and sold to consumers. Testing and verification is of course involved at each development stage, along with further implementation such as Design for Test debugging infrastructure (see [31] for further details). However, even in this simplified description of the manufacturing process the complexity of the supply chain is apparent. Vulnerabilities may be introduced not only by engineers and factory workers but also by governmental intelligence agencies in each of the countries through which the CPU transits, or by software bugs in the EDA tools.

A variety of hardware trojan designs have been proposed which are extremely challenging to detect and could be implemented at a third-party fab, such as [32], which modifies the dopant polarity of transistors to compromise a system without adding any additional digital logic. An alternative to IC modification of the CPU itself - as this requires high technical sophistication - is to compromise adjacent system components with privileged access to the CPU. For example, a hardware implant can compromise the baseboard management controller on servers to exfiltrate data processed by the CPU and even to control the CPU [33]. This risk was dramatically highlighted by Bloomberg's recent claim that server manufacturer Supermicro had been targeted by the Chinese government in a hardware supply chain attack. Allegedly, embedded implants smaller than a grain of rice were used to target servers used by the US government and major US companies, including Amazon and Apple [34]. This has been widely denied by the companies involved and the US government [35] and, with evidence not forthcoming from Bloomberg, it remains unclear whether the attack did occur or whether vested interests were at work in the article's publication. However, the story clearly illustrates the wide reach such an attack could have if successful.

Furthermore, nation states are now establishing dedicated cyber military units and investing significant resources into cyberwarfare: we have already seen state-sponsored electoral interference [36] and attacks on critical national infrastructure such as hospitals [37], the power grid [38], and nuclear facilities [39]. In particular, Russia has conducted cyber attacks on numerous countries and has been engaged in hybrid warfare in Ukraine since 2014 [40]. In light of this, state-sponsored cyberwarfare via sophisticated hardware modification now seems plausible. Huang has noted that supply chains are essentially a TOCTOU (time of check, time of use) security problem [41]: the gaps in time and location between when hardware is tested at each stage of manufacturing, and when it is actually used, provide nation states and other malicious actors with ample opportunity to compromise a product during manufacture or transit. (Note that state-sponsored hardware tampering for cyberwarfare is distinct from hardware tampering for mass or targeted surveillance of a state's own citizens, as discussed in the next section, because a cyberwarfare attack specifically targets other nations.)

1.2.2 Backdoors and Surveillance

The 'crypto wars' provide a long history of governments attempting to compromise encryption, in particular via export controls weakening cryptographic protocols and *key escrow* schemes, in which the government holds a master key or set of keys to decrypt all encrypted communications [42]. The risks of master keys, or *backdoors*, are obvious: such privileged access violates privacy norms, is ripe for governmental abuse, and can be maliciously used by a third party if the keys are leaked. Removing the backdoor after the leak may be impossible if it forms an integral component of the cryptosystem or is implemented in hardware. The leaks of the TSA's master keys are an excellent example. The US Transportation Security Adminstration (TSA) requires access to air passengers' luggage for inspection: if passengers wish to lock their luggage, they must use approved locks which are designed to be opened by a set of master keys, or risk their luggage being opened by force. Yet thanks to a photo of the keys mistakenly published by the Washington Post, the master keys

have been reverse-engineered and can now be cheaply ordered online or even, for those with access to a 3D printer, printed at home [43].

The Wikileaks disclosures of 2013 established that government intelligence agencies in the USA, UK, Canada, Australia, and New Zealand are complicit in mass surveillance of global digital communications [44]. This mass surveillance included the insertion of backdoors into commercial products: a budget leaked to the New York Times detailed that the US National Security Agency (NSA) requested $250 million in 2013 for the SIGINT Enabling Project, which "*actively engages the US and foreign IT industries to covertly influence and/or overtly leverage their commercial products' designs...[to] make the systems in question exploitable*" [45]. Whilst holiday luggage does not typically contain highly valuable or highly sensitive items, digital systems contain a wealth of sensitive data and so a leaked digital backdoor would have a far more significant impact. The leak of EternalBlue from the NSA[5], for example, had a significant global impact: its use in the WannaCry malware alone is estimated to have caused over US $8 billion of damage [46]. Given the complex global supply chains of most digital products, it is unlikely that a digital backdoor could be contained within one nation's jurisdiction, making the legality of such governmental mechanisms highly questionable [42].

Unfortunately, lawmakers continue to overlook these concerns. The recent Assistance and Access (AA) Bill in Australia is a case in point, appearing to ask the impossible: whilst it includes *"an explicit prohibition against...a systemic weakness or vulnerability"*, it requires providers to 'selectively deploy' such weaknesses or vulnerabilities if asked [47]. A selectively deployable backdoor is still a backdoor: as a provider cannot know in advance which users they may be asked to deploy the backdoor against, they must necessarily implement a mechanism for deployment of the backdoor *for all users of their product*. In the case of hardware, the provider will not (in the vast majority of cases) have physical access to the hardware after its sale, and so must implement a *remote deployment* mechanism. Such mechanisms could be maliciously leaked or reverse-engineered, and then - in the case of a remote mechanism - could be potentially used by a malicious third party to compromise all users of the product globally. Essentially, the AA Bill constitutes a supply-chain compromise for all products developed or maintained in Australia.

Given these precedents, it is therefore plausible that Intel, AMD and other CPU manufacturers may have been legally compelled to insert backdoors into their products and/or their products' cryptographic mechanisms (such as the `rdrand` instruction and underlying random number generator). This further motivates the need to research undocumented CPU behaviour. Such a backdoor would almost certainly be undocumented, as typically such legal compulsion includes a requirement for strict secrecy regarding the nature of the request. Warrant canaries are the only known legal mechanism for a company to disclose that it has been subject to such a request [48], and neither Intel nor AMD maintain one, although Intel state that their "product development policy and practices prohibit any intentional steps to allow undocumented device access" [49].

1.2.3 IP, Licensing, and RISC-V

As introduced in Section 1.2.1, many CPU manufacturers are now fabless, and a common business model is to produce designs of CPUs and subcomponent IP blocks which are then licensed to third-parties. Arm, the dominant manufacturer in the mobile and embedded markets [50], is a prominent example of a 'manufacturer' whose business model primarily revolves around licensing and royalties for IP. They issue very few architectural licenses, i.e. licenses to design new microarchitectures for the Arm ISA, and so a company entering the market would need to purchase a license for one of Arm's existing CPU designs. The costs involved are substantial: an 'entry-level' single-use license for a Cortex-A CPU costs approximately $1 million upfront plus 2% royalties from every CPU sold, whilst the upfront costs for other licenses can exceed $10 million [51]. Obtaining a license for the x86 architecture, meanwhile, appears to now be close to impossible: Intel and AMD have a history of trying to lock competitors (including each other) out of the x86 market via a stranglehold of patents, cross-licensing agreements and legal proceedings, which has led to their current monopoly over the ISA[6] [52].

ISA licenses therefore present a significant legal and financial barrier to entry to CPU design. An open-source, freely-licensed and widely-supported ISA could open up the market to newcomers, potentially reducing the industry's monopolization, spurring microarchitectural innovation, and increasing the economic incentive

[5]Note that it is believed EternalBlue was a stockpiled vulnerability rather than a deliberately implemented backdoor.

[6]This is excepting Centaur Technology/Via Technologies; they are the only other current license holders and manufacturers of x86 processors. Their x86 market share outside China appears to be negligible, but no recent statistics are available.

Application
Algorithm
Programming Language
Operating System | Drivers
Instruction Set Architecture (ISA)
Microarchitecture
Register-Transfer Level (RTL)
Physical Implementation

Figure 1.3: Software (grey) and hardware (white) abstraction layers.

for security by improving profit margins. Along with the potential for new microarchitectural security features, the resulting design heterogeneity might be beneficial to security in itself: considering security from the perspective of biological immune systems as suggested by [53], microarchitectural diversity could help prevent future cross-manufacturer CPU 'superbugs' such as Spectre and Meltdown. An open-source ISA could start afresh, learning from the mistakes of older architectures such as x86 to improve efficiency without sacrificing security. The ISA's openness would make analysing its security far more accessible, enabling any ISA-level vulnerabilities to be more rapidly discovered and fixed. Could this dream ISA be the solution to the CPU industry's current security crisis?

Proponents of RISC-V, an open-source ISA developed by the RISC-V Foundation, argue that it could play this role. It is not the only open-source ISA, with others including OpenRISC and the recently open-sourced MIPS, but so far it appears to have the most momentum. In particular, Western Digital have developed and open-sourced the RTL for a RISC-V CPU (SweRV) [54], and plan to gradually transition their entire product stack to using RISC-V CPUs, anticipating shipping around 2 billion annually once completed [55]. Nvidia chose RISC-V for the next generation of their Falcon microcontroller used in all their GPUs, specifically citing the cost of an Arm out-of-order CPU as motivation [56], whilst other RISC-V Foundation members include Google, Qualcomm, Samsung, NXP Semiconductors, Rambus, and Thales. Widespread adoption is crucial to an ISA's success, as companies are unlikely to choose an ISA without strong toolchain support or with insufficient support for their customers' applications. RISC-V's substantial industry support - far exceeding that of other open-source ISAs - therefore suggests that it has potential to become a major competitor to Arm (particularly for embedded and edge-computing devices) and in the longer-term also to x86.

Whilst many companies adopting RISC-V are still creating proprietary microarchitectures to manufacture or license, others such as Western Digital and SiFive have already open-sourced designs, and the Linux Foundation's CHIPS Alliance aims to support the development and distribution of open-source microarchitectural designs [57]. Combined with open-source firmware such as Coreboot [58] and the already well-established open-source software ecosystem, there is now potential for entirely open-source and fully auditable systems, with open-source designs or code at every abstraction layer. From the perspective of reducing undocumented behaviour to improve security, this is extremely promising. However, although proprietary RISC-V security IP has been produced such as Dover Microsystems' Coreguard [59] and Rambus' CryptoManager Root of Trust [60], very little public RISC-V security research has been conducted, and in particular not on the limited range of hardware platforms currently available to consumers. A significant concern regarding open-source designs is that open-source verification tooling is very limited in comparison to industry-standard tools [61]. It therefore seems timely to investigate whether undocumented CPU behaviour exists on current RISC-V hardware platforms.

1.2.4 Leaky Abstractions

Hardware-enforced security isolation. In order to cope with the complexity of modern computer systems, both hardware and software developers work with the physical implementation at varying levels of detail, known as the hardware-software abstraction layers (depicted in Table 1.3). This has benefits for both productivity and security (not to mention developers' sanity): requiring a website developer to consider the effects their code might have on the quantum tunnelling occurring in each user's 7nm CPU transistors, for example, would

be utterly absurd, whilst abstractions such as high-level languages protect developers from making memory-management mistakes which compromise security. Typical security threat models depend on enforcement of isolation between abstraction layers. We assume, for example, that the privileged device drivers will not try to attack the operating system, and that the unprivileged user cannot access privileged operating system memory, with this separation of privileged and unprivileged software layers enforced in hardware by the CPU's own privilege levels (see Section 2.2). The rise of virtualisation added the concept of a 'guest' level with even further reduced privileges. In a cloud computing setting with multiple customers processing sensitive data in virtualised environments on the same hardware, it is crucial that this separation of privileges (and access to data) is enforced.

Substantial research has been conducted into software vulnerabilities which compromise this separation (enabling privilege escalation) through misuse of hardware enforcement. However, the hardware's implementation of this privilege enforcement, and indeed the entirety of the hardware abstraction layers, has long been assumed to be trustworthy. [24] suggests a range of reasons for this, notably that proprietary designs render hardware a "black box", encouraging us to ignore its implementation; and that assessing hardware requires a different skillset to software, with the two skillsets rarely being found together. This, perhaps, is the heart of the problem: the CPU complexity explosion enabled parallel complexity explosions at every software and hardware layer, so that each layer requires different skillsets. With no developer now able to comprehensively understand the system, they must therefore blindly trust the security guarantees layers beyond their own claim to provide.

A major weakness in hardware-enforced security isolation is the gap between the ISA and the microarchitecture. As previously discussed in Section 1.1, the ISA guarantees portability of software across different microarchitectural hardware implementations. However, this microarchitectural agnosticism introduces weaknesses which render insecure any setting in which untrusted code shares hardware with trusted code. This occurs on every modern computer. It is common to run multiple software applications together, and each website you visit also runs untrusted code; even with only a single application running, it shares hardware with the privileged operating system and drivers (which - we hope - are trusted code). And of course, this hardware sharing is drastically pronounced in a cloud computing setting. Such hardware sharing introduces resource contention which produces exploitable *side channels*.

Side channels and transient execution. A side channel is any aspect of a system which unintentionally leaks information [62]. Polygraphs attempt to exploit side channels in humans: they monitor changes in physiological indicators such as cardiovascular activity and respiratory activity which may indicate a person's deceptiveness, although their accuracy is disputed [63]. Similarly, computer systems have side channels such as power consumption, acoustic emissions, electromagnetic emissions, and - most crucially - timing. Early side-channel attacks were limited by the requirement for physical access to a device, for example to measure a smart card's power consumption to infer its cryptographic key [62]. Timing attacks, however, can be conducted remotely provided the ability to run untrusted code on a device. Some timing attacks are enabled by insecure software implementations with secret-dependent timing, such as a password checking mechanism which checks user input character by character and immediately aborts the check if a character is wrong.

Such software vulnerabilities can easily be patched once detected. However, there is now a growing body of research targeting secret-dependent timing in hardware due to microarchitectural implementation details, and this is far more challenging and costly to patch; in some cases, it is impossible, and only complete replacement of the hardware will suffice. By ignoring microarchitectural side channels to provide a portable timing-independent (and power-independent, etc.) specification, ISAs fail to provide adequate security guarantees, and this is the basis for recent transient execution vulnerabilities such as Meltdown and Spectre. Microarchitectural features so far found to be exploitable include the cache (see Section 2.2.3) and out-of-order execution [20], branch prediction and speculative execution [21], execution ports [64] and simultaneous multithreading [65] [66], interrupts [67], the floating-point unit [68], and the memory order buffer [69]. [1] provides a systematic overview of many such attacks and proposed mitigations, although new attacks continue to be published regularly. Section 2.2 provides technical details regarding the most relevant of these microarchitectural features.

Hardware roots of trust. Since the Roman era - and likely long before - designers of systems of all kinds, from computing to politics, have grappled with the question of *Quis custodiet ipsos custodes?* (or, approximately, "Who will guard the guards?") [70]. How should powers be separated and safeguarded to minimise their exploitation? This is highly pertinent to the debate concerning government surveillance (see Section

1.2.2) but also to the concept of ultra-privileged 'roots of trust' in hardware. Safeguarding of powers is built into many corporate IT infrastructures at the software level, for example via access controls and auditing which ensure even system administrators cannot exploit their privileged access [71]. At the hardware level, however, such safeguarding appears to have been completely abandoned, and various ultra-privileged security coprocessors have been implemented in an attempt to provide a trusted execution environment for sensitive data processing and to protect the integrity of the boot process and firmware (alongside other motives, such as implementing digital rights management for audio and video).

Most concerning of these are the Intel Management Engine (ME) and AMD Secure Processor. They are the most privileged components of the CPU - Intel ME, for example, has direct access to most of memory [72] - and yet are frequently found to be vulnerable to remote code execution attacks, for the most part enabled by buffer overflows, stack overflows, and other insufficient input validation (see [73] [74] for the Intel ME and [75] [76] for the AMD PSP). Intel AMT was even found to accept an *empty string* in place of the administrator password, permitting an attacker remote control of the system via the web interface [77]. Such attacks illustrate manufacturers' lack of concern for security in these ultra-privileged coprocessors. Bounds checking coding errors are well known and numerous static analysis tools exist to detect them [78], but it would appear that Intel and AMD were unable to check for these *in the most security-critical section of the CPU*. Similarly, the AMD PSP does not implement common software and hardware stack/buffer overflow mitigation techniques such as stack cookies, the NX flag, or address space layout randomisation (ASLR) [75]; the lack of these techniques means that if an an overflow vulnerability is present, exploiting it to run arbitrary code is trivial.

Both the Secure Processor and ME are opaque and essentially unauditable, although there have been promising reverse-engineering attempts to disable the ME [79]. Rutkowska highlights that this makes them an ideal location for backdoors and malware, and that the trend of moving as many sensitive tasks as possible to opaque coprocessors or enclaves is highly concerning for system openness and auditability in general [80]. Hardware roots of trust present the ultimate target for an attacker: once compromised, they could be used to persist malware across even operating system reinstallation and reflashing of firmware.

1.3 Undocumented Instruction Behaviour

1.3.1 Undocumented Instructions

In early microprocessors of the 1970s and '80s such as the Zilog Z-80, Intel 8086 and Motorola 6800, undocumented instructions were common. Some were artefacts of the instruction decoder design: if the decoder did not check every single bit (to minimize logic circuitry), an undocumented encoding similar enough to a documented one would activate the same decoding logic, functioning as an alias for that instruction or perhaps encoding an undocumented operand type [81]. Other undocumented instructions were deliberately implemented, either for debugging purposes or in failed attempts to add new instructions: early fabrication processes were unreliable, so to increase chip yield if the logic for a specific instruction were not fabricated correctly it might simply be left undocumented on the die (rather than repeating the entire fabrication run) [82].

On the x86 architecture, the invalid opcode exception (#UD) was introduced with the 80186: the CPU would no longer attempt to execute *every* instruction encoding presented to it. However, many undocumented instructions still ran without an exception, including `aad`, `aam`, `icebp`, `salc`, and `loadall` [83]. The first four are now at least partially documented (although in the case of `icebp`, it took Intel 30 years to document, despite the fact the instruction triggers an entirely independent class of interrupt), whilst `loadall` is no longer supported (its two encodings are now used for `syscall` and `sysret`). The latter was a particularly interesting case: its purpose was for testing and debugging via in circuit emulation, and it enabled loading 'all' registers (including some normally inaccessible microarchitectural registers) with custom values, thereby permitting access to the entire memory address space. Although a privileged instruction, it enabled a level of control over the system typically not possible even with ring 0 privileges. Documentation for the instruction was only available from Intel under a non-disclosure agreement (NDA); although its existence on the 80286 was widely known, Intel denied its existence on the 80386 [84].

1.3.2 Undocumented and Poorly Documented Behaviour

Errata. No manufacturing process is perfect, and as discussed previously verification and test tooling has not kept pace with increasing design complexity. Each CPU model therefore exhibits some physical defects, and

those known to the manufacturer are published in errata datasheets. These are now available publicly online, although the datasheets typically only contain brief descriptions, with the detail necessary to immediately reproduce an erratum sometimes only available under NDA. For many years, it was standard practice for all errata to be confidential and under NDA; there was concern that users might refuse to buy affected systems or demand upgrades if they were aware of the CPU's flaws [85]. Many errata result in undocumented instruction behaviour, and two infamous cases of this affected certain steppings of the Intel Pentium. The `fdiv` bug caused floating point division to return incorrect results, prompting a recall of affected CPUs at the cost of $450 million to Intel [86], whilst the F00F bug (`f0 0f c7 c8-f`) meant that a single invalid (unprivileged) instruction could halt the CPU permanently until it was rebooted. The latter is colloquially referred to as a 'halt-and-catch-fire' instruction, after the `hcf` debug instruction mnemonic on the Motorola 6800 [82]. Errata are often ambiguously described as causing "unpredictable system behavior" under "complex microarchitectural conditions" [87] [88] [89]. Whilst manufacturers understandably wish to prevent malware developers exploiting an erratum (particularly one which causes a machine check or hang), the lack of detail provided also makes it impossible for legitimate developers to deliberately avoid triggering the erratum.

Poor documentation. Some CPU behaviour is not actually undocumented but is so poorly or misleadingly documented that developer confusion leads to software vulnerabilities. The x86 `pop ss/mov ss` vulnerability is a prominent example: "unclear and perhaps even incomplete documentation on the caveats" of the instructions in Intel's manuals led to a vulnerability across Windows, MacOS, Linux, FreeBSD, and various hypervisors whereby an unprivileged user could crash the kernel, access kernel memory, or even run arbitrary code in kernel mode, because a debug exception triggered in user mode could be delayed across a context switch into kernel mode [90]. In other cases, documentation is ambiguous or inconsistent, for example in Intel documentation for the 'undefined reserved opcode' `d6` or `salc`: Intel documentation variously claims both that it never generates an #UD exception (Vol. 3A, Section 6.15) and that it generates it in 64-bit mode [91, Vol. 3B, Section 22.15 and Vol. 3A, Section 6.15]. Attempts at formally specifying the ISA have found many other such cases [92] [93].

Confidential features. Manufacturers are known to implement so-called 'golden screwdriver' features intended for use only by certain customers, such as those purchasing a software-based CPU upgrade (as offered by the Intel Upgrade Service from 2010-2011) or a specific high-volume customer who pays for custom features to be incorporated into mainstream CPUs (rather than the more expensive option of creating separate custom CPUs) [94]. Manufacturers have also been known to restrict documentation of new features under NDA in an attempt to maintain their competitive advantage, as in the case of Appendix H in the Pentium manuals [95]. Similarly, undocumented debug and test mechanisms are common. Their 'security' is sometimes dependent on certain implementation details remaining confidential, such as register password values (e.g. 0x9C5A203A on AMD) [96].

Differing ISA interpretations. x86 is typically considered a single coherent architecture across Intel and AMD CPUs, and it would seem a reasonable assumption on the part of a developer that if their software is implemented in accordance with AMD's x86 documentation then it will also function correctly on Intel CPUs. Ensuring software portability across microarchitectures is, after all, the very purpose of an ISA. However, in practice the two manufacturers define certain details of the ISA very differently, and often in unclear or obscure sections of documentation. There are several precedents of these architectural differences leading to software vulnerabilities, including the `bsf` instruction enabling sandbox escape in Google's Native Client [97] and the infamous `sysret` privilege escalation [98].

Security concerns. HCF instructions are a serious concern for a system's availability: if, for example, unprivileged code could halt a server CPU hosting cloud infrastructure services then an attacker could easily conduct a denial of service (DoS) attack. This is likely to result in data loss for other current users of the system (for example if there are pending writes to disk at the time of the hang). Similarly, defects such as `fdiv` are a concern for data integrity. Many other types of undocumented instruction behaviour pose a threat to a system's confidentiality. An undocumented instruction might create a microarchitectural side-channel; this is already known to be the case with documented instructions such as `clflush` and `prefetch` [99]. Undocumented debug mechanisms or backdoors might enable total compromise of the system; whilst this may initially sound far-fetched, there are alarming precedents such as a debug instruction on Via x86 processors (unprivileged by default on certain models) which unlocks an alternate instruction set providing complete privilege escalation and control over the system [100].

```
10000010100100001000001010010011       addi t0,ra,-2007
10000010100100001000001010010010       illegal; c.mv t0, tp
```

Listing 1.1: Modifying a single bit transforms the 32-bit ADDI instruction into two compressed instructions, one undocumented.

Case study. Listing 1.1 shows two possible encodings of 32 bits in the RISC-V ISA. Note that, as in the RISC-V specification, bits are numbered right to left, from 0 to 31. The first is a 32-bit `addi` instruction, which adds the immediate value -2007 to the return address register `ra`. The value 11 in bits 1:0 encodes the instruction's length of 32 bits. However, if we change just one of these instruction length bits so that bits 1:0 are now 10, it will instead be interpreted as two 16-bit instructions from the Compressed extension: bits 15:0 encode a `c.mv` instruction, and bits 31:16 encode a *reserved* instruction. This instruction hasn't been defined in the instruction set specification yet, so if we try to execute it on a RISC-V platform such as the SiFive HiFive Unleashed then the CPU should produce an illegal instruction exception.

Except it doesn't: it executes with no exception at all! What just happened? Did it do anything? When we compare the CPU's register values before and after we see no change. Did it change some other aspect of architectural state - perhaps it stored a value into memory, or simply moved the value of one register into the same register? Or did it cause no architectural state change, but rather a microarchitectural state change (such as loading a value into the cache) which could be exploited in an attack? Given that bits in memory can be flipped by a remote unprivileged attacker using the Rowhammer attack [101] [102], it is plausible that an attacker could maliciously modify our `addi` instruction in this way. We have no idea what this instruction does, and our lack of tools to answer this question is the motivation for this project.

1.4 Prior Work

CPU security. As discussed in Section 2.2, there has been considerable research into transient execution attacks since the disclosure of Meltdown and Spectre [20] [21] in January 2018. However, despite this volume of transient execution research other CPU security topics remain severely under-researched, with undocumented instruction behaviour representing a particular gap in the literature with very little existing work.

Fuzzing. Fuzzing is a testing technique involving automatically generating large quantities of malformed or unexpected inputs (either systematically or randomly) and observing their effects on a system. The aim is to find inputs which transition the system into a weird state, indicating a vulnerability which can subsequently be fixed (if using fuzzing for verification or penetration testing) or exploited (as such tools are also used maliciously). Whilst fuzzing's concept is unsophisticated, it has proved effective for bug and vulnerability detection and has been adopted into the secure development practices of major technology companies such as Google and Microsoft [103]. However, the vast majority of fuzzing tools target software rather than hardware, and the research literature reflects this: two recent surveys, for example, define fuzzing exclusively in the context of testing software [103] [104]. In this project fuzzing is applied instead to CPU instructions.

Undocumented instruction fuzzing. Some CPU manufacturers use instruction fuzzing (also known as Random Instruction Stream/Sequence (RIS) generation) as part of their verification process, but the process is not documented in detail [105]. There has been very limited public research into instruction fuzzing or undocumented instruction behaviour. Two direct predecessors to this project are the 1995 paper by Sibert et al., the first in-depth security analysis of x86, which highlighted the need for CPU auditing tools [24], and Domas' open-source Sandsifter tool for x86 CPU fuzzing, which contributed the tunneling algorithm for searching the instruction space (see Section 2.3.1) and found numerous security concerns due to undocumented instruction behaviour [5]. Other related research includes Domas' methodology for fuzzing undocumented model-specific registers via a timing attack [106] and the 'CPU Security Benchmark' tool of Zhu et al [107]. The latter claims to be the first comprehensive CPU security tool, detecting undocumented instructions, compromise of control flow integrity, memory errors, cache side-channels, and transient execution vulnerabilities. If made available publicly, such a tool would be a tremendous contribution to the field; however, it has not yet been released and the authors' description of their undocumented instruction detection raises some concerns about its feasibility (such as the confusion of the terms CISC and RISC with specific ISAs, and their claim to achieve full coverage of the instruction search space).

Given this prior work, why develop another fuzzing tool? Firstly, whilst Sibert et al. highlighted the need for further penetration testing of CPUs and stated that a penetration testing suite would be produced in subsequent research, there appears to be no public version of their tool. As the research project was conducted as part of the US National Security Agency's Trusted Product Evaluation Program, it may perhaps have been developed and retained for internal agency use only. The Sandsifter project produced a valuable initial tool for the field, but it suffers from numerous flaws. It finds many false positives, as the Capstone disassembler it uses does not recognise all documented x86 instructions; it does not infer the functionality of an undocumented instruction, or provide any data for manually determining this (e.g. register state changes); and it assumes that #UD faulting instructions must be permanently invalid (see Section 4.2.2).

The strongest motivation however, is that despite this initial research there is little ongoing public work in the area. As discussed, the instruction space on x86 is vast, and therefore many research questions remain unanswered without improved tooling to support their investigation. Sandsifter is not currently publicly maintained; Domas has reportedly continued developing Sandsifter and investigating undocumented instructions, but has not made this research publicly available, and was unavailable for comment during this project. A "major update" to the tool and full details of an x86 HCF instruction were scheduled to be presented at Shakacon 2018 [108], but no details of the presentation have been released. One promising fork of the tool is Baresifter, which aims to produce a bootable instruction fuzzer [109]. Baresifter's approach is particularly interesting because running without an operating system enables testing in CPU modes other than long 64-bit mode (for 64-bit CPUs with a 64-bit OS) or legacy 32-bit mode (for 32-bit CPUs). The disadvantages however are that a bootable tool is less portable across architectures, and that undocumented behaviour has greater potential to damage the system (e.g. by overwriting files) without the memory protection offered by an operating system. Unfortunately the tool is still under development and too incomplete to test for this project.

OpcodeTester. This relative lack of research and development motivated a research project I conducted into undocumented instruction behaviour in 2018, which identified the weaknesses of Sandsifter for the first time and produced the open-source OpcodeTester tool [2]. I identified several areas for further work, including improving the stability of the tool, developing improved search strategies for the instruction space, investigating the instruction decoder (particularly with regard to a bug identified in kernel mode, see Section 4.4.1), and testing for potential transient execution of #UD faulting instructions. This project aims to research these areas and extend the tool correspondingly to facilitate further research into undocumented instruction behaviour. A severe limitation of the project was that it was conducted on a single 64-bit x86 microarchitecture. With this project, I will ensure the tool is portable across Intel microarchitectures (in particular testing on a 32-bit x86 microarchitecture for the first time) and compare the results on each.

RISC-V. Excepting [107], all existing tools (including OpcodeTester) support only the x86 architecture, and there is no publicly-available research concerning undocumented behaviour on RISC-V. Given RISC-V's growing popularity and the known deficiencies of current open-source verification tooling [61], it seems timely to also port OpcodeTester to this architecture to provide a tool for auditing of RISC-V CPU behaviour, and to investigate undocumented behaviour on existing RISC-V CPUs.

1.5 Project Aims

The objective of this project is to further develop my OpcodeTester tool - extending it and porting it to the RISC-V architecture - and to use it to conduct experiments investigating undocumented instructions and undocumented instruction behaviour from a security perspective. Undocumented behaviour is known to introduce security vulnerabilities, and there is a clear gap in the literature: the topic is seriously under-researched on x86 and this is the first project to investigate it on RISC-V. My experimental hypotheses are motivated by unanswered questions from my previous research project. By releasing the tool as open-source I hope to make security auditing of CPUs more accessible and to encourage further research in this area. More concretely, the project's aims are to:

1. Research and survey literature on CPU security and undocumented CPU behaviour, and provide an accessible introduction to CPU security.
2. Produce an undocumented instruction testing tool cross-compatible with the x86 and RISC-V architectures, and publicly release it as open-source.

3. Use the tool to conduct experiments into undocumented instruction behaviour on the microarchitectures under test (Intel Bonnell and Broadwell for x86, and SiFive Freedom E310 and U540 for RISC-V):
    - Hypothesis 1: undocumented instructions exist on the microarchitectures under test.
    - Hypothesis 2: undocumented exception and/or instruction decoding behaviour occur on the microarchitectures under test.
    - Hypothesis 3: #UD-faulting undocumented instructions are transiently executed and leave microarchitectural traces after rollback on Intel Broadwell (the only microarchitecture under test supporting transient execution).
4. Assess the scope for further research into undocumented instruction behaviour and for further development of tools for such research, and the outlook for CPU security in general.

Chapter 2

Technical Background

> "There are some undocumented internal-use MSRs used for low-level hardware testing purposes. Attempts to read or write these undocumented MSRs cause unpredictable and disastrous results..."
>
> VIA Technologies [110]

Security and computer architecture are both complex fields and a comprehensive introduction to either would be far beyond the scope of this chapter. Instead we provide here a brief introduction to security and architectural concepts crucial to understanding of the project, before describing the instruction set architectures and microarchitectures under test. For further reading on computer architecture, see [7]; similarly for security applied both within and beyond the context of computing, see [111].

2.1 Security Concepts

> *"Many well-meaning persons suppose that the discussion respecting the means for baffling the supposed safety of locks offers a premium for dishonesty, by showing others how to be dishonest. This is a fallacy. Rogues are very keen in their profession, and know already much more than we can teach them respecting their several kinds of roguery...If a lock...is not so inviolable as it has hitherto been deemed to be, surely it is to the interest of honest persons to know this fact"* - A. C. Hobbs, 1868 [112]

Offensive security. This project approaches security from an offensive perspective, with the premise that vulnerability research is an essential complement to the defensive perspective of designing security mechanisms. As the quote above highlights, debate has long raged concerning vulnerability research and its potential to compromise so-called 'security by obscurity', whereby systems rely on secret mechanisms or password values whose security depends on their confidentiality. Modern critics have argued against vulnerability research on both economic and ethical grounds [113]. However, like Hobbs I argue that security by obscurity offers no security at all. As in the case of the TSA locks in Section 1.2.2, their secrets are likely to eventually be reverse-engineered by or leaked to the 'rogues', and it is 'in the interest of honest persons' to discover that such insecure mechanisms exist, ideally before the rogues do - although they probably already know! Offensive security practices such as penetration testing, red teaming, and bug bounty programs have been adopted in major organisations both within and beyond the technology industry, and responsible (or coordinated) disclosure practices which significantly reduce the risk of harm from public disclosure of security vulnerabilities are now the norm [114]. This project was conducted on this ethical basis, with a commitment from the outset to responsibly disclose any security vulnerabilities discovered.

The weird machine. The computational system a hardware or software developer intends to implement can be described as a finite state machine, with a set of *sane* states and a set of *sane* transitions between these states, i.e. the states and transitions which the developer anticipates. This intended finite state machine (IFSM) is the bug-free, idealised version of the system. The IFSM is then emulated by the hardware or

software implementation; this may be a perfect emulation, or - as is far more likely - it may have bugs. A bug or vulnerability is anything which puts the system into a *weird* state, i.e. a state which has no equivalent in the IFSM. This may be due to human error in the implementation, a hardware fault, or a 'transcription error' when communicating the implementation (e.g. a manufacturing error when attempting to reproduce the provided design). The weird machine is the new computational system produced by applying the emulated transitions of the IFSM to one or more weird states, i.e. the results of the system continuing as normal despite being in a weird state. Weird states and transitions may enable an attacker to compromise the system's security properties, for example enabling them to bypass a password mechanism to access confidential data, and so from a security perspective the aim of fuzzing is to identify if any of these weird states and transitions exist. This theoretical framework (formalised by Dullien in [16]) highlights one of the most fundamental challenges in computer security: containing *latent computational power*. Even a seemingly minor vulnerability can lead to a weird machine which provides an attacker with powerful capabilities.

Vulnerabilities, exploits, and attacks. This project uses Dullien's definitions of vulnerabilities, exploits, and attacks in the context of the weird machine [115]. As discussed, a bug or vulnerability is a method for transitioning the system into a weird state; an exploit is a program or other sequence of actions which uses a vulnerability/bug to produce a weird machine and then uses this machine to violate one or more security properties of the system; and an attack is an event where an exploit is used maliciously. A vulnerability/bug is not necessarily exploitable, but given the latent computational power of a typical system it is highly likely that it is, even if this potential is not apparent when it is initially discovered. The attack surface is the sum of all known, unknown, and potential vulnerabilities across the entire system, minus those which are (reliably) protected against by security mechanisms [31].

Trustworthiness principles. [116] provides a good overview of other overarching security and trustworthiness principles. Principles such as least privilege, separation of privilege, and minimisation of what has to be trustworthy (i.e. the trusted computing base, or TCB) directly informed this work. In particular, the concept of *predictably-composable principled architectures* is highly relevant: abstraction is necessary for the design of complex systems, and can improve security when designed correctly, but the current leaky abstraction layers must be replaced with abstractions which *can* be securely composed together without requiring in-depth knowledge of lower layers.

2.2 Architectural Concepts

2.2.1 Exceptions and Interrupts

Exceptions and interrupts are events which the CPU must respond to. Interrupts are asynchronous signals typically from hardware external to the CPU, whilst exceptions occur when the CPU detects an error condition, such as attempting to run an illegal instruction or access an invalid memory address. The operating system must set up exception and interrupt handlers providing code for the CPU to run when such events occur. If a user program causes an exception, the operating system sends a signal to the program; the default behaviour for most signals is to kill the user program. However, a program can also register a signal handler to resolve the error condition by itself and continue running, which is necessary when testing undocumented instructions as exceptions are a constant occurrence. The vast majority of undocumented instructions fault with an illegal instruction exception (#UD on x86 and exception 2 on the HiFive Unleashed and HiFive1), whilst others fault with memory-related exceptions, commonly referred to by the legacy term 'segmentation fault' (on x86 memory violations typically produce the #GP exception, whereas the HiFive boards subdivide memory exceptions into different categories with exceptions 0-1 and 4-7). Note that only the HiFive1's exceptions are referred to by number in this work; the HiFive Unleashed runs Linux and so its exceptions are referred to by their POSIX signal codes (4 or `sigill` for an illegal instruction and 11 or `sigsegv` for a segmentation fault). Other less common exceptions include debug exceptions or divide errors.

Exceptions are typically classified as faults, traps, or aborts [91]. A fault is an exception that can usually be corrected; the CPU restores the architectural state to its state before the faulting instruction, and execution resumes at the faulting instruction rather than at the instruction following it. As exceptions such as #UD and #GP are faults, a user program signal handler for these must modify the instruction pointer to prevent the faulting instruction repeatedly being executed and faulting again. A trap such as a debug exception does not restore state and resumes execution after the trapping instruction, whilst an abort is an exception which cannot be recovered from (such as a severe hardware error). Page faults are the one exception type which do

not indicate an error condition: whilst they must be handled by the operating system, they are an essential component of paged memory management rather than an error, and user programs do not have to handle them manually.

2.2.2 Instruction-Level Parallelism

Not every instruction takes the same amount of time to run. Ideally, every instruction would complete in just a single cycle, but there are inevitably bottlenecks which slow down certain instructions. A load instruction must wait for memory to return the requested value, and memory is far slower than the CPU, whilst complex arithmetic operations such as division can require many cycles (the E51 core in the HiFive Unleashed, for example, has a latency of up to 65 cycles for division [117]). Instruction-level parallelism techniques seek to reduce the number of cycles wasted due to these latencies. Some techniques can be applied by the compiler to optimise code before it is ever executed, but some parallelism is only apparent dynamically at runtime; the instruction-level parallelism techniques discussed here are implemented in the CPU itself to exploit this dynamic parallelism.

Pipelining. Since the mid-1980s, pipelining has been ubiquitous in CPUs to improve instruction throughput [7]. Akin to a factory assembly line, the lifecycle of an instruction is separated out into logically distinct stages so that the CPU can overlap multiple instructions. For example, if we split the lifecycle up into an extremely simple three-stage 'fetch/decode - execute - commit' pipeline we can have up to three instructions in the pipeline at once: provided the stages are implemented by separate functional units, one instruction can be fetched and decoded whilst, simultaneously, another is executed and the results of another are committed to architectural state. This increases instruction throughput. However, hazards in the pipeline pose a major challenge. Structural hazards occur when the CPU does not have sufficient functional units or other resources to overlap certain instructions; this can occur with complex instructions such as floating-point divide which require sub-pipelining of the execution stage, with the relevant functional unit unavailable for multiple cycles until the sub-pipeline has completed. Far more common are data hazards, where an instruction depends on the (not yet committed) result of a previous instruction in the pipeline, and control hazards due to branching. Techniques such as forwarding, i.e. passing a not-yet committed result to an instruction that requires it, can resolve some hazards, but if the hazard cannot immediately be resolved then the pipeline must stall the instructions following the hazard until it is resolved.

Superscalar and out-of-order execution. An obvious next step is to provide more than one functional unit per stage so that multiple instructions can be processed simultaneously at each stage, which is known as superscalar execution. However, if instructions are executed in-order then the performance benefit of this is limited, as having more instructions in the pipeline significantly increases the rate of hazards and entails a greater cost when the pipeline must be flushed. What if the instruction stream could be reordered to reduce hazards? This is the premise of out-of-order execution. Instructions are tracked in a reorder buffer so that they can be executed out-of-order - as and when the operands and resources they need are available - and then committed (or retired) to architectural state in-order from the reorder buffer so that users and software developers (in theory) never notice that computation was reordered. Data hazards can further be reduced with register renaming, which eliminates name dependencies: where two instructions use the same register but no value is transmitted between them (e.g. they both happen to write to register 2), their registers can be renamed so they are using different physical registers and do not interfere with each other. CPUs support this by having a much larger set of physical registers than the set offered to developers in the ISA (the architectural registers), providing sufficient capacity to maintain multiple copies of each architectural register.

Branch prediction and speculative execution. Out-of-order execution helps reduce data and structural hazards but does not reduce control hazards. Branching is extremely common in software and so mitigating control hazards is crucial. The simplest options for handling branches are to either flush the pipeline when a taken branch is detected (as the instructions following the branch should no longer be executed) or to stall until the result (taken or not taken) is known; these both have a high performance cost, however. Equipping the fetch/decode stage with branch prediction helps reduce this performance impact: unconditional branches can be taken immediately to redirect instruction fetching (so no flushing is necessary) and the outcomes of conditional branches are predicted to also redirect fetching. Outcomes can be predicted with various techniques, from simply assuming that branches will always be taken, to maintaining extremely complex histories of previous branches (as in typical software branches are likely to happen repeatedly, such as in loops). The true benefit of branch prediction, however, is realised when the instructions fetched on the basis of predictions are also

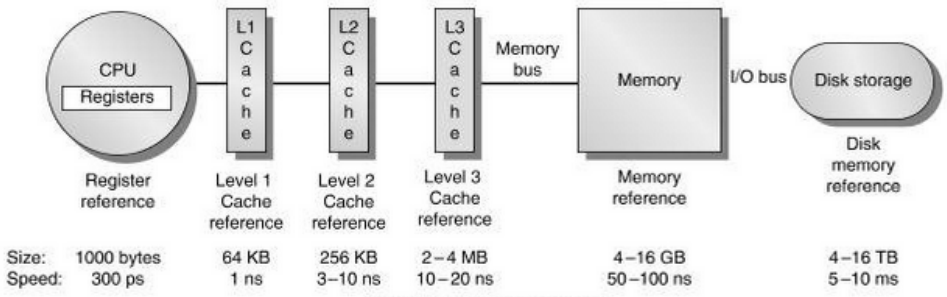


Figure 2.1: A typical desktop/server memory hierarchy. Image reproduced from [7].

executed before the branch outcome is known, which is known as speculative execution. As with out-of-order instructions, speculatively executed instructions are not committed from the reorder buffer until their branch prediction is known to be correct to avoid erroneously modifying architectural state, and are flushed from the buffer in the event of a misprediction. However, microarchitectural changes (such as data loaded into the cache) are not rolled back, and it is the microarchitectural effects of these 'transient' speculative or out-of-order instructions - in combination with precise exception handling, where exceptions (such as a failed memory privilege check) are only generated at commit time so that they occur in-order rather than out-of-order or speculatively - which enable transient execution vulnerabilities.

2.2.3 The Memory Hierarchy

An ideal computer system would have unlimited quantities of fast memory. However, computer architects have been struggling with the reality of the trade-offs necessary since 1946 [7]. Faster memory is both costly and power-intensive, and larger memory is inherently slower to access because addressing entails more complex logic across greater distances. Fortunately, most programs access data and code with *temporal and spatial locality*: the most recently-used data/code is likely to be needed again soon (temporal locality) and data/code close to a recently-used region is likely to also be needed soon (spatial locality). This enables memory to be structured hierarchically. As shown in Table 2.1, a small set of very fast memory (the registers) is located closest to the CPU and used for the most recently-used data. This is followed by larger and slower layers of cache memory, the Random Access Memory (RAM), and finally the exceptionally slow but large hard disk or solid state drive (HDD/SSD).

When a program accesses a memory address, the CPU searches up the hierarchy for its value. In the event of a *cache hit* (the value is already in a cache), the value can be returned significantly faster (approx. 1-20ns, depending on the cache level) than if a *cache miss* occurs and the value must be obtained from RAM (50-100ns). If this occurs, the value and a certain number of neighbouring values (together comprising a *cache line*) are moved together into the cache to exploit temporal and spatial locality. If any given cache is full, a previous cache line must be evicted. Multiple cache lines are commonly grouped into *cache sets* and the number of lines per set is known as the cache's set associativity (or 'wayness', e.g. 4-way). Cache implementations vary, with a variety of trade-offs involved such as addressing overhead and miss rate, but typically a portion of a cache line's starting address is used as an index to determine the line's location in the cache. In a 1-way (direct-mapped) cache, a given cache line will always be placed in the same location, whereas in a fully-associative cache a cache line can be placed anywhere; most caches compromise in between, with a 4-way set associative cache for example mapping a line onto a set and then placing the line at one of four possible positions in the set [7]. As each CPU core may have its own caches and operate on the same addresses as another core, consistency of values must be maintained across cores via a *cache coherency* policy; to facilitate this, the last level cache (L3 in current desktop CPUs) is typically inclusive across cores, containing all the cached data held in the per-core caches.

Unfortunately the timing differences induced by this memory hierarchy create the potential for cache side-channel attacks, which exploit timing of memory accesses and cache operations to determine if a victim process has accessed a certain memory address. Although this leaks access patterns rather than data explicitly, if the victim process makes data-dependent memory accesses then the access pattern reveals the data. In the Flush+Reload attack [118], for example, an attacker evicts a cache line of data from a shared memory page with the `clflush` instruction, waits whilst the victim potentially accesses the address, and then measures the

Ring 3 (unprivileged user mode)
Ring 2 (unused)
Ring 1 (unused)
Ring 0 (privileged kernel/OS mode)
Ring -1 (hypervisor)
Ring -2 (system management mode)
Ring -3 (Intel ME / AMD PSP)

Figure 2.2: Privilege rings on x86 (negative rings unofficial).

User mode
Supervisor mode
Machine mode

Figure 2.3: Privilege levels on RISC-V.

time taken to access the address. If the victim accessed it there will be a cache hit; otherwise there will be a cache miss. The timing difference between these two cases is substantial enough that they can be distinguished even when the attacker's access to architectural timing mechanisms is limited [22].

2.2.4 Privilege Levels

As discussed in Section 1.2.4, security threat models rely on hardware-enforced security isolation between abstraction layers. Such isolation is now known to be vulnerable to cross-layer attacks exploiting hardware side-channels via software. Traditional privilege layer models also violate the principle of least privilege, as the most privileged layers typically exercise absolute power with no safeguards. Table 2.2 and Table 2.3 present the privilege levels used on x86 and RISC-V.

On x86, rings 1 and 2 (originally intended for drivers) are almost exclusively unused, which has given rise to the simplification of 'privileged' (operating system/kernel) and 'unprivileged' (user mode) code. Rings -1 through -3 are unofficial terms used to describe the extra-privileged access permitted to three aspects of the system excluded from the traditional abstraction layers: the hypervisor (for virtualisation), system management mode, and hardware roots of trust (Intel ME / AMD PSP). RISC-V uses the privilege levels of user mode, supervisor mode, and machine mode, approximately corresponding to ring 3, ring 0, and ring -2 respectively [119]. Note that the RISC-V privileged ISA specification is still in draft, and a former hypervisor privilege layer has been removed from the hierarchy.

2.2.5 Instruction Set Architectures

To expand upon their initial description in Section 1.1, beyond specifying the instructions a CPU can execute ISAs also specify how these instructions are encoded and the runtime CPU state accessible by programs, in particular the size and number of architectural registers which are available. ISAs can be classified in numerous ways, but the most crucial distinction is between RISC (Reduced Instruction Set Computer) and CISC (Complex Instruction Set Computer) ISAs. RISC ISAs feature primarily single-cycle, fixed-length simple instructions; this decreases hardware complexity and makes pipelining straightforward. In contrast, CISC ISAs typically feature variable-length instuctions supporting highly complex functionality and a variety of addressing modes (in particular memory-to-memory) [7]. This simplifies compilation and improves code density, thus reducing memory requirements. However, it essentially transfers the compiler's burden into hardware, as this entails increased hardware complexity.

Microcode. A common strategy to counter the complexity of CISC ISAs is to make substantial use of microcode, essentially a second more RISC-like instruction set which the ISA's instructions are translated into, and which is the instruction set actually implemented in hardware. Simple instructions may map directly to a single micro-operation (referred to here by the x86 abbreviation μop), whilst complex instructions may map to

a long program of µops requiring many cycles. On x86 these microcode 'programs' can be updated via non-persistent runtime patches applied by the firmware or operating system. This brings the concept of software patching to hardware and is extremely useful for mitigating hardware bugs quickly and cheaply without any need for hardware replacement. However, it also poses the risk of a malicious microcode update, which might for example modify the microcode of cryptographic instructions to weak encryption. As x86 microcode is opaque (updates are encrypted and undocumented) such an attack would be challenging to conduct but also extremely difficult to detect.

Model-specific registers (MSRs). These are control registers used to modify the CPU's low-level functionality which typically require a privileged instruction (ring 0 on x86, or machine mode on RISC-V) to access. This is to prevent a malicious unprivileged user using them to change the system's behaviour. They are typically intended for test and debug purposes, although they can also play a role in the boot process and firmware update mechanisms, and can disable hardware features if they are found to be flawed after manufacture [120]. Many MSRs are undocumented and may only be disclosed under NDA (for example to OEMs) or remain solely known to the manufacturer. Documented MSRs relevant for this project include the hardware performance counters: typically, a microarchitecture will provide a set of MSRs for selecting microarchitectural parameters to measure (such as the number of instructions executed or retired) and counter registers to read the measured values from. These are intended for performance benchmarking but have also been used for exploit detection [121] and reverse-engineering microarchitectural implementation details [122].

2.2.6 The CPU Pipeline

As Broadwell is a die shrink of Haswell with no substantial microarchitectural changes (see [123]), at this high level of abstraction the two microarchitectures are identical, and so the pipeline details in this section are highly relevant to Section 2.3.1. This section particularly focuses on the front-end decoding process to provide background for the investigation of decoding in Sections 4.2.3 and Section 4.4.2. Many of these details are also more broadly applicable to Intel CPUs in general; the reference for this section, [123], provides details of all recent Intel microarchitectures.

Figure 2.4 illustrates the pipeline stages in the Intel Haswell microarchitecture. Conceptually, the Haswell pipeline is split into three sections: an in-order front-end, an out-of-order engine and an in-order retirement stage. Within the pipeline, instructions are split into µops as described in Section 2.2.5. As discussed in 2.2.2, branch prediction enables speculative execution of instructions before the outcome (or target address) of a branch is known. Modern Intel microarchitectures make extremely aggressive use of speculation. The branch prediction unit (BPU) makes predictions for 32 bytes at a time; this is double the rate of instruction fetching into the pipeline (16 bytes) so there is effectively no penalty for predicting that branches are taken. The exact details of the BPU are proprietary, but it is known to include units for tracking the history of branch outcomes (taken or not taken), branch targets (the branch target buffer, or BTB), and return addresses of function calls (the return stack buffer, or RSB) [21]. The BPU guides instruction fetching, fetching one 16-byte block of instructions into the front-end per cycle from the predicted address. It fetches the block from (in order of preference):

- The decoded instruction cache (decoded icache), which caches µops output from the decoders rather than instructions;
- The instruction cache (icache) in the legacy decode pipeline, which caches full instructions;
- The L2 cache (which caches instructions and data together);
- The last level cache (LLC);
- Or, as a last resort, memory.

The legacy decode pipeline consists of the instruction translation lookaside buffer (ITLB), instruction cache, predecode unit, and decoders. 'Legacy' might suggest that this pipeline is rarely used, but it is in fact always used the first time an instruction is executed. If that instruction is used again before it is evicted from the decoded icache, however, it can skip the decoding process in this pipeline and be fetched immediately from there (this occurs for >80% of µops, according to Intel's estimates). First, the ITLB is used to perform a lookup into the icache to fetch a 16-byte block. This is processed by the predecode unit, which determines instruction lengths and marks instruction boundaries and properties (such as "is a branch") for the decoders.

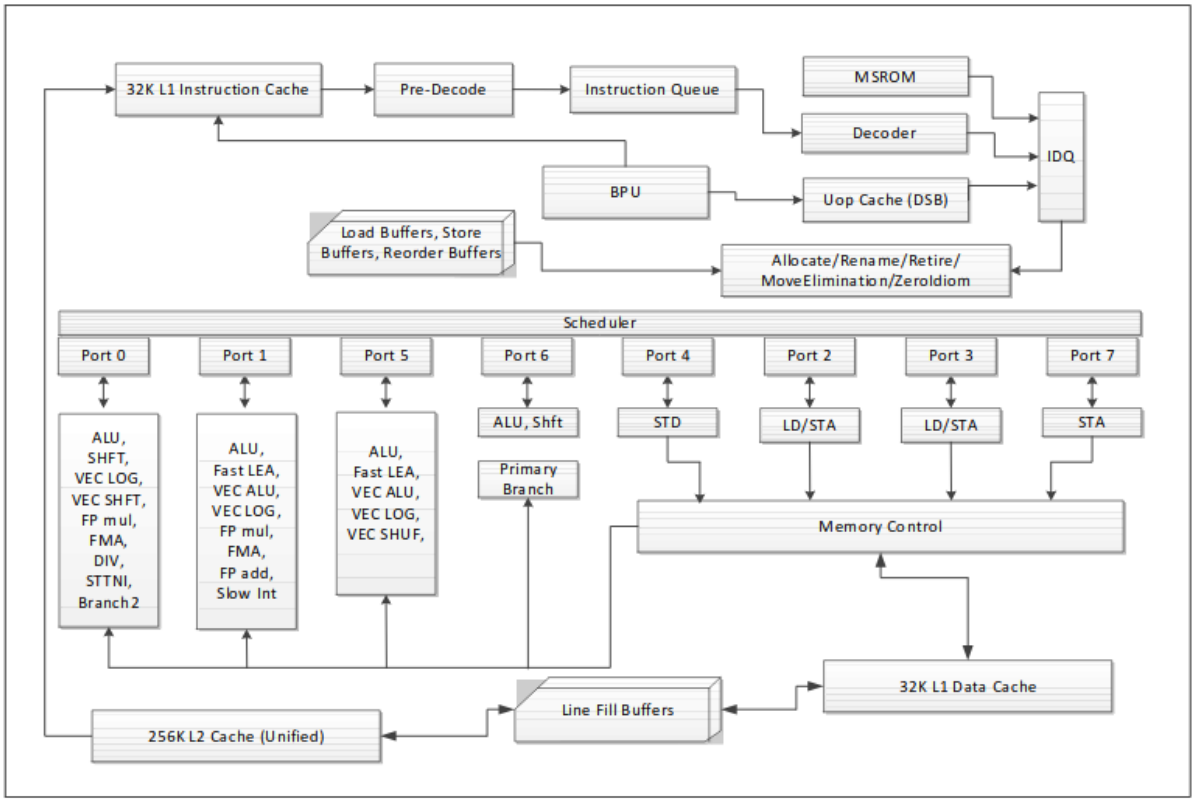


Figure 2.4: A pipeline diagram (depicting execution flow rather than physical layout) of an Intel Haswell microarchitecture core. The Intel Broadwell microarchitecture is extremely similar and at this high level of abstraction is identical. Image reproduced from [123].

It is slowed down by three cycles per length changing prefix (66 and 67), which is important to bear in mind when analysing decoding timings/cycles (as in Section 4.2.3 and Section 4.4.2). The marked instructions are sent (via the instruction queue) to the four decoding units, which translate each instruction into μops. Three of these decoders can only handle single-μop instructions, with the fourth translating instructions of up to four μops. Very complex instructions involving >4 μops come instead from the microcode sequencer. These are the only instructions affected by microcode updates. If a runtime microcode update modifies an instruction's microcode, then the update's μops are fetched; otherwise, the μops permanently encoded into the MSROM (microcode sequencer ROM) are fetched at a rate of 4 μops per cycle. Note that microfusion and macrofusion can also occur during these front-end stages: macrofusion merges two instructions into a single μop, whilst microfusion merges multiple μops from the same instruction into a single complex μop.

After decoding, μops enter the instruction decode queue (IDQ). The IDQ contains a loop stream detector (LSD) which can detect loops up to 56 μops long so each iteration's μops can be issued directly from here; the loop executes without its instructions being fetched/decoded until ended by a branch mispredict. It cannot handle loops with `call` or `ret` instructions, mismatched stack operations, or more than 8 taken branches. From the IDQ, μops enter the out-of-order engine. The renamer moves μops from the IDQ to their relevant dispatch ports in the reservation station (RS, also called the scheduler), from where they are dispatched to an execution port in the execution core. Each μop must be allocated an entry in the reorder buffer (ROB), an entry in the RS, and a load/store buffer if it accesses memory, so if any of these resources are not available then μops are stalled at the IDQ. As its name suggests, the renamer carries out register renaming (as described in Section 2.2.2), but also various optimisation tasks. For example, certain register move operations can be carried out here without entering the execution core, although of course their results are not committed architecturally until retirement. The aim is to rename and reorder μops so that they can execute as efficiently as possible, executing as soon as their their dependencies are met and resources are available (e.g. a relevant execution port is free). Up to 192 μops can be "in-flight" in the out-of-order engine at once.

The RS can dispatch up to 8 μops per cycle (provided their dependencies are met and sufficient resources

are available) as there are 8 execution ports specialised for different tasks. Ports 0, 1, 5, and 6 perform arithmetic and logic operations, whilst ports 4-7 handle loads and stores from/to memory (see Figure 2.4); branches are normally handled on port 6, with port 0 as a backup if resources are not available on port 6. Each port has several 'stacks' to handle different types of data, and moving results between stacks can incur a delay, which is relevant when working with SSE, AVX, or the legacy floating-point x87 instructions.

Once executed, µops enter the retirement stage. They (and their results) wait in an entry in the ROB; the ROB handles µops both in the execution core and in retirement, with the two strictly separate and allocated half the ROB each. A µop does not modify architectural state until it is committed/retired in-order once all previous µops have been committed and (if speculative) it has been confirmed to not have been on a branch mispredicted by the BPU. However, within the pipeline results can be forwarded to µops in flight which require them. The ROB ensures that exceptions occur in-order; other than aborts (which indicate a catastrophic event in the processor), they are only ever generated at retirement.

These details are crucial for investigating undocumented instruction behaviour: in order to identify strange and possibly undocumented behaviour, it is of course necessary to know what the normal documented behaviour should be! The hardware performance counters enable (fairly) accurate measurement of the number of µops passing through each of these units, allowing us to investigate unanswered questions such as whether undocumented instructions might ever execute in the execution core and produce microarchitectural effects; Section 4.4.2 presents experimental results regarding this.

2.2.7 Verification

Verifying a modern CPU is exceptionally challenging and compromises a substantial proportion of total development time. The exact time required varies by manufacturer and by project. Arm, for example, estimate that each CPU design undergoes more than 10,000 hours of verification, with additional time for system-level validation (6,000 hours for the Juno CPU) [105], whilst Intel's first formal verification of an entire execution engine (for the Core i7 CPU) required "twenty person years" of work [124]. Such verification can be broadly divided into *dynamic* verification (testing a simulation of the design, or an emulation using programmable logic such as an FPGA) and *formal* verification (proving the design's correctness with respect to a formal mathematical specification). Dynamic verification is typically easier and faster to conduct, but suffers from limited coverage. The 'complexity explosion' of hardware has led to a corresponding explosion in possible states: for example, as discussed in Section 2.3 it is impossible to exhaustively test every possible x86 instruction encoding, even before one considers the vast array of architectural or microarchitectural states the CPU might be in when it executes a given x86 instruction. Formal verification offers the potential to achieve full coverage without resorting to exhaustive testing in simulation, which has motivated its increasing adoption at Intel [124].

Note that in the context of this project, all references to 'verification' refers to functional verification, i.e. ensuring that an implementation meets its functional specification. Other forms of verification not considered here assess non-functional aspects such as layout or power consumption. Dedicated security verification procedures also exist which verify security properties and information flow models; whilst valuable, functional verification alone is currently sufficiently challenging that I believe the best progress that can be achieved for security in the short-term via verification is to improve functional verification.

2.3 ISAs and Microarchitectures under Test

Instructions versus opcodes In this project, 'instruction' is used to refer to any distinct encoding: two encodings are considered distinct instructions even if they only differ by one operand bit. This contrasts with other definitions which distinguish only functionally distinct encodings as distinct instructions, although there has been considerable debate concerning what defines 'functionally distinct'. However, I believe this definition is the most useful because it is the least ambiguous. By this definition of instruction there are 1,334,440,654,591,915,542,993,625,911,497,130,240 possible x86 instructions[1] (a dire situation indeed for fuzzing!) and 4,295,032,832 possible RISC-V instructions (considering only 32- and 16-bit instructions, i.e. RVGC). [93] provides two alternative definitions: instructions are distinct if their assembly mnemonics differ, or if their combinations of mnemonic and operand type differ (981 and 3,684 'instructions' on x86 respectively).

[1] This is $2^8 + 2^{16} + 2^{24} + 2^{32} + 2^{40} + 2^{48} + 2^{56} + 2^{64} + 2^{72} + 2^{80} + 2^{88} + 2^{96} + 2^{104} + 2^{112} + 2^{120}$; as the meaning of previous bytes may change when additional bytes are added, it is necessary to consider all possible encodings at all possible lengths, rather than merely all possible 15-byte encodings.

However, this approach fails when attempting to count undocumented instructions for which no mnemonics exist. Depending on the consistency of instruction encoding in an ISA, it may be possible to define a set of 'functional' bits, i.e. the only bits which are documented to alter instruction behaviour rather than encoding operands, and define an instruction as any distinct encoding of functional bits. This is possible on RISC-V (see Section 3.5) but appears to be impossible on x86 due to the ambiguity of instruction encoding.

What does it mean to be 'documented'? As discussed in [2], the Intel XED disassembler recognises far more x86 instruction encodings than other disassemblers such as Capstone; most of Sandsifter's results appear to be false positives when checked against XED, and I therefore use XED as the 'golden reference' for which instructions are and are not documented on x86 in this project. However, the idea of an instruction being 'documented' is somewhat ambiguous. There are some x86 instructions which are supported only by one manufacturer or on selected CPU models, and many more instructions which are so poorly documented as to be almost undocumented. For example, the `ffreep` instructions (with the same functionality as the documented `ffree` instructions plus an x87 stack pop) are named by XED but absent from Intel's opcode maps. The hex encoding and its behaviour is briefly mentioned in an obscure 'Architecture Compatibility' section of one manual [91, Section 22.18.9, Vol. 3B], but never referred to by name or defined in the main instruction set reference manual. Can these instructions really be considered documented given they are not in the instruction set reference? This ambiguity is a constant challenge when investigating x86. Michael J. Clark's disassembler is used as the golden reference for RISC-V [4]; fortunately, obscure and partially-documented instructions are far less of a concern on RISC-V, although the continual evolution of the specification does result in certain opcodes being implemented in only certain tools, or being documented only in GitHub commits of draft specifications.

2.3.1 x86

The x86 ISA was introduced in 1978 with the Intel 8086 CPU. Since then, it has been expanded from 16-bit word size to 32-bit and now 64-bit, and a wide variety of instruction extensions have been added such as x87, MMX, 3DNow!, SSE, AVX, and TSX (with some extensions having redundant equivalents produced due to intense competition between Intel and AMD [19]). Throughout this time almost complete backwards compatibility has been maintained, resulting in an extremely complex ISA with many deprecated and seldom-used features, and thousands of pages of documentation. It is impossible to do x86 full justice here; of relevance to this project are its instruction encodings, techniques for searching the instruction space, Intel TSX, and the retpoline construct. [91], [123] and [125] provide further documentation. Note that all x86 encodings in this work are in hexadecimal even where not marked as such with a '0x' prefix.

The original 16-bit ISA had 8 general purpose registers: `ax` (accumulator), `bx` (base address), `cx` (count string/loop), `dx` (multiply/divide), `sp` (stack pointer), `bp` (base pointer), `si` (string source pointer), and `di` (string destination pointer). The first four registers are further subdivided into high and low sections (e.g. `ah` and `al` for `ax`) so they can also be used as 8-bit registers. These registers were then extended into exx versions (e.g. `eax`) by the 32-bit ISA and into rxx versions (e.g. `rax`) by the 64-bit ISA, but even on the 64-bit ISA the smaller register versions remain accessible. Other registers present since the original 16-bit ISA include the segment pointers (`cs`, `ds`, `es`, `fs`, `gs`, `ss`), the instruction pointer (`rip/eip/ip`), the flags register for condition codes (`rflags/eflags/flags`), 8 floating-point registers (`st0`–`st7`) and the floating-point status register (`fpsr`). These are the most common registers; there are also control registers (`cr0`–`cr15`), debug registers (`dr0`–`dr15`), and a vast array of registers for multimedia and SIMD (Single Instruction Multiple Data) processing.

Instruction encoding. x86 instructions are variable-length. Any valid instruction must contain at the minimum a 1-3 byte opcode, which defines the instruction's basic functionality. A `nop` instruction, for example, is simply a single opcode byte, `90`. However, many other opcodes require operands, i.e. displacement and/or immediate bytes, which can each be up to 8 bytes in length. The ModR/M and SIB bytes are optional and essentially extend the opcode with addressing information. The mod field of the ModR/M byte combines with R/M to encode one of 32 possible register or addressing mode values, whilst the reg/opcode field specifies either a register or 3 further bits of opcode information. The value of the reg/opcode field is written as /digit (e.g. `0f 0d /1`, where `0f 0d` is a 2 byte opcode), which is pertinent to Section 3.7. The SIB byte is less common and is used in combination with certain ModR/M encodings for base/scale-plus-index addressing; see [91, Section 2.1.5, Vol. 2A] for full definitions of possible ModR/M and SIB combinations, and note that some instructions encode addressing directly in the opcode instead. There are a wide array of possible

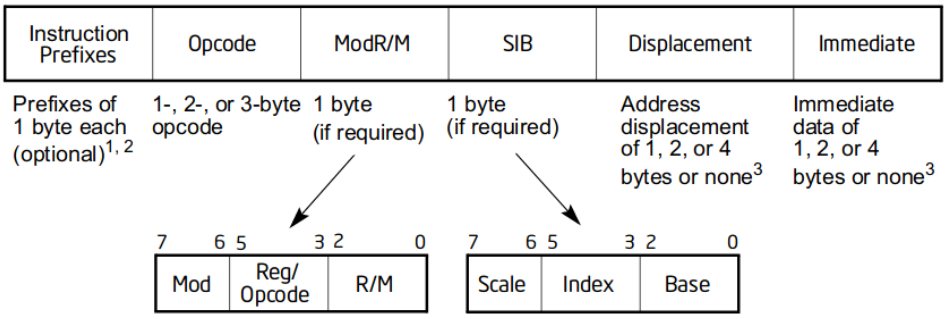


Figure 2.5: The x86 instruction format. Image reproduced from [91].

prefixes: the legacy prefixes 26, 2e, 3e, 36, 64, 65, 66, 67, f0, f2, and f3, which control memory segments, operand and address size, atomicity, and repetition for string operations; the REX prefixes 40 to 4F, which extend the ModR/M and SIB bytes so that the 8 extra general-purpose registers added in the 64-bit ISA can be addressed; the VEX and XOP prefixes C4, C5, and 8E, and the 4-byte EVEX prefix (62 00 00 00). VEX, XOP and EVEX are essentially new instruction formats in themselves and can be used to encode instructions with up to 5 operands.

Maximum length. Whilst Intel and AMD variously claim that only 4-5 legacy prefixes can be used, they are largely irrelevant with most instructions (particularly in 64-bit mode with its limited use of segmentation) and so instructions can be encoded comprising up to 14 legacy prefixes. Based on experimentation, this appears to be a hard limit: any instruction longer than 15 bytes triggers a #GP exception, and an instruction must contain an opcode, so 15 bytes of prefixes trigger this. This matches Intel's and AMD's documentation that the maximum instruction length is 15 bytes. However, astonishingly there is some ambiguity regarding this basic characteristic of the ISA. Hennessy and Patterson variously claim the maximum is 17 or 18 bytes [7, p.A-23, p.K-41, p.14], but do not provide examples or cite their sources. It is unclear where these values come from, as whilst the maximum instruction length has changed during x86's development it has never been documented to be 17 or 18 bytes; the Intel 8086 did not have a maximum instruction length, the 80286 introduced a 10 byte limit and a 15 byte limit has been documented since the 80386 [123].

Searching the instruction space. The astonishing complexity of x86 instruction encoding is an immense source of frustration for instruction fuzzing. As discussed in Section 1.1, it is impossible to test all possible encodings of 15 bytes in a feasible length of time. This naturally raises the question of whether the meaningful search space could be reduced. One obvious approach is to attempt to cover most opcode encodings by testing all encodings of 3 bytes (for a total of 16,777,216 encodings), which is explored in Section 4.2.3. Ideally, all encodings of 5 bytes would be tested to also include the ModR/M and SIB bytes. However, this increases the search space to a (currently) infeasible size of 1,099,511,627,776 encodings, and also reduces stability as instruction encodings are so ambiguous: for many instructions these extra bytes will be interpreted as immediate or address values rather than as ModR/M and SIB bytes. An alternative is the only known algorithm developed for searching the space, the tunnelling algorithm introduced by Domas in the Sandsifter tool [5]. It makes the assumption that changes in an instruction's length indicate changes in functionality. Starting from an initial instruction of 15 zero bytes, the instruction is executed, its length is determined, and its last byte (based on its length) is incremented. This is repeated until the observed length changes, whereupon the new last byte is incremented. If all possible values of the last byte have been exhausted with no length change, the incrementation moves to the second-to-last byte. This reduces the search space down to under 1,000,000,000 instructions. This approach has the benefit that no prior knowledge of the instruction set is required, which makes the algorithm portable across variable-length architectures provided it is possible to determine an instruction's length both when it faults and when it executes successfully. It is also more likely to find deliberately concealed instructions than an algorithm guided by knowledge of the ISA (such as the instruction encodings defined in the specification). However, the assumption that an instruction's functionality is solely determined by its length is substantial. Domas, for example, notes that a change in exception type might also indicate a functionality change, but did not implement this in the publicly-available version of Sandsifter [5].

```
push (address of escape)
call trampoline

speculoop:
  pause
  lfence
  jmp speculoop

trampoline:
  lea rsp, [rsp+8]
  ret

escape:
...
```

Listing 2.1: An example retpoline.

Determining instruction length. Instruction length of valid instructions can be determined using the trap flag, which triggers a #DB exception after a single instruction has executed. Using a user-mode signal handler, the address of the instruction pointer reported in the exception can be obtained, and subtracting this from the known start address of the instruction bytes provides the instruction length. However, this fails for faulting instructions (such as privileged instructions, or instructions valid only in other modes) because these exceptions take priority over #DB and report the start address as the instruction pointer. [5] overcame this by monitoring page faults: one byte of the instruction is placed at the end of an executable page (with a non-executable page following), the trap flag is set, and the instruction is executed. If the instruction is longer than one byte, a page fault exception occurs with the CR2 register set to the address of the page boundary; otherwise, the trap flag will trigger a debug exception. The exception type and CR2 value can be checked by registering a signal handler, and then the process can be repeated with two bytes if the instruction length has not yet been determined.

Intel TSX. Intel Transactional Synchronization Extensions (TSX) are a set of instructions introduced in the Haswell microarchitecture for hardware transactional memory support. (The AMD equivalent is the proposed, but not yet implemented, Advanced Synchronization Facility extension.) TSX essentially provides the architectural illusion that instructions within a transaction are being executed as a single atomic instruction: architectural updates will only be visible to other logical processors when the transaction completes, and all memory operations will appear to have occurred instantaneously. If the transaction aborts before completion, all updates are discarded and the prior architectural state is restored. Two software interfaces are provided: Hardware Lock Elision (HLE), and Restricted Transactional Memory (RTM). For the purpose of this project, the crucial difference between the two is that if a HLE transaction fails then execution restarts from the beginning of the HLE section (an instruction prefixed with `xacquire`), whereas a distinct fallback path (akin to `try/catch` exception handling) can be specified for an RTM transaction. Developers "can use any instruction safely inside a transactional region", but various conditions may cause a transaction to abort, such as interrupts, exceptions (particularly page faults), certain instructions (such as `syscall`), setting the Accessed/Dirty flags of a page table entry (which occurs on the first access/write to a page), conflicting data accesses by other logical processors, or limited transactional resources. The transaction's read set and write set are tracked in the L1 cache, which means that on Haswell, Broadwell, and Skylake a transaction writing to 9 or more different locations in the same cache set will abort [123]. Crucially however, whilst an exception within an RTM transaction will cause an abort, the exception itself is suppressed and so no exception handling is necessary. RTM provides a limited set of abort status codes placed in the `EAX` register for determining the cause of an abort. In the majority of cases the status code is 0, and it is not possible to determine the cause of the abort, for example to distinguish between a segmentation fault and an illegal instruction exception. However, of interest to this project is that manual aborts due to the `xabort` instruction and debug exceptions can be distinguished from other causes, respectively by bit 0 being set (with the argument provided to `xabort` in bits 31:24) and by bit 4 being set [91].

Retpoline. A `jmp` or `call` is indirect when the address to transfer control to is not an immediate value: the CPU must calculate or load the address. As this causes a delay, on Intel CPUs the indirect branch predictor

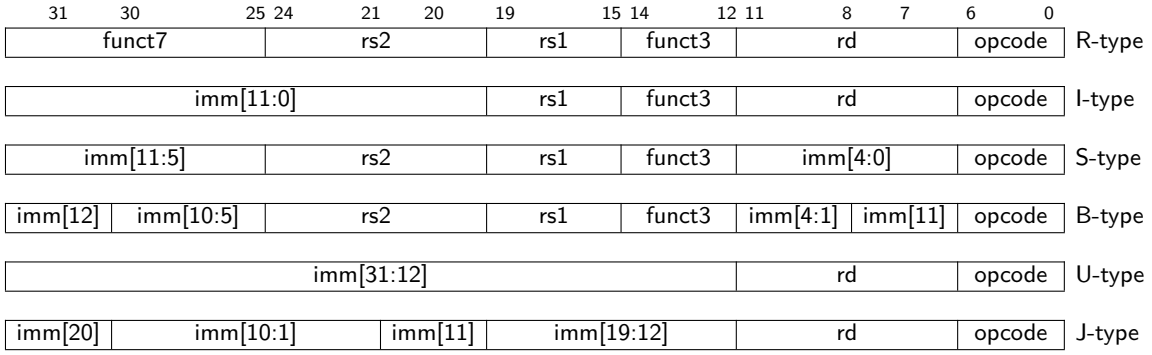
speculates the target of the indirect branch and speculative execution occurs from this address until the branch target address is determined. This is the basis for the Spectre v2 (branch target injection) vulnerability: an attacker on the same CPU core can deliberately mistrain the indirect branch predictor in a process under their control [21]. After the mistraining, speculative execution in the victim process will occur at the address of a 'gadget' in the victim's code. The gadget is a code sequence which unintentionally leaks sensitive data via a side channel (e.g. it results in a data-dependent memory address being loaded into the cache, which can be detected by the attacker with a cache timing attack). The retpoline is one of the primary mitigations for this vulnerability, with sensitive code (the kernel and web browsers in particular) being recompiled so that all indirect branches are replaced by the retpoline construct. Listing 2.1 shows one possible retpoline implementation (note that on 32-bit architectures `[rsp+8]` should be replaced with `[rsp+4]`). First, the address of the actual indirect target is pushed to the stack; the CPU's indirect branch predictor is not influenced by this. Next, an indirect call is made to the 'trampoline' section. This section adjusts the stack pointer so that the `ret` instruction will return to the address of 'escape'. However, the CPU incorrectly speculates that it will return - as any normal function would - to the instruction immediately after 'call trampoline', i.e. the speculative loop ('speculoop') section, and so speculatively executes from here. These instructions are specifically chosen to trap speculation to avoid leaking sensitive data: the `jmp` instruction creates an infinite loop and then the `pause` and `lfence` instructions somewhat inhibit speculation to mitigate the performance impact of this loop (Intel are ambiguous about the exact effect they have on speculation; see [126]).

2.3.2 RISC-V

Full documentation of the ISA is available in [127], [119], and [128]. [129] discusses the ISA's design choices at length.

RISC-V is a new ISA under development since 2010. Originally designed at UC Berkeley for education and research purposes, it is now being promoted as a free and open-source architecture for use across the entire CPU industry. As a RISC ISA, it features far fewer, simpler instructions than x86, but is designed to be highly extensible so it can be adapted for use in a variety of computing contexts. A RISC-V CPU must at a minimum support either the base integer extension I or the base extension for embedded platforms E, but can also be extended with instructions for multiplication and division (M), floating-point operations (F, D, and Q for single, double and quad precision respectively), atomic instructions (A), and compressed instructions (C). The terms RV32, RV64, or RV128 describe an implementation's datapath width, and the extension name G is used as a shorthand for extensions I, M, A, F, and D combined; the specification of these extensions has been 'frozen' so that they are a stable implementation target for CPU designs. In comparison to x86, RISC-V is still relatively volatile. Many extensions are still in draft (such as B for bit manipulation and V for vector operations), as are the privileged and debug instruction specifications and memory model [119] [128]. Instructions are 'fixed-length' in the sense that all extension G instructions are 32-bit. However, the Compressed extension adds support for 16-bit instructions and has been widely implemented due to its benefits for performance and energy efficiency [127]. The ISA is also designed to support instructions any multiple of 16-bit instruction 'parcels' in length, including instructions longer than 192 bits! Bits in the least significant byte of an instruction identify its length; for the 16- and 32-bit instructions considered in this project, this is determined by bits 1:0 (numbering from the least significant bit), with 11 identifying a 32-bit instruction and 00, 01, and 10 identifying a 16-bit instruction's quadrant in the C extension. Unlike x86's rather ambiguous instruction format, RISC-V has only a small number of well-defined instruction formats; the 16-bit and 32-formats are illustrated in Figure 2.6.

The ISA features 32 general-purpose integer registers and, on microarchitectures supporting the F, D or Q floating-point extensions, a separate floating-point register file with a further 32 registers. The integer register length `XLEN` is 32, 64, or 128 bits for RV32, RV64, and RV128 respectively; similarly, the floating-point register length `FLEN` is 32, 64, or 128 bits depending on the highest precision floating-point extension implemented (F, D, and Q respectively). As shown in Table 2.7, both sets of registers are addressed using the binary values 00000 to 11111 and therefore the register type is inferred based on the instruction the register is encoded within. The instructions `fmv.x.w,d,q` and `fmv.w,d,q.x` (available on RV64 and RV128 only) enable values to be transferred between the integer and floating-point registers. For the compressed instruction formats which use 3-bit register encodings, only the registers `s0` - `a5` (integer) and `fs0` - `fa5` (floating point) are available and are encoded with the values 000 to 111.



Format	Meaning	15 14 13 12	11 10 9 8 7	6 5	4 3 2	1 0
CR	Register	funct4	rd/rs1	rs2		op
CI	Immediate	funct3 imm	rd/rs1	imm		op
CSS	Stack-relative Store	funct3	imm	rs2		op
CIW	Wide Immediate	funct3	imm		rd′	op
CL	Load	funct3	imm rs1′	imm	rd′	op
CS	Store	funct3	imm rs1′	imm	rs2′	op
CA	Arithmetic	funct6		rd′/rs1′ funct2	rs2′	op
CB	Branch	funct3	offset rs1′	offset		op
CJ	Jump	funct3	jump target			op

Figure 2.6: The RISC-V 16- and 32-bit instruction formats, reproduced from [127].

2.3.3 Microarchitectures under Test

Comprehensive discussion of each microarchitecture's is beyond the scope of this section; for further information regarding their pipelines, the most useful resource is [130].

Intel Bonnell. The Bonnell CPU under test is the Intel Atom N270 1.60GHz, which is an ultra-low power single-core CPU for the 32-bit x86 ISA supporting the MMX, EMMX, and SSE through SSSE3 extensions. It has a 16-stage pipeline which is less complex than the Haswell/Broadwell pipeline described in Section 2.2.6; it features only two instruction decoders and two execution ports, and supports branch prediction but not out-of-order or speculative execution [130]. It has a a 32 kB 8-way L1 instruction cache, 24 KB 6-way L1 data cache, and 512 KB 8-way L2 cache, with no L3 cache [131]. Relevant errata to the project are AAG18 concerning unsynchronized cross-modifying code, which is not a concern provided the test program is locked to a single logical core (whilst single-core, the CPU supports multithreading so has two logical cores), and the more troubling AAG40, which states that instruction page remapping or self-/cross-modifying code may lead to "unpredictable system behavior" under a "complex set of internal conditions" [132]. The ambiguity in this description unfortunately means that such conditions cannot be avoided experimentally, so although suspicious behaviour was not observed in testing, this may have affected the results.

Intel Westmere. The Westmere CPU under test is the Intel Core i3 M330 2.13GHz, a dual-core mobile CPU for the 64-bit x86 ISA supporting the MMX, EMMX, and SSE through SSE4.2 extensions. Multithreading is supported, so the CPU has 4 logical cores. The pipeline appears to be at least 18 stages, based on microbenchmarking [130], with 4 instruction decoders and a predecoder for instruction length detection, branch prediction, out-of-order execution, and speculative execution. It has a 64KB 4-way L1 instruction cache, 64 Kb 8-way L1 data cache, 512 KB 8-way L2 cache and 3 MB 12-way L3 cache. Westmere is a die shrink of Nehalem, so most microarchitectural data available for Nehalem applies. Unfortunately the Westmere platform (Dell Inspiron 1564) suffered hardware failure early in the project (believed to be unrelated to any instruction testing!), so few results could be obtained on this microarchitecture.

Intel Broadwell. The Broadwell CPU under test is the Intel Core i7-5600U 2.6GHz. Similarly to the Westmere CPU, it is a dual-core mobile model supporting 64-bit x86 with the MMX, EMMX, SSE through SSE4.2, AVX2, and TSX extensions. As in Westmere, there are 4 instruction decoders and a predecoder, with a maximum of 16 bytes of instructions fetched per cycle and no cycle penalty for decoding multiple prefixes, and

Name	Encoding	Description
zero	00000	Hard-wired zero (writes ignored)
ra	00001	Return address
sp	00010	Stack pointer
gp	00011	Global pointer
tp	00100	Thread pointer
t0	00101	Temporary/alternate link register
t1-2	00110-00111	Temporaries
s0/fp	01000	Saved register/frame pointer
s1	01001	Saved register
a0-1	01010-01011	Function arguments/return values
a2-7	01100-10001	Function arguments
s2-11	10010-11011	Saved registers
t3-6	11100-11111	Temporaries
ft0-7	00000-00111	FP temporaries
fs0-1	01000-01001	FP saved registers
fa0-1	01010-01011	FP arguments/return values
fa2-7	01100-10001	FP arguments
fs2-11	10010-11011	FP saved registers
ft8-11	11100-11111	FP temporaries

Figure 2.7: Integer and floating-point (FP) architectural registers on RISC-V, adapted from [127].

support for branch prediction, out-of-order and speculative execution; at a high level, the microarchitecture is identical to Haswell, as described in Section 2.2.6. Relevant errata to the project are that numerous hardware performance counters have accuracy issues (these were avoided where possible, although unfortunately the `INST_RETIRED.ALL` counter crucial to Section 4.4.2 is affected by errata BDD11 and BDD53, so the number of instructions retired may be inconsistently over- or under-counted); several instructions are noted to produce the wrong exception with certain instruction encodings (errata BDD14, BDD23, BDD39, BDD74) and there are multiple possible HCF instruction conditions (BDD5, BDD86, and BDD100), although none of these were encountered in testing; and finally Intel TSX may "behave unpredictably...under a complex set of internal timing conditions and system events" (BDD99) [87]. It is important to note that whilst there are indeed many errata affecting this microarchitecture (119, including some fixed errata), it is by no means a particularly flawed microarchitecture which should have therefore been avoided; rather, the number of errata is indicative of the many verification and manufacturing challenges involved in producing a CPU. This highlights both the necessity of CPU auditing tools and also their limitations; runtime instruction testing is highly valuable, but accuracy cannot be guaranteed when the mechanisms available for measuring instruction behaviour are inherently flawed.

SiFive Freedom U540. The SiFive HiFive Unleashed board's U540 is the first commercially available Linux-capable multi-core RISC-V SoC, featuring one E51 monitor core and four U54 application cores. The E51 supports RV64IMAC and the machine and user modes with a 16KB 2-way L1 instruction cache and 8KB data tightly integrated memory (DTIM). The U54 cores support RV64GC, machine, supervisor and user modes, and have separate 32KB 8-way L1 instruction and data caches. Writes to instruction memory must be synchronised with the `fence.i` instruction. Both cores have single-issue, in-order 5-stage pipelines with branch prediction. Relevant errata include that the high 24-bits of virtual and physical addresses are 'sometimes' unchecked (ROCK-2) and that atomic operations are not ordered correctly on the E51 (ROCK-3) [117].

SiFive E31. The HiFive1 microcontroller features the SiFive E31 CPU as part of the Freedom E310 SoC, which has a a single-issue, in-order 5-stage pipeline supporting RV32I machine mode and the M, A, and C extensions. It features branch prediction, a 16KB DTIM and a 6 KB 2-way set-associative instruction cache; writes to instruction memory must be synchronised with the `fence.i` instruction [133].

Chapter 3

Implementation and Results: RISC-V

"Expected results: Squid is restarted. Actual results: All files are deleted on the machine."

Swapna Krishnan [134]

This project's contributions on RISC-V are as follows:

- The first known investigation of undocumented instructions on RISC-V, using the SiFive HiFive Unleashed and HiFive1 platforms.
- Identification of 2048 undocumented instructions on the HiFive Unleashed, at least 608 of which modify architectural state, and partial reverse-engineering of their instruction format and functionality.
- A method for reducing the size of the meaningful instruction search space by 94%.
- Identification of bugs in a popular RISC-V disassembler.
- A preliminary method for inferring instruction functionality from register state changes.
- Debugging of the HiFive Unleashed bootloader to resolve a failure to boot with the Debian demo image, which resulted in a small patch to the SiFive Freedom Unleashed Software Development Kit [135].
- A version 2 release of the OpcodeTester tool compatible with both RISC-V Linux and SiFive Freedom Metal incorporating the above contributions, released as open-source on GitHub [2].

3.1 Initial Framework

OpcodeTester currently only supports GNU Linux environments (and Freedom Metal for the HiFive1). However, with slight modification the code should be portable to other POSIX operating systems, and with more substantial modification even to Windows. Windows' implementation of exception handling is quite different, but the broader concepts such as making a memory page executable and registering an exception handler remain the same.

As described in [2], in my prior research project I developed an initial testing framework for integrating execution of arbitrary machine code into a larger test program in C. Undocumented instructions have no assembly mnemonics, so this cannot be achieved with inline assembly code (arbitrary bytes can be inserted with `.byte`, but this does not support variable byte values). Instead, custom machine code can be run by creating an array of unsigned characters (the *test array*) either as a global variable or allocated on the heap, making the page containing the array executable (with `mprotect`), and then creating and calling a function pointer to the array. Additionally, a mechanism is needed for restoring control flow after each instruction test. For instructions which run without faulting, this is as simple as placing the relevant byte(s) for a `ret` instruction, `82 80` on RISC-V and `90` on x86, after the test instruction bytes (although see Section 4.3.1 for concerns regarding using this on x86!).

To manage instructions which fault, the test program must register a signal handler for the POSIX signals `SIG_ILL`, `SIG_SEGV`, `SIG_TRAP`, `SIG_BUS` and `SIG_FPE` [1], and modify the instruction pointer (IP) in this signal handler so that the program can continue after the faulting instruction. With the exception of `SIG_TRAP`, the corresponding exception for each of the caught signals returns the address of the faulting instruction as the IP to return to, so the program will loop repeating the same instruction over and over again unless the IP is modified in the signal handler. In my prior project, I used Sandsifter's method of modifying the IP within the `mcontext` struct to point to a global assembly label positioned after the instruction call. Executing arbitrary machine code in a kernel driver can be similarly constructed with equivalent kernel functions; however, exception handling is far more challenging, which is discussed further in [2].

3.2 Porting OpcodeTester to RISC-V

3.2.1 HiFive Unleashed

A minimal framework to test arbitrary machine code on the HiFive Unleashed is provided in Listing 3.1; the actual test framework incorporates more safeguards, such as a separate stack for the signal handler.

I began porting the tool to RISC-V to run in the HiFive Unleashed's default buildroot OS. `mcontext` is not fully implemented in the default version of `glibc` so cannot be used to modify the IP in the signal handler. I replaced this mechanism with `sigsetjmp` and `siglongjmp`. These functions implement a non-local `goto`, additionally restoring the execution context and signal mask. `sigsetjmp` returns with a different value when it is called after `siglongjmp` has restored the context, enabling the faulting instruction to be skipped. The definition of 'execution context' varies by C standard library implementation and is architecture-specific, but typically entails unwinding the stack and restoring a limited set of registers. The values of non-volatile local variables are undefined after the context is restored, so contrary to standard practice most variables in OpcodeTester are global. (However, the `.data` segment may still occasionally be corrupted by undocumented instructions.) To further guard against this corruption I provided the signal handler with a separate stack.

Despite the `sigsetjmp` mechanism, testing was frequently interrupted by program hangs which required a manual restart of testing: many of the tested instructions failed to return correctly. After several unsuccessful attempts to prevent these failures, I accepted that some instructions will always corrupt control flow and amended the test program to fork a separate child process for each instruction. Note that it is crucial to manually set the behaviour of `SIGCHLD` to `SIG_IGN` in the parent process to avoid hitting the forked processes limit; without this, the child processes live on as zombies in the process table [136]. As this prevents corruption of the parent process' memory or control flow, stability is significantly improved, although program hangs can still occur if the parent becomes stuck waiting for the child process' exit signal. It is crucial for the parent to wait before forking the next test process, as otherwise results are output in the wrong order; the parent will typically receive an exit signal if the child process exits normally or is killed by the operating system due to repeated exceptions. However, certain control flow corruption (i.e. producing an infinite loop) will cause the child process to never exit, making the parent process hang. I experimented with manually killing each child process from the parent process but found this did not prevent the issue.

3.2.2 HiFive1

The minimal framework for testing arbitrary machine code on the HiFive 1 is provided in Listing 3.2.

The HiFive1 supports machine mode only. Whilst operating systems such as FreeRTOS and Zephyr have been ported to the board, I chose to target SiFive's Freedom Metal framework, as it promises to enable code portability across all future SiFive microcontrollers. As the framework is very new and still under development this was somewhat risky, but I felt it was justified by the resulting enhanced portability. The framework provides a bare-metal C environment and API for accessing CPU features.The CPU's volatile memory is executable by default, so no alteration of memory protection is necessary [133]. POSIX signals are unsupported, but the CPU API enables exception handlers to be registered for exceptions 0-11 (8-10 are reserved with no documented functionality; 12-31 are also reserved, but attempting to register a handler for them

[1] This is not the complete set of POSIX signals, but across millions of tested instructions I have not yet witnessed any other signal.

returns an error). `longjmp` must be used to return from a fault rather than the method provided in the documentation using API calls `metal_cpu_get_exception_pc`, `metal_cpu_get_instruction_length`, and `metal_cpu_set_exception_pc`, as this method sometimes causes an infinite loop within the Freedom Metal exception handling code. I attempted to debug this without success, as debugging the HiFive1 with gdb via OpenOCD is extremely buggy with frequent crashes (particularly errors concerning invalid register cache entries). On a fully variable-length architecture I would assume that the culprit was an incorrect instruction length being returned, but given the predictability of RISC-V's instruction length encoding (determined by bits 1:0) this seems improbable. There is no support for processes, so it is not possible to fork and test each instruction in a separate process, which increases the frequency of program hangs. Use of `fflush(stdout)` seems to produce erratic behaviour, with frequent program hangs; fortunately this is unnecessary on the HiFive1 due to the lack of child processes (on the Unleashed it needs to be flushed to ensure output ordering and prevent the children inheriting the contents of the parent's buffer).

3.3 Inferring Instruction Functionality

Register state. I implemented a comparison of register state before and after instruction execution to determine if undocumented instructions which successfully execute might be register operations, recording the state of all integer and floating-point registers in addition to the floating-point control registers `fcsr`, `frm`, and `fflags`. `ra` and `a5` are always modified, so the tool ignores these (the compiler uses `a5` to store the pointer to the instruction array, and to index the arrays used to store the register state; with a different toolchain this might be altered). The values of the floating-point control registers cannot be read directly and must be copied into other registers, so `t3–5` are used for this, with their 'before' values recorded afterward and their 'after' values recorded before the floating-point control registers are read again. Motivated by the observed effects of the RES1 and RES2 undocumented instructions (see Section 3.7), I then attempted to automate identification of some arithmetic instructions by comparing register state over multiple tests of each instruction. However, I found this was of limited utility and ultimately identified most instruction functionality manually.

Performance counters. On x86, the hardware performance counters provide extremely valuable fine-grained insight into the microarchitectural effects of instructions, and can be used to be infer instruction functionality (for example, cache hits or misses demonstrate that the instruction is attempting a load). I attempted to implement performance counter monitoring on the HiFive Unleashed but unfortunately found that the counters are too unreliable to provide any useful insight into instruction functionality. The store counter, for example, always reads the same value irrespective of how many store instructions have been executed. In debugging this issue I confirmed that the compiler orders the assembly instructions correctly and does not mistakenly overwrite the registers containing the counter values before they are printed to the console; the behaviour persists even with serializing instructions placed either side of the executing instruction(s) and with a long delay to allow the counter to update. In fact, the counter value remains unchanged between program runs until the system is rebooted, suggesting the counter is completely inoperative and is randomly initialized at boot time. Beyond their inaccurate values, a further challenge is that hardware performance counters can only be configured in machine mode, rather than in supervisor mode as on x86. This means that the bootloader must be amended with code to initialize the programmable counters `counter3` and `counter4` (just two events can be configured at a time) and reflashed to the board's SD card in order to monitor different counters - a significant bottleneck compared to the multiple counter changes per second achievable on x86! A solution to this would be to implement a machine mode driver (akin to an x86 SMM driver) to run continuously alongside the operating system and provide an interface for modifying counters at runtime; unfortunately I was unable to implement this within the time constraints of the project. The HiFive1 does not implement any performance monitoring counters (beyond a real-time clock).

3.4 Challenges

Disassembler bugs and variable-length instructions. During the course of testing I discovered that many of the instructions identified by the disassembler as illegal were false positives. [4] does not recognise any of the compressed microarchitectural hint instructions as valid; possibly it was produced using an older version of the RISC-V specification before these were defined. Removing these false positives from the results was straightforward; more challenging however was tackling these false positives when testing '32-bit' instructions actually interpreted as two 16-bit instructions. Initially, I determined if an instruction was documented by

disassembling the entire 32 bits. However, if the disassembler detects a 16-bit instruction in the two least significant bytes, it will disassemble this instruction only and ignore the other two bytes. To add to the confusion, the disassembler 'lifts' the compressed instructions into their equivalent 32-bit mnemonics (a somewhat bizarre design choice), so it was not initially apparent that this was occurring. This challenge posed a major bottleneck for my investigation of RISC-V: I initially believed that the memory corruption and program hangs caused by execution of a valid 16-bit instruction (e.g. `c.addi4spn`) and a false positive 'illegal' 16-bit instruction actually indicated undocumented 32-bit instructions, and spent significant time ($>$ 1 week) identifying and blacklisting these instructions in an attempt to complete a full exhaustive search of the instruction space.

I found that for all instructions interpreted as 16-bit by the disassembler (bits 1:0 != 11) it is necessary to disassemble the upper two bytes and lower two bytes separately and - in addition to skipping instructions where both sections are valid 16-bit instructions - to employ two heuristics: all instructions with a valid lower two bytes are skipped, and all instructions with an invalid lower two bytes but a valid `c.addi4spn` instruction in the upper bytes are also skipped. This does mean that some entirely invalid instructions are skipped in testing, but entirely mitigates the many program hangs and crashes I observed due to valid 16-bit instructions executing and corrupting program state. Ultimately, these issues could be resolved by implementing the microarchitectural hint instructions in the disassembler, but due to time constraints I was unable to complete this. Another potential solution would be to add a full function prologue and epilogue to the test bytes (to set up / tear down a stack and restore clobbered registers); I did not implement this as in the vast majority of cases it is unnecessary and adds significant overhead to performance monitoring values (assuming on future microarchitectures the performance counters are usable).

Beyond the disassembler's false positives, the fundamental challenge is that having any kind of variability in instruction length complicates instruction interpretation; the extreme 1 to 15 byte variability of x86 is by no means necessary to produce this. Variable-length instruction encoding is therefore a security concern as it renders all tasks involving instruction interpretation significantly more challenging, including not only instruction testing but also tasks such as producing emulators and virtualisation software or reverse-engineering obfuscated malware.

Self- and cross-modifying code. When using self-modifying code as in the case of the test array, it is important to ensure that the changes are visible to the CPU before the code is executed. The RISC-V memory model provides no guarantee that stores will be visible to instruction fetches on the same hart (hardware thread) unless a `fence.i` instruction is executed after the store to serialize the instruction and data streams [127]. I initially overlooked the necessity of including this, which produced erroneous results as the instruction under test was often an instruction prior to the instruction I believed I was testing. Additionally, a `fence` instruction must be executed to prevent cross-modifying code if the process is rescheduled onto a different hart which has not yet observed the changes made on the previous hart. Unfortunately the RISC-V specification permits both `fence` and `fence.i` to be implemented as a `nop` to reduce hardware complexity [127], so there is no guarantee a RISC-V microarchitecture will provide a means of serializing the instruction and data streams. Due to the visible behaviour change this is clearly not the case for `fence.i` on the HiFive Unleashed. However, as I was unsure of the exact implementation of `fence` and observed no behaviour change when using it, I opted to additionally lock the test program to a single hart using a call to `sched_setaffinity` to ensure no cross-modification could occur; child processes inherit the processor affinity of their parents, so each child process will also test its instruction on the same hart.

Of the two mechanisms, `fence.i` appears to be most crucial; it is important to be aware that the tool may produce erroneous results on other microarchitectures which implement the instruction as a `nop` or provide only weak serialization guarantees, and indeed this appears to be the case on the HiFive1. The manual states that "the instruction cache is not kept coherent with the rest of the platform memory system" and that `fence.i` should be used when modifying instruction memory, which suggests that it is implemented. However, despite using `fence.i` I repeatedly observed documented instructions executing; these are skipped in the test program and should never execute unless the instruction cache fails to remain in sync with writes to the test array, so this behaviour suggests that the HiFive1's serialization is too weak to reliably support self-modifying code. Surprisingly, whilst there is some variance in the documented instructions which trigger this across runs, the behaviour typically occur in the same encoding regions (in particular the region 0100111000100000 to 0101111111111111) rather than occurring randomly. This still occurs even with precautions to slow down execution such as adding long sections of unrelated code to modify the instruction cache and repeatedly inserting `fence.i` and `fence` instructions (which clearly do have some effect on the pipeline, as excessive

usage severely slows execution).

Brute-force search. RISC-V's instruction space is theoretically small enough to be searched exhaustively: there are 4,294,967,296 possible 32-bit encodings, 3,464,099,880 of which the RISC-V disassembler produced by Clark [4] recognises as valid, leaving 830,867,416 to test. Using the disassembler to skip all documented RISC-V instructions, assuming a perfect hardware implementation and a fully correct disassembler then every instruction tested should fault with a single illegal instruction exception, with no change to architectural state and therefore no crashes or memory corruption. However, in practice such a search is infeasibly slow, particularly on the HiFive1. After multiple days attempting an exhaustive search on the HiFive Unleashed (with frequent intervention required to transfer log files and restart the program after crashes) I abandoned the attempt. The time invested was preventing progress in other areas of the project, and all results featured either false positives (the microarchitectural hints) or two specific encodings, which I subsequently investigated further (see Section 3.7). One interesting outcome of the search, however, was identifying that the speed of the HiFive Unleashed's heartbeat LED varies. I initially thought this was linked with CPU or disk activity (and therefore could act as a side channel), but an increase in blink speed seems to be permanent (until the system is rebooted) after repeated undocumented instruction testing; this does not appear to be linked to execution of a specific instruction and does not occur in normal usage of the system.

Logging output on the HiFive1. Using the HiFive1 in machine mode with the Freedom Metal framework there is no support for file creation, so the only logging method is to output results with `printf` and view these on another computer via the UART console. However, there is a limit to how many lines can be buffered with this method, and attempting to simultaneously write the output to a file on the host computer (such as with `screen -L`) produce garbled output. This limits the extent to which testing can be automated as the output must be repeatedly copied to a file manually before it is lost. Printing no output for instructions which correctly fault with an illegal instruction exception helps reduce the quantity of output but does not resolve the problem entirely.

Kernel output. The HiFive Unleashed runs the buildroot Linux kernel in debug mode, so all `printk` messages logged by the kernel appear immediately on the console. However, this output is only visible when connected via USB and not when connected via `ssh`; as such these silent errors can easily go undetected when using `ssh`, leading to the incorrect assumption that an instruction does nothing when in fact it causes an unhandled exception or even a kernel oops.

Testing on Debian. I attempted to set up the Debian demo image provided by SiFive so that I could compare the results of instruction testing in a full Linux environment compared to the minimal environment provided by buildroot; I was concerned there might be differences in exception handling. This required debugging of the bootloader which resulted in a patch to the Freedom Unleashed Software Development Kit [135], as the prior boot configuration led to a kernel panic on boot. Preliminary testing on Debian appeared to replicate the results on buildroot. Unfortunately, I was unable to obtain full results after an update prevented the `ssh` server from starting; this meant I could no longer transfer files.

Undefined behaviour. Finally, it is important to note that several of the practices used in the test framework, such as continuing execution after handling a segmentation fault and using asynchronous functions after calling `siglongjmp`, technically produce undefined behaviour according to the C specification [137]. I am not aware of any alternatives to these mechanisms which avoid this; when the entire premise of the tool is to break the architectural 'rules', some use of undefined behaviour appears to be inevitable. These techniques compile and execute correctly with GCC version `riscv64-linux-gcc.br_real (Buildroot 2019.02-07449-g4eddd28f99) 8.3.0` and functioned correctly on both the HiFive Unleashed (with buildroot and the Debian demo image) and the HiFive1. However, it is possible that this may not be the case with a different toolchain; as the behaviour is undefined, future compilers have no obligation to produce the same behaviour. Similarly, with a different Linux kernel or Freedom Metal version the tool may not function correctly if exception handling or `sigsetjmp` were to be substantially modified.

3.5 Reducing the Instruction Search Space

As an exhaustive search proved infeasibly slow I identified a 'functional pattern' for RISC-V 32-bit instructions, FUNCT, as shown in Figure 3.1, based on the instruction fields documented to alter instruction functionality

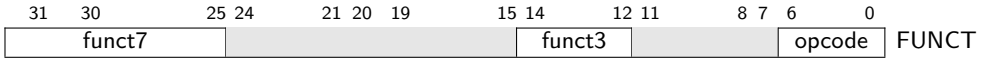


Figure 3.1: The functional pattern for 32-bit instructions.

Platform	Test	Run	Seg.	Illegal	No Ret
Unleashed	RES1/2 16-bit	1024	896	0	128
	All 16-bit (excl. RES1/2)	56	0	8231	0
	Functional 32-bit	3297	1	16170	0
HiFive1	RES1/2 16-bit	728	1300	20	0
	All 16-bit (excl. RES1/2)	32	0	8255	0

Table 3.1: Experimental results on the HiFive Unleashed and HiFive1.

[127]. The pattern covers bits 31-25 (for funct7), 15-12 (for funct3), and 6-0 (for the opcode field); theoretically this is sufficient to cover all implemented instruction behaviour, with the other bit fields modifying operands without significantly altering functionality. In this interpretation, a `mv` instruction is still a `mv` regardless of its source and destination operands; however, as discussed earlier there are varying definitions of what constitutes an instruction variant (see Section 2.3), and this does make the assumption that no undocumented instructions are defined by a specific pattern of operand bits (which might occur if an undocumented instruction were deliberately implemented and hidden in the design).

Testing the FUNCT pattern reduces the search space to 262,144 instructions (a reduction of 94%) and the test space to just 75,823 instructions the diassembler considers illegal. By keeping all other bits set to 0, crashes and memory corruption are significantly reduced as operand values of 0 will typically either leave architectural state unaltered (such as adding 0 or writing to the zero register) or cause a segmentation fault (rather than modifying memory within the test program's address space).

3.6 Results

Many of the undocumented instructions found on the HiFive Unleashed were false positives (microarchitectural hints). However, 2048 were indeed undocumented instructions of the form 100xxxxxxxxxx00. This is an encoding from quadrant 0 of the Compressed RISC-V extension which is reserved for future standard extensions [127]. It should therefore fault on current microarchitectures with an illegal instruction exception, and should never be used for implementing custom instructions. However, as shown in Table 3.1 1024 of the instructions execute, 896 cause a segmentation fault, and 128 seemingly cause memory or control flow corruption, as the child process is killed before it can output results. Similarly, the undocumented RES1/2 also execute on the HiFive1, although with markedly different exception behaviour, and were the only undocumented instructions found; the 32 non-RES1/2 instructions which run in the all 16-bit encodings test are again false positives due to the disassembler. These RES1/RES2 instructions are discussed further in the next section.

Due to the erratic behaviour of the HiFive1 I was unfortunately unable to obtain meaningful results from the 32-bit functional pattern search; control flow was repeatedly corrupted, seemingly defying any attempts at blacklisting individual instructions or binary patterns. These problems also occur to a lesser extent on the other tests; testing the RES1/2 encodings shows erratic exception behaviour for 6 instructions, but no corruption of control flow, whilst testing the 16-bit space required substantial manual intervention and blacklisting of specific documented instructions due to its weak serialization of the instruction and data streams.

3.7 Reverse-Engineering RES1 and RES2

SiFive do not have a responsible disclosure process or designated security contact; I disclosed my findings via email on 22nd April 2019 but have received no response. The HiFive Unleashed is a development board rather than a consumer desktop platform and is not in widespread use, and I have not yet identified any way in which

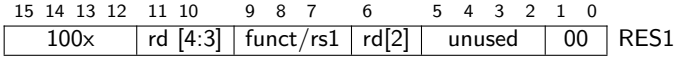


Figure 3.2: RES1, the main reverse-engineered undocumented instruction format; RES2 features value 111 in the funct/rs1 field and appears to also use (but not consume) the data in the 16 bits following the instruction.

Funct/rs1	Effect
000	Load 139264 into rd
001	Segmentation fault
010	Load 0 into rd
011	Segmentation fault
100	Segmentation fault
101	Load 0 into rd
110	Load 0 into rd
111	Load 32-bit immediate (RES2)

Table 3.2: The funct/rs1 encodings.

these instructions could be used to construct an exploit, so I believe that publishing these results does not violate my project commitment to responsible disclosure.

The RES1 instruction format is highly counter-intuitive and required substantial effort to determine. I began by trying to match the results to any of the documented 16-bit formats, but none of these were matched by the binary patterns. For example, the patterns 1001111110xxxx00 and 1001111111xxxx00 appear to encode the same source but different destination registers (`s8` and `t3` respectively; they both appear to be loaded with the value 18446744071570628480+4z, with the z multiplier encoded by the 4-bit offset field xxxx; however, as described below the value loaded is actually encoded by the RES2 format). These patterns only differ in bit 6 (counting, as in the RISC-V specification, from right to left starting with bit 0) and therefore this bit must be part of the destination register field `rd`, but no documented instruction format in the Compressed extension features this. Other encodings demonstrate the same, with the only reliable register encoding patterns occurring split across the rd[4:3] and rd[2] fields. All register encodings require an extension of 00 in bits 1:0 to match the binary encodings specified in Figure 2.7; this is further supported by the fact that no register with bits 1:0 != 00 is ever visibly modified by these instructions (registers `zero`, `tp`, `s0`, `s4`, `s8`, `a2`, `a6` and `t3` are modified). Whilst it is an extremely surprising result, these instructions do appear to use an entirely new instruction format. Encodings 1000 and 1001 for bits 15:12 appear to have identical functionality.

The next stage of analysis was determining that there are two distinct instruction formats: a 16-bit format (RES1) for funct/rs1 values 000 - 110, and a 32-bit format (RES2) for funct/rs1 value 111, which is identical to RES2 but appears to use the next 16 bits together with its own entire 16-bit value as a 32-bit immediate. RES2 in particular appears a highly implausible format, but is so far the only format which matches the observed architectural state changes. I determined the format by modifying the epilogue bytes of my test function. By default the instruction is immediately followed by the bytes `0x82 0x80` for `ret` to return from the function; noticing that the value moved to each register by RES2 instructions appeared to be the instruction and `0x82 0x80` together read as a 32-bit value, I confirmed this by inserting other 16-bit values before the `ret` bytes such as `0x00 0x01` (nop) and found the RES2 values changed correspondingly, with RES1 instructions unaffected. However, surprisingly the decoder still appears to treat RES2 as a 16-bit instruction: if the following 16-bit value is a faulting compressed instruction (such as the defined illegal value `0x00 0x00`) the RES2 instructions execute before faulting with an illegal instruction exception on this next 16-bit instruction (except RES2 instructions which are known to fault or not return; the consistent behaviour of these demonstrates that the RES2 instructions do indeed execute).

I have not yet managed to determine how the source registers/addresses are encoded due to the limited data available. As the performance counters are unreliable, monitoring register state changes is unfortunately the only method available for obtaining data about the instruction's functionality. Only the values loaded by the RES2 instructions ever vary (when the 16 bits following the instruction are modified) and the segmentation faults and 0 values are uninformative, as a variety of sources could produce

these. I also cannot locate the source of the value 139264 (it is not present in any of the registers, for example). As would be expected, modifying the thread pointer `tp` is typically catastrophic for the child test process, and in particular the `tp` encodings `1000000101xxxx000` and `1001000101xxxx00` for xxxx from 0000-1001 trigger kernel oops (`"unhandled signal 11 code 0x1 at 0xfffffffffffff8f0 in ld-2.28.so[2000000000+1c000]"`) which demonstrate that for at least the 101 funct/rs1 value bits 5:2 *are* relevant to instruction functionality. However, as the address of the segmentation fault does not vary, it is unclear what exactly they encode.

The only HiFive Unleashed erratum which might be potentially relevant to this behaviour is ROCK-2: "the high 24-bits of virtual and physical addresses are sometimes unchecked" [117]. However, this seems an unlikely explanation for the instruction behaviour of RES1 and RES2, because both appear to be interpreted as 16-bit instructions, so surely could not encode a long enough address to be affected (with the possible exception of the RES2 format). I have also confirmed experimentally that the instruction bytes are ordered correctly in memory (i.e. these results are not erroneously due to confusion between little-endian and big-endian ordering or other endianness variants).

The question remains: why do these bizarre instructions exist? The distinct formats suggest the instructions may have been deliberately encoded, as two entirely distinct instruction formats are likely to both require dedicated decoding logic. They may perhaps be debug instructions which are either intended for internal SiFive use only, or were found to be unreliable after fabrication and left on the die but not documented. The only documented debug interface requires JTAG input and so should not create any undocumented instructions in regular instruction decoding. It seems unlikely that such a large number of undocumented instructions could have been missed in verification unless the absence of undocumented instructions was not verified whatsoever, which would be a surprising oversight. I have found no trace of the RES1 or RES2 encodings in publicly-available RISC-V verification frameworks such as riscv-formal; however, the author of this framework does note that some early RISC-V cores implemented reserved opcodes incorrectly [138].

```
#define _GNU_SOURCE
/* Note: include stdio.h, stdlib.h, setjmp.h, signal.h,
sys/mman.h, unistd.h, string.h, sys/wait.h, inttypes.h, sched.h */

unsigned char execInstruction[6];
volatile sig_atomic_t lastSig = 0;
sigjmp_buf buf;
struct sigaction handler;

void signalHandler(int sig, siginfo_t* siginfo, void* context){
  lastSig = sig;
  siglongjmp(buf, 1);
}

int main(){
  cpu_set_t mask;
  CPU_ZERO(&mask);
  CPU_SET(0, &mask);
  sched_setaffinity(0, sizeof(cpu_set_t), &mask);

  memset(&handler, 0, sizeof(handler));
  handler.sa_flags = SA_SIGINFO;
  handler.sa_sigaction = signalHandler;
  sigaction(SIGILL, &handler, NULL); sigaction(SIGFPE, &handler, NULL);
  sigaction(SIGSEGV, &handler, NULL); sigaction(SIGBUS, &handler, NULL);
  sigaction(SIGTRAP, &handler, NULL); signal(SIGCHLD, SIG_IGN);

  size_t pagesize = sysconf(_SC_PAGESIZE);
  uintptr_t pagestart = ((uintptr_t) &execInstruction) & -pagesize;
  mprotect((void*) pagestart, pagesize, PROT_READ|PROT_WRITE|PROT_EXEC);

  //set instruction bytes in execInstruction[0:3], LSB first
  execInstruction[4] = 0x82; execInstruction[5] = 0x80;  //C.ret
  asm volatile("fence; fence.i");

  //forking only necessary for stability if testing multiple instructions
  pid_t pid = fork();
  if(pid == 0) {                    //we are the child process
    if(!sigsetjmp (buf, 1)){
      lastSig = 0;
      ((void(*)())execInstruction)();
      printf("Instruction ran\n");
    }
    else if(lastSig == 4) printf("Illegal opcode\n");
    else printf("Exception %d\n", lastSig);
    fflush(stdout);
    exit(0);
  }
  else waitpid(pid, NULL, 0);       //parent waits for child
  return 0;
}
```

Listing 3.1: Testing instructions on the HiFive Unleashed in Linux.

```
#include <stdio.h>
#include <metal/cpu.h>
#include <setjmp.h>

unsigned char execInstruction[6];
volatile int lastSig = 0;
jmp_buf buf;

void exception_handler(struct metal_cpu *cpu, int ecode) {
  lastSig = ecode;
  longjmp(buf, 1);
}

int main(){
  struct metal_cpu *cpu0 = metal_cpu_get(0);
  struct metal_interrupt *cpu_int = metal_cpu_interrupt_controller(cpu0);
  metal_interrupt_init(cpu_int);
  for(int i=0; i<12; i++){
    metal_cpu_exception_register(cpu0, 0, exception_handler);
  }

  //set instruction bytes in execInstruction[0:3], LSB first
  execInstruction[4] = 0x82; execInstruction[5] = 0x80;  //C.ret
  asm volatile("fence; fence.i");

  if(!setjmp(buf)){
    lastSig = 0;
    ((void(*)())execInstruction)();
    printf("Instruction ran\n");
  }
  else if(lastSig == 2) printf("Illegal opcode\n");
  else printf("Exception %d\n", lastSig);
  return 0;
}
```

Listing 3.2: Testing instructions on the HiFive1 with the Freedom Metal framework.

Chapter 4

Implementation and Results: x86

"ECCASLURV - Elect CPU Core As Supreme Leader Using Runoff Voting"

@x86instructions, Twitter

This project's contributions on x86 are as follows:

- Significantly improved stability of the established instruction testing technique in OpcodeTester, and implementation of directed and random search.
- A novel instruction search approach conducting a timing attack on the three-byte opcode space.
- Investigation of the instruction decoder: a mapping of the instruction space using decoded lengths, and identification of suspicious #UD decoding behaviour as a CPU bug now resolved in microcode.
- Two novel instruction testing techniques (using TSX RTM and the specpoline construct) which suppress exception generation for significantly improved stability and performance (RTM is 95.8% faster than the established instruction testing technique).
- An experimental investigation of transient execution of #UD-faulting instructions, with results supporting the observation of [1] and establishing that #UD-faulting speculative instructions demonstrate uniform behaviour from the instruction decoders onwards.
- A version 2 release of the OpcodeTester tool incorporating the above contributions; a fork of the Sandsifter tool using the Intel XED disassembler to resolve its most significant flaw of false positives; and a fork of the NanoBench microbenchmarking tool enabling testing of arbitrary machine code and exception handling. All tools have been released as open-source on GitHub [2] [139] [140].

4.1 Undocumented Prefetches

As a baseline, I began by testing all the x86 microarchitectures under test with my OpcodeTester tool from my previous research project (previously only the Broadwell microarchitecture had been tested). This version 1 release of the tool uses Sandsifter to search the instruction space with the tunnelling algorithm, filters the results found through the Intel XED disassembler and re-tests the instructions with support for further analysis and testing in kernel mode. Table 4.1 illustrates the extent of false positives identified by Sandsifter (due to its use of the Capstone disassembler, which has incomprehensive coverage of the ISA); instructions were tested only in user mode as I previously found in [2] that kernel mode testing is not worthwhile (it significantly increases the risk of crashes without noticeably altering the results). The runtimes provided are for the Sandsifter search only, as this is the major factor in the test's speed (OpcodeTester's analysis runs in around a minute, depending on the number of instructions in the Sandsifter logfile). The unknown and approximate values on Bonnell are due to the frequency of program crashes when running Sandsifter scans.

The lengthy test times and high number of false positives in these tests prompted me to port Sandsifter to use XED directly in the hope that this would accelerate testing. I did not integrate XED with Sandsifter's

Microarch	Search	Runtime (hours)	Tested	Valid	Unsupported	Fully Undoc
Broadwell	Tunnel	06:17:27	826627376	1851679	306576	0
Westmere	Tunnel	10:16:32	248430338	540076	89418	0
Bonnell	Tunnel	approx. 17:00:00	>179635398	6361631	119325	0
Bonnell	Random	approx. 08:00:00	unknown	1203382	507	0

Table 4.1: Baseline results with OpcodeTester v1 on the three x86 microarchitectures. Instructions which executed without faulting are identified as valid (Sandsifter false positives), documented but supposedly unsupported on the given microarchitecture, or fully undocumented.

Python command-line interface, as XED provides only a C API and I was reluctant to introduce a dependency on the unmaintained `pyxed` bindings. However, Sandsifter can still be used effectively without this interface (using the injector directly). This port has been published on GitHub as a fork of the main Sandsifter repository [139].

The tunnelling algorithm (and the three random fuzzing tests conducted on Bonnell) found no instructions which are entirely undocumented in the x86 ISA, but did find many `prefetchwt1` instructions (in fact, all of the 'unsupported' instructions identified are prefetches). These had previously been identified in my prior research project [2] and in the Sandsifter project [5]. However, I decided to investigate them more thoroughly as a potential source of side-channel vulnerabilities. `prefetchwt1` (`0f 0d /2`) is the "prefetch vector data into caches with intent to write and T1 hint" instruction. It *may* prefetch data from the location specified in the operand into the second level cache and put it into the 'Exclusive' state. There is no guarantee that it will; "no data movement occurs" if the data is already present in this state in the cache, and prefetches can be "overloaded or ignored by a processor implementation." `prefetchwt1` is supposedly supported only on Intel Xeon Phi CPUs, and both `cpuid` and XED agree it is unsupported on the microarchitectures under test (`cpuid` leaf 7, ECX bit 0 [91, Vol. 2A]). XED's datafiles indicate it as available on the Kaby Lake microarchitecture only [3]. However, the uops.info dataset notes Intel IACA performance values for the instruction on Nehalem, Westmere, Sandy Bridge, Ivy Bridge, and Haswell (1 µop on port 2 or 3 for Haswell) [141]. So it appears to be a 'documented undocumented instruction': supposedly unsupported on the microarchitectures under test, with the reality known to some Intel teams (IACA) and not others (XED). A potential source of the confusion is AMD's implementation of the `prefetch` instructions. They define `0f 0d /0` and `0f 0d /1` only, but note that other destination register values in the ModRM byte will not cause an #UD exception and are defined as aliases to `0f 0d /0` [125], so perhaps Intel has also implemented these aliases for compatibility purposes (and failed to document them). But if `prefetchwt1` has no effect on architectural state, then does it matter that it is undocumented on the microarchitectures under test? Of course! An undocumented instruction which operates on the cache state is an *undocumented side channel*. [99] showed that other prefetch instructions (`prefetchnta` and `prefetcht2`) could be exploited in side-channel attacks to defeat the supervisor mode access prevention (SMAP) and address-space layout randomisation (ASLR) security mechanisms, and they can also be used in cache timing attacks against user-level processes.

Investigating the x86 prefetch instructions highlights the unreliability of x86 documentation. For example, Intel's instruction set reference says that the T2 temporal parameter on a prefetch instruction specifies that data will be prefetched into the last level cache or *"an implementation-specific choice"* and that the amount of data prefetched is implementation-dependent but "a minimum of 32 bytes" [91], whereas Intel's optimisation manual declares that there is "no implementation difference between PrefetchT1/T2 on any microarchitecture"[1] and that prefetches "always fetch 64 bytes of data" [123]. It is unclear why these ambiguities persist in the ISA when Intel are clearly aware that their claims that the behavour is implementation-dependent are incorrect. These distinctions matter when implementing or defending against cache attacks: the amount of data fetched is relevant given that the attack aims to determine whether the victim process has accessed a given address or not, and the last level cache is shared across all cores whereas the second level cache is per-core, so an attacking process using the second-level cache must be on the same core as the victim process. Despite these differences however, cache attacks have been successfully conducted at all cache levels [118], and therefore any undocumented instruction operating on the cache is a potential vulnerability.

I adapted the NanoBench microbenchmarking tool to determine whether the undocumented encodings

[1]The T1 and T2 temporal distinctions are redundant, with both filling L2 and LLC but not L1 on Nehalem and newer microarchitectures except Xeon CPUs [123]

actually behave as prefetches. The original tool is designed to be used via the command-line; I modified it to implement support for exception handling (so that faulting instructions can also be benchmarked, albeit with overhead from the OS- and user-level exception handling) and to support its usage as a shared library, then developed a Python script using this shared library to read in and benchmark instructions from a file (such as a parsed Sandsifter or OpcodeTester log). As there are other use cases for this adapted tool beyond undocumented instruction testing, I have released it as a fork of the original repository on GitHub [140]. The main development challenges were understanding how NanoBench produces its benchmarking code and correctly parsing the instructions from a file. NanoBench assembles the provided assembly code (a step which can be skipped in this case), reproduces it many times to increase the measurement window and then adds in the relevant machine code for using the performance counters before and after; when handling exceptions with `sigsetjmp` and `siglongjmp` this 'after' code never runs, so unless this code is isolated and called manually afterwards the performance counters are not stopped and the measurement values overflow. Parsing instructions from a file was surprisingly challenging: for readability, the best way to log instructions is as an ASCII representation of the hex bytes (e.g. `0x90` for `nop`), but such representations frequently contain '00' (particularly instructions produced by tunnelling, as the algorithm aims to keep as many bytes at `0x00` as possible to reduce the risk of memory corruption). '00' is misinterpreted by C string functions as the null byte string terminator and so instructions are only partially parsed. This was the motivation for parsing the file in Python, but even in this setting correctly converting the ASCII to hex and then into a C-compatible ctypes format required testing of many different strategies; ultimately, using the `bytearray.fromhex()` function and converting this to a `ctypes` char array proved effective.

I used the modified tool to compare the behaviour of `prefetchw` and the undocumented `prefetchwt1`, measuring µops retired, µops dispatched onto ports 2 and 3 (the ports used by documented prefetches, according to the uops.info dataset [141]), and µops triggering memory loads with L1, L2, or L3 hits or misses (with each cache statistic measured separately). I found no discernable difference between the instructions - or indeed between `prefetchw` and the other ModR/M values - suggesting that Intel have indeed followed AMD and implemented `0f 0d /2` and /3-7 as aliases of `prefetchw` (other than on the Kaby Lake microarchitecture). They consistently demonstrated a single retired µop, an average of 0.5 µops on both port 2 and port 3 (as the scheduler spread the repeated measurements between the two ports), and a single L1 cache hit.

4.2 Searching the Instruction Space

I improved OpcodeTester's instruction testing framework by making the same changes discussed in Section 3.2 for RISC-V, such as adding a separate stack for the signal handler, using `sigsetjmp` and `siglongjmp`, and forking a different child process for each instruction test. This improved stability sufficiently to conduct a timing attack on the entire space of possible three-byte instructions (see Section 4.2.3), which was previously impossible due to the frequency of crashes [2]. In addition to this search method, I also implemented directed and random search strategies in OpcodeTester, and explored how the tunnelling algorithm can be used to visualise the instruction space and to reverse-engineer instruction decoding behaviour.

4.2.1 Directed and Random Search

Directed search. I manually identified 765 gaps in the Intel and AMD opcode maps to produce an instruction test tailored to encodings most similar to documented encodings. Although this approach appears naive (a manufacturer trying to conceal an instruction would presumably not choose a conspicuous blank in the opcode map for the encoding!), there are precedents of this occurring on x86 (such as `salc`, as discussed in Section 1.3), and if the instruction decoders do have any design flaws then instruction encodings differing by only one or two bits from documented encodings are the most likely to activate the same bitfields in the decoding circuitry. All 765 instructions correctly produced an illegal instruction exception on Broadwell.

Random sampling. Sandsifter's random fuzzing implementation uses the low-entropy pseudo-random number generator (PRNG) function `rand` to create 'random' instructions. Domas notes in the source code that this produces an uneven distribution, and this was apparent in the random Sandsifter tests run on Intel Bonnell. To achieve a more even distribution for better coverage of the instruction space I added support for random fuzzing to OpcodeTester using the PRNG function `arc4random` function from OpenBSD to randomise instruction lengths from 1 to 15 bytes and instruction bytes from 00 to ff. `arc4random` uses the ChaCha20 stream cipher to generate its number stream, which offers significantly higher-quality pseudo-randomness compared to `rand` [142]; the PRNG is also manually reseeded with data from `getentropy` every 10 instructions. I

then conducted random fuzzing with OpcodeTester on Broadwell and Bonnell. These tests produced some fascinating results, identifying hundreds of instructions beginning with `c5` (on Broadwell) and `f0` (on both microarchitectures) as potentially undocumented instructions which displayed erratic exception behaviour and occasionally executed. Unfortunately, after considerable investigation these were found to be false positives, but this did help identify an insidious bug in the test mechanism used both by OpcodeTester and by NanoBench; see Section 4.3.1 for discussion.

4.2.2 Mapping Instruction Decoding

As described in Section 2.3, Sandsifter's tunnelling algorithm uses decoded instruction lengths to guide its search through the instruction space. Using Sandsifter's page-fault technique, instruction lengths can be determined even for faulting instructions, which may in fact be valid instructions in a different CPU mode or at a higher privilege level (than the testing context of ring 3 and 64-bit mode). However, Sandsifter incorrectly assumes that such instructions will always fault with a #GP exception [5], when in fact both Intel's and AMD's documentation state that such an instruction may also fault with an #UD exception: despite its 'UnDefined' memonic, the exception indicates merely that the instruction is *currently*, but not necessarily *permanently*, invalid. Examples of this include instructions invalid in 64-bit mode such as `salc` and `popa`, `sysenter` and `sysexit` (when in long mode), and `rsm` (when not in SMM) [91] [125]. The behaviour of the decoder is therefore potentially far more informative than monitoring exception behaviour, as we cannot assume the distinction that #GP-faulting instructions are valid under other conditions and #UD-faulting instructions are completely unimplemented. To investigate this, I explored how the outputs of tunnelling could be graphed as a 'map' of the decoded instruction space. Even when reduced by tunnelling, the size of the instruction space poses significant challenges for visualisation; the quantity of data (approx. 5GB for each microarchitecture) causes out-of-memory errors with `matplotlib` unless data structures and memory usage are very carefully managed, and the vast range of instruction values means that individual instruction anomalies are easily missed when scaled to fit all instructions into a single graph.

Figures 4.1 and 4.2 are the outcome of this investigation, mapping 1-byte opcodes across the instruction space on Broadwell and Bonnell respectively (note that 2- and 3-byte opcode instructions have also been plotted, but are visually indistinguishable, as their 0f prefix compresses them all into the region marked in red on Figure 4.1). Whilst some information is inevitably lost at this scale, the 'maps' are remarkably informative. They facilitate visual comparison of the two microarchitectures' decoders, demonstrating that their decoding behaviour is almost identical despite their numerous other implementation differences. The key differences visible are that the `a0` encoding of `mov` takes a 64-bit operand on Broadwell (note: AMD's documentation is far clearer regarding this than Intel's [125] [91]) vs. a 32-bit operand on Bonnell (producing the bar of length 9 on the Broadwell graph and length 5 on the Bonnell graph), and that there is a denser concentration of length 7 instructions around 62/63 (marked as `bound` on the Broadwell graph) on Bonnell. The denser instructions around `bound` are due to the `arpl` instruction, which is not encodable in 64-bit mode; opcode 63 is repurposed to implement the `movsxd` instruction. Given the numerous implementation differences between the two microarchitectures and the fact that Bonnell supports the 32-bit version of the ISA rather than the 64-bit version supported by Broadwell (as tested in 64-bit mode), the similarity is quite surprising and highlights the advantages of examining decoding behaviour rather than exception behaviour. Rather than discovering only the instructions we are currently allowed to execute, we can discover other exceptions potentially executable only in other modes or under certain architectural conditions. For example, 32-bit mode instructions unavailable in 64-bit mode (which are not repurposed as different instructions and so are entirely invalid) can clearly be seen in Figure 4.1, such as `push/pop es`, `push cs`, `push/pop ss`, `push/pop ds`, `daa`, `das`, and `aad` (marked as 06/07, 0e, 16/17, 1e/1f, 27, 2f, and 37 respectively).

In addition to facilitating comparison of decoding across microarchitectures, this approach facilitates visual identification of decoding anomalies. In particular, there are 505,123 decoded instructions beginning with the opcode c7 which are entirely undocumented (the penultimate bar of length 11, and along with its neighbouring instructions). The opcode c7 /0 is documented as one of the many `mov` encodings, but all other ModR/M register field values are only defined if preceded by an EVEX prefix (/2 and /6-7) or by 0f (/1 and /3-5). These encodings are particularly intriguing because 67,584 of them also have the longest instruction length possible with a one-byte opcode without using prefixes (11), which suggests they comprise an opcode, ModR/M byte, SIB byte and 64-bits of displacement and/or immediate bytes. (Note: `c7 f8` is also documented as `xbegin`, which is not included in these encodings.) I hypothesised that the decoder might be ignoring the absence of an EVEX prefix when determining instruction length for /2 and /6-7 encodings, as the fact that Broadwell

and many other microarchitectures can decode an unlimited number of (non- length-changing) prefixes per cycle suggests that prefix decoding may be implemented separately to other decoding logic [130]. It seemed unlikely that the decoder might ignore the absence of 0f for /1 and /3-5 because 0f is part of the opcode for these encodings rather than a prefix. However, this proved incorrect: although the length decoded varies (6, 7, 8, 10, or 11 bytes) it is not dependent on the ModR/M reg field or any other aspect of the ModR/M byte, so the variations in decoded lengths are clearly an artefact of a different aspect of the decoding logic. Whilst the decoding behaviour of these 505,123 instructions remains a mystery, this example demonstrates how mapping the instruction space can facilitate reverse-engineering of decoding.

4.2.3 Three-Byte Timing Attack

[106] presented a methodology for identifying undocumented MSRs via a timing attack. Theoretically, an MSR which is implemented but protected by privilege checks or a state condition such as a register password might take longer to access than an unimplemented register (due to the logic overhead of the privilege/state check), or alternatively might be faster if recovery from an unimplemented MSR access is more complex than the logic required for the privilege check. Regardless of whether it is faster or slower, it should produce a timing anomaly because the access will necessarily activate different logic in the CPU. This same timing attack methodology can also be applied to instruction testing. The mapping of the decoder in Section 4.2.2 demonstrates that the decoder still correctly decodes instructions which are invalid in the current mode (rather than having entirely separate decoding circuitry for each mode); at which stage in the pipeline, then, is an instruction actually identified as illegal? Could an instruction have effects prior to exception generation (such as a register password check for a debug instruction, as with [96]) which would introduce detectable timing anomalies? To investigate this I conducted a timing attack on all possible illegal 3-byte encodings on Broadwell and Bonnell. The results shown in Figures 4.3 and 4.4 strikingly demonstrate the execution differences between the 32-bit and 64-bit versions of the ISA: whilst decoding behaviour may be almost identical for each instruction, timing is certainly not!

The timings for each instruction were produced by averaging 10,000 measurements using the combination of `cpuid` and `rdtscp` (or `rdtsc` on Bonnell) recommend by Intel in [143] to serialise the measurements and instruction testing (excluding steps only possible in kernel-mode such as disabling interrupts). Before averaging, outliers were removed: the first and second tests of an instruction are always slower (presumably due to the instruction not yet being present in the instruction cache) and so are skipped, as are any subsequent measurements >1000 cycles higher than the current average (as these are presumed to indicate external events such as the process being scheduled or the CPU entering and returning from SMM). The timings include the overhead of OS- and user-level exception handling (which cannot be entirely eliminated when testing in user mode) and of reading and storing the values of all architectural registers to detect any register state change. Note that the seemingly solid bar of instructions with timings below 4000 cycles in Figure 4.3 is an artefact of the sheer number of illegal instructions to test on 64-bit vs. 32-bit x86 (1,676,054 vs. 478,795, as many one-byte encodings are valid only in the 32-bit version of the ISA); there are actually legal instructions within this space which were not tested and at a larger scale would appear as gaps. Due to the performance differences between the two microarchitectures, the timings are at very different scales; the time taken for a typical illegal instruction on Bonnell (approx. 20000 cycles) exceeds the very highest anomalies on Broadwell, where a typical time is approximately 3600 cycles; these reliable 'baselines' are the overhead produced by generation of an #UD exception, with timing anomalies higher than this indicating that an additional event occurred in the pipeline. Red lines indicate instructions faulting with a segmentation fault rather than an illegal instruction exception; it is apparent that these are faster on average than #UD exceptions on Bonnell, suggesting that #GP exception handling may have a substantially different implementation in the CPU. However, this could instead indicate that the front-end detects #GP instructions faster: it is not apparent from these timings at which stage in the pipeline delays occur.

On Broadwell, the vast majority of timing anomalies are concentrated between `00 00 00` and `3f ff ff`. This is the space most densely populated by instructions valid in 32-bit mode only, and examining the instructions with the 15 highest average times suggests that this is indeed the cause of the anomalies, as they are all sequences containing at least one valid 32-bit mode instruction. The few anomalies outside this space are also at least partially valid 32-bit sequences. It appears that the presence of a single valid 32-bit instruction at the start of the sequence is sufficient to produce an anomaly, even if the sequence is not valid in its entirety. There were however some exceptions to this which are discussed further in Section 4.3.1.

On Bonnell these 32-bit mode anomalies do not occur, as the instructions are valid and are therefore not tested. Instead, the anomalies found appear to demonstrate the limitations of Bonnell's two instruction decoders. Each decoder can decode a maximum of one instruction per cycle [123]; however, it appears that in this test instructions are decoded more slowly with initially only the first byte being processed, as the top anomalies are instructions which based on their length and first byte alone *could* be valid instructions, but are rendered invalid by their second or third bytes. For example `fe 22 c3`, which has the highest average time of 35,166 cycles, contains no valid instructions but with a different second byte such as `00` would be a valid instruction sequence (`inc` and `ret`). This suggests the decoders may be initially checking the first byte alone before checking later bytes in subsequent cycles.

These observed behaviours may be an artefact of the test mechanism (for example, the instruction under test might by chance be lying across the boundary of two instruction fetches on Bonnell) rather than indicative of the behaviour of the microarchitectures' decoders in general. However, as the behaviour is reproducible it demonstrates how a timing attack could detect undocumented instructions, which would be indicated by unexplainable timing anomalies. A slower instruction on Broadwell which contains no valid 32-bit instructions might indicate an undocumented instruction only valid in another mode/state (such as an SMM-only instruction). Bonnell's behaviour is unfortunately less useful, as there is no first byte which is guaranteed to be invalid for any documented instruction, but anomalies with uncommon first bytes would suggest a potentially undocumented instruction worthy of further investigation.

Westmere 'HCF' Instruction

Whilst carrying out the 3-byte timing attack on Intel Westmere I found an instruction I suspected to be a HCF: `01 52 96` (a documented instruction, `add dword ptr [rdx-0x6a],edx`). When executed within the timing attack code it appeared to hang the system, although with constant hard drive activity; rebooting subsequently required removing the laptop's battery. After reproducing this several times I tested the instruction within a minimal test program outside of the timing attack code and found that under these conditions it executed normally; in order for the crash to occur, the test function must be run in a child process with a parent process waiting for its completion. I could not reproduce the crash on Broadwell (executing identical assembly code) but did note that the program took considerable time to complete, showing high memory usage and high CPU usage by `kswapd0`, an OS process which manages virtual memory. This led me to suspect that the instruction was somehow causing significant memory allocation which led to the system running out of memory on Westmere. Subsequently I found that that the system remained somewhat usable via the TTY console; the system was running out of memory after the test program had allocated more than 12GB of virtual memory, explaining the constant hard-drive activity. I suspect this is a bug in the specific kernel version's memory management or perhaps in a driver, but have been unable to investigate further as the Westmere platform suffered hardware failure soon afterward.

4.3 Instruction Testing Techniques

4.3.1 Pitfalls of Variable-Length Instructions

As described in Section 3.4, on RISC-V the Compressed extension complicates testing for undocumented instructions, as it is not obvious whether an executed 'instruction' is a single 32-bit instruction or two 16-bit instructions. The variable-length encoding of the x86 ISA increases this challenge enormously, as an instruction can have any length from 1 to 15 bytes. During the random fuzzing and timing attack tests on Broadwell and Bonnell I encountered bizarre instruction behaviour ultimately caused by an insidious flaw in my test mechanism. The ambiguity of variable-length instruction encoding makes a flaw such as this very challenging to detect because in the vast majority of cases testing appears to work correctly, which is why I believe this issue is worth highlighting for any future implementation of self-modifying code on x86.

The behaviour I observed was that `f0f6` often (but not always) ran on both microarchitectures, as did a variety of undocumented instructions beginning `c5` on Broadwell. Others even appeared to produce architectural state changes (Listing 4.1), and on Bonnell behaviour was even stranger. Data corruption of the instruction bytes appeared to be occurring, as shown in Listing 4.1 with the sudden jump from `0f0d13` to `c10d14` and `d70d15` before returning to `0f0d16`. Note that the error codes in the Listing are those returned by the XED disassembler for each instruction, and the avg/min/max values are cycle counts.

```
c50a  RAN 0 GENERAL_ERROR avg 152 min 20 max 638
f0f6  RAN 0 BAD_LOCK_PREFIX avg 48 min 47 max 344
0f0d10  RAN 0 INVALID_FOR_CHIP avg 19 min 18 max 145
c57b  RAN 0 GENERAL_ERROR avg 19 min 18 max 142
c5f5  RAN 0 GENERAL_ERROR avg 19 min 18 max 142
State change: rcx: 0 2451587878, rflags: 582 646

0f0d13  RAN 0 INVALID_FOR_CHIP avg 452 min 372 max 42708
c10d14  EXCPT 11 INVALID_FOR_CHIP avg 9616 min 9432 max 94356
d70d15  EXCPT 11 INVALID_FOR_CHIP avg 9637 min 9444 max 110472
0f0d16  RAN 0 INVALID_FOR_CHIP avg 454 min 372 max 2412
```

Listing 4.1: False positives from the timing attack on Intel Broadwell (above) and 'data corruption' in the timing attack on Intel Bonnell (below).

After considerable debugging attempting to reliably reproduce the behaviour, I found a bug in my code: the test array was not reset after each instruction test, so in a small number of cases the `ret` byte following the test code would be interpreted as part of a longer multi-byte instruction rather than as a `ret`, and execution would continue using previously-tested bytes and even beyond the limit of the test array until an exception occurred (often coincidentally producing VEX-encoded instructions if the instruction began with `c4` or `c5`, hence the frequency of strange behaviour with these instructions). Overwriting the test bytes with zeroes at the start of each test iteration appeared to have entirely resolved this issue, until the 'data corruption' in the log files appeared again on Intel Bonnell. The bug had in fact not been resolved at all; it had merely been made even more infrequent and insidious. The fundamental problem is that x86's instruction encoding is so ambiguous that there is no instruction which can be inserted after the tested instruction to guarantee a return from the test mechanism (either via a jump, return, or triggering an exception); there is always a slight risk that the instruction's bytes will be interpreted as part of a longer instruction. The `int 3` instruction (`cc`, which produces a debug exception) seems the most plausible option, given that it is specifically designed to be inserted into code for debugging. However, in the three-byte timing attack 122,884 encodings featuring a `cc` byte are tested, none of which trigger a debug exception on Broadwell or Bonnell, so this instruction too cannot be relied upon. There is also no byte pattern which can be used to safely overwrite the test bytes, as even zero bytes can encode add instructions (`00 00` is `add byte ptr [rax], al`). This is a crucial difference from RISC-V where 16- or 32-bit patterns of zeroes are defined as illegal.

Whilst no reliable mechanism exists which can be inserted *after* the tested instruction, setting the trap flag before the instruction is executed guarantees that only a single instruction will run before the CPU generates a debug exception. To set it, the relevant machine code[2] must be inserted directly into the test array before the instruction under test; setting the trap flag before calling the test function will trap on the function call, and I found that Linux's handling of the trap flag prevents it being successfully re-enabled in a user mode signal handler on each exception. Exceptions due to invalid encodings, such as #UD and #GP, take precedence over the debug exception and so the exception behaviour of each instruction can still be determined. This is the mechanism used by Sandsifter, and it is important to not overlook this as a mere design choice: it is an *essential prerequisite* for accurate results due to the inherent ambiguity introduced by variable-length encoding. Unfortunately, handling the debug exception adds substantial overhead to any performance counter measurements or timings, which makes this mechanism unsuitable for attempting to infer instruction functionality with performance counter measurements or for microbenchmarking as in the NanoBench tool, which still uses `ret` after the tested instruction. This may explain some of the variance in instruction latency observed in production of the uops.info dataset with NanoBench [141], and raises some concern about its accuracy; however, it appears that in the absence of any other reliable mechanism a trade-off in accuracy is necessary to conduct any microbenchmarking or timing of instructions.

4.3.2 TSX RTM

As discussed in Section 2.3, Intel Transactional Synchronization Extensions (TSX) can be used for exception suppression. If code faults within a Restricted Transactional Memory (RTM) transaction, not only will the exception be suppressed but also all modifications to architectural state within the transaction will be rolled

[2] `9c 48 81 0c 24 00 01 00 00 9d` for `pushfq; orq $0x100,(%rsp); popfq` on 64-bit - remove the REX 48 byte for 32-bit.

```
asm volatile ("xbegin ABORT");
((void(*)())execInstruction)();
asm volatile ("xend; ABORT:");
asm volatile("mov %%eax, %0;" : "=r" (abortCode) : :);
if(abortCode == 16) printf("Instruction ran\n"); //reached INT3
else printf("Instruction faulted\n"); //exception before INT3
```

Listing 4.2: Using TSX RTM instructions to test instructions and suppress exceptions.

back. This provides a compelling alternative to the existing instruction testing framework: the state rollback prevents memory or control flow corruption (restoring the entire architectural state, unlike the limited execution context restored by `siglongjmp`). Listing 4.2 provides a minimal code example: if an exception occurs between `xbegin` and `xend`, the transaction is aborted, architectural state is rolled back, and control flow is transferred to the `ABORT` label. Exception suppression is of particular value for kernel-mode testing: the travails of kernel-mode exception handling [2] can be completely avoided with this method. Using RTM is on average 95.8% faster than the initial framework when testing the faulting `ud2` instruction over 1,000,000 repetitions; as the exception is never delivered to the OS, the overhead of OS- and user-level exception handling is removed entirely. This astonishing performance improvement could enable significantly improved coverage of the search space to be achieved.

However, this approach also has major disadvantages. The trap flag cannot be used (so we must face the potential instruction misinterpretation described above); we cannot distinguish between different exception types, beyond the categories 'no exception', 'debug exception' and 'other exception'; hardware performance monitoring counters cannot be used to infer an instruction's behaviour; and there are a range of valid instructions which cause an immediate abort and so will appear invalid (such as privileged instructions, x87 instructions, and MMX instructions [123]).

Setting the trap flag within the transaction causes an abort, and similarly if we set it before the transaction it aborts immediately with a debug exception. Setting bits 11 and 15 of `IA32_DEBUGCTL MSR` enables advanced RTM debugging [91], but all this allows us to do is trap and loop back to the start of a transaction when the first debug exception occurs, and as we must set the trap flag before the start of the transaction, the first debug exception will always be before the instruction is tested (it occurs immediately after `xbegin`). To somewhat mitigate this issue, we can add an `int 3` instruction byte to the unsigned character array after the instruction bytes to be tested. As discussed above, this is an instruction specifically designed for setting breakpoints at arbitrary code locations, so the instruction decoder is presumably optimised to detect this as a one-byte instruction in most cases, but is not guaranteed to do so. We can easily distinguish between the debug exception generated by `int 3` and the other exceptions caused beforehand if the tested instruction faults, as the abort code for the transaction is 16 for a debug exception and 0 for other exceptions. (Note: for each execution of the program, an error of 0 is returned the first time the instruction array is executed within a transaction; I am unsure why this occurs, as whilst setting the Accessed and Dirty bits of a page can cause a TSX abort, these should already be set for the test array.)

Finally, the Broadwell microarchitecture under test also has an erratum concerning TSX: *"Under a complex set of internal timing conditions and system events, software using the Intel TSX (Transactional Synchronization Extensions) instructions may behave unpredictably"* [87]. This is so vague that it is difficult to verify whether this 'unpredictable' behaviour is occurring, but the error code behaviour certainly does seem to be erratic. In testing, valid, non-aborting instructions generally produce the expected error code of 16, incorrectly return 0 at a rate of approximately 30-70 incorrect return codes per 10 million. Whilst this does reduce the reliability of the results, this is still a low error rate of $<= 0.0007\%$. It would be worthwhile to investigate this further on alternative TSX-supporting microarchitectures to test whether this is caused by the erratum.

4.3.3 Specpoline

[144] initially described repurposing the speculative loop of a retpoline (see Section 2.3) for microbenchmarking, coining it the *specpoline*, and [68] subsequently used the mechanism to leak FPU register state. Akin to TSX, the specpoline can also be used to suppress exceptions from instruction testing, but adds an additional layer of indirection by only ever executing the instruction speculatively. Placing the instruction under test in the

```
asm volatile("push %1; \n\
  call trampoline; \n\
  speculoop: \n\
  call %2; \n\
  jmp speculoop; \n\
  trampoline: \n\
  mov %%rsi, %0 \n\
  mfence; \n\
  lea rsp, [rsp+8] \n\
  ret" : "=m"(slug) : "r"(&&escape), "r"(&execInstruction) :);
escape:
```

Listing 4.3: Using the modified specpoline within C code (GCC inline assembly, Intel syntax). A second indirect call is added to automate instruction testing, along with a 'slug' to slow non-speculative execution.

speculative loop (or speculoop) of the retpoline (using `.byte` to insert arbitrary instruction bytes into the assembly) forces the processor to speculatively execute the instruction many times before it identifies that it has mispredicted the branch target. Such repeated execution amplifies any transient execution effects so that they are more noticeable. The specpoline is a valuable complement to Intel TSX as it can be used to test instructions unsupported by RTM (which always cause transactional aborts) without abandoning the stability and speed benefits of exception suppression. Unfortunately, instruction testing cannot be automated with the retpoline shown in Listing 2.1 in Section 2.3.1 because the byte(s) inserted into the assembly must be immediate rather than variable values. In order to change the bytes at runtime, we must add yet another layer of indirection: the speculoop itself makes an indirect call to the array of test bytes. This adds considerable speculation overhead; however, if the the trampoline is modified so that non-speculative execution is much slower then there is sufficient time for speculative execution of the instruction. (Speculation overhead was determined by comparing the ratios of µops issued to µops retired; see Section 4.4.2 for further discussion of specpoline results.) Listing 4.3 demonstrates how this stall can be achieved by storing the stack pointer to an irrelevant address in memory (the `slug` variable) and calling `mfence` to force in-order execution to stall until this write is complete. Unlike when testing with TSX RTM transactions, speculative execution does not require an initial, non-speculative instruction test (either within or without a transaction) in order to succeed in transient execution.

4.4 Investigating #UD Instructions

4.4.1 Decoding Bug

In my prior research project I found undocumented exception behaviour on Broadwell when testing UD-faulting instructions in ring 0 on Ubuntu 17.10, 17.04 and 16.04 [2]. When instructions longer than 4 bytes were tested within a kernel driver, the return instruction pointer provided to the die notifier was two bytes into the instruction, contradicting Intel's documentation that all trap and fault exceptions are guaranteed to be reported on an instruction boundary. This behaviour reliably occurred across hundreds of thousands of different invalid instructions, excluding the possibility that the first two bytes formed a valid instruction by chance which was then executed before the exception. This suggested mishandling of the return instruction pointer by the the Linux kernel or the CPU itself, or perhaps even partial execution of the invalid instruction - could this perhaps indicate that these were undocumented instructions which could be valid given certain architectural state? I was unable to determine the cause of this behaviour and so in this project I planned to investigate further. It is a serious concern if the CPU returns the wrong instruction pointer to resume execution at after a fault exception such as #UD, as this could potentially be exploited to crash the system or manipulate control flow.

To my surprise I found that it is no longer possible to replicate this behaviour on Intel Broadwell. I initially suspected that this was due to small changes in handling of the return address in the Linux kernel in traps.c [145]. However, the corrected behaviour occurs not only in Ubuntu 18.04 (kernel 4.15.0-46) but also on a VM (Ubuntu 17.04, kernel 4.13.0-43) and a live USB install (Ubuntu 16.04, kernel 4.13.0-43) which had both previously exhibited this behaviour and had not been updated since the previous project. The change is not due to updated microcode loaded by Linux, as this is unchanged across the Ubuntu versions (revision 0x2b, 2018-03-22). This appears to rule out the possibility that the behaviour was caused by the operating

system. However, a major BIOS update was conducted after the research project (from Dell Inc. Latitude E7450/06HN6G BIOS A09 to BIOS A19) which installed multiple microcode updates and patches for Intel ME and SMM. Due to the risks of a BIOS downgrade I was unable to attempt this during the project to confirm the BIOS update as the cause, but it appears to be the only remaining possibility, suggesting the behaviour was indeed a bug which was identified by Intel and patched in microcode. This highlights both the benefits and dangers of microcode: it is positive that such a serious hardware bug can be fixed entirely in microcode, but it is troubling that opaque microcode updates can make such substantial changes without any disclosure. Whilst this particular update likely did not break any software beyond my own as exception handling in a kernel driver is rare, it is foreseeable that other such unannounced changes to exception handling might introduce insidious vulnerabilities into drivers or, in particular, into the operating system's exception handling; as discussed previously the `mov/pop SS` vulnerability highlighted how security-critical correct OS exception handling can be [90].

4.4.2 Transient Execution

> *"In Intel 64 and IA-32 processors that implement out-of-order execution microarchitectures, this exception is not generated until an attempt is made to* ***retire the result of executing an invalid instruction****; that is,* ***decoding and speculatively attempting to execute an invalid opcode*** *does not generate this exception."* [91]

In my previous research project I discovered this suspicious wording in Intel's documentation and identified this as an area for further research. As previously discussed in Section 4.2.2, some instructions which are currently invalid but valid in a different mode/state fault with #UD. In contrast to fully undocumented instructions, these instructions clearly have defined implementations in the CPU pipeline; given recent evidence that instructions do leave transient execution traces before a #GP exception [20], this raises the question of whether such instructions might partially or fully transiently execute prior to generation of the #UD exception. If we can indeed detect transient execution of #UD-faulting instructions, then this enables a new strategy for detecting undocumented instructions: if we can detect transient traces of an instruction which has no documented function in any CPU mode/state, then we can infer that it is implemented in the pipeline and will execute under certain conditions. This would allow us to narrow the search and focus intensely on fuzzing this instruction with a wide range of different states; whilst the complexity of microarchitectural state renders full coverage of the state space impossible, a focused search would have a much higher probability of success than attempting to cover all register states for all possible instructions.

[1] appears to be the only prior research into transient execution of #UD-faulting instructions. This work observed no evidence of transient execution following an #UD exception (or following #DE, #PF, #AC, #SS, or any trap or abort class exceptions), and hypothesised that exceptions triggered in the instruction fetch or decode stages are handled immediately, with the corresponding instruction never entering the reorder buffer. However, this seems unlikely. We already have Intel's statement in their documentation that the CPU may decode and then speculatively attempt to *execute* an invalid opcode, and immediate exception handling at fetch/decode would surely also produce imprecise exceptions inconsistent with x86's architectural guarantees; Intel's documentation is unequivocal that (non-abort) exceptions are not signalled until the retirement stage, which suggests they must enter the reorder buffer [123]. Their explanation for this hypothesis is also based on the incorrect assumption that the CPU cannot transiently execute instructions beyond an illegal instruction because illegal instructions have an *undefined length*. As discussed in [5] and visualised in Section 4.2.2, illegal instructions do have defined lengths, with the decoder consistently stopping after a given number of bytes for each illegal encoding.

One possibility is that the pipeline is stalled as soon as the exception is generated at the fetch or decode stage: all instructions already in the pipeline (including the excepting instruction) continue executing, the results of non-speculative instructions are committed in order and those of speculative instructions are ignored, and no subsequent instructions are allowed to enter the pipeline until the excepting instruction reaches the commit stage and then the exception is allowed to occur. However, given Intel's aggressive performance optimisation it seems unlikely that they would wish to stall the pipeline entirely when an illegal instruction is encountered. Whilst with in-order, non-speculative execution it may be a sensible design choice to stall subsequent instructions as the pipeline will only need to be flushed anyway, with out-of-order and speculative execution this might be completely unnecessary; the exception may never actually commit, for example if the instruction's branch were mispredicted.

A search of the patent literature suggested several possibilities: [146] describes how an exception event might be generated at the reorder buffer, whilst [147] describes a separate exception pipeline parallel to the regular pipeline, and another features an invalid instruction circuit in the decoder which tests for invalid instructions and inserts a preset flow of µops to indicate the #UD exception should be generated [148]. The latter would explain the findings of [1], although of course manufacturers may patent many inventions which are subsequently unused or discontinued in their products, so there is no guarantee that the Broadwell microarchitecture uses any of the mechanisms described. As the potential for transient execution was still unclear I decided to run experiments using the specpoline mechanism to identify which instruction types showed detectable traces of speculative execution (I did not investigate out-of-order execution).

4.4.3 Experimental Implementation

To clarify the exact settings used on Broadwell as the OS, compiler, microcode revisions and enabled Spectre mitigations are highly relevant to the behaviour of the specpoline: the experiments were conducted on Ubuntu 18.04.2 (kernel 4.15.0-47.50-generic) with Intel microcode revision 0x2b, GCC 7.3.0, HyperThreading manually disabled in the BIOS, and the following transient execution mitigations: PTI (Meltdown), PTE inversion and VMX conditional cache flushes (L1TF), user pointer sanitization (Spectre v1), full generic retpoline, IBPB, IBRS_FW (spectre v2), and speculative store bypass disabled via `prctl` and `seccomp`. Of these, only the Spectre v2 mitigations are of direct relevance to the specpoline results. Retpoline is only applied to the kernel and any software specifically compiled to use retpoline; the indirect branch prediction barrier (IBPB) prevents user mode and guest processes controlling predicted branch targets of other processes across context switches; and with the IBRS_FW setting indirect branch restricted speculation is only applied for firmware calls. These mitigations therefore do not prevent use of the specpoline to target indirect branch speculation on a controlled user mode process, provided a context switch does not occur. I also confirmed the specpoline was unaffected by the Spectre v2 mitigations by repeating several of the experiments with the mitigations disabled on kernel boot, producing identical results. Recompiling with the GCC `-mindirect-branch=thunk-extern` compile flag also does not affect the results or the object code disassembly; it appears GCC does not apply retpoline to inline assembly.

With the specpoline mechanism for transient execution of undocumented instructions (as described in Section 4.3.3) in place, hardware performance counters can be used to measure the transient effects of undocumented instructions. Bad speculation is a metric for the quantity of µops which the CPU mispredicted and subsequently had to flush from the pipeline. Intel provide the formula

$$100 * ((uops\_issued.any - uops\_retired.retire\_slots + 4 * int\_misc.recovery\_cycles)/N)$$
$$where\ N = 4 * cpu\_clk\_unhalted.thread$$

for determining the percentage of bad speculation [123]. If there is significant bad speculation, the speculoop must have executed unhindered and therefore we can infer that the instruction did produce transient µops which entered the pipeline. Conversely, if there is low bad speculation then we can infer the instruction did not produce transient µops and therefore exception detection must occur early at the fetch/decode stage. I began testing this on a small set of instructions using the original specpoline without a second indrect call (inserting bytes into the retpoline of Listing 2.1).

As shown in Table 4.4.3, if the speculoop is removed there is only 6.35% bad speculation. With an empty speculoop, this increases to 9.96% as there are many mispredicted speculative instructions issued (for the speculoop's `jmp` instruction as it loops over and over); and with a valid `nop` instruction within the loop, there is yet more bad speculation at 13.48% (with `jmp` and `nop` now being issued repeatedly). Note that the abbreviations used to categorise instructions are: UD64 (#UD faulting when not in SMM, but valid in 64-bit mode); UD32 (only valid in 32-bit mode); UDA (architecturally-defined illegal instruction whose exception occurs after execution, unlike other #UD-faulting instructions); BP (produces a #BP trap); V (valid, does not produce an exception); S (fully or partially serializing); and GP (produces a #GP exception). The results are averaged over 100000 iterations and are reproducible: between separate tests there is very little variance in the counts, particularly the execution port counts.

The first three tests would all have run without faulting had they been executed non-speculatively. With these control results as a baseline, the behaviour of faulting instructions can be compared. The serialising instruction `wbinvd` behaves almost as if there were no speculoop at all, with an identical number of µops issued and retired. This is as expected given that the instruction is serializing. Speculation of other faulting

Instruction	Type	Spec%	Issued	Retired	Recovery	Clocks	Ports
no loop	V	6.35	57	64	10	130	6 12 2 3 7 18 18 2
empty loop	V	9.96	75	64	10	128	6 12 2 3 7 30 30 2
xbegin	V	13.57	93	63	10	129	12 17 3 5 13 30 30 2
nop	V	13.48	93	64	10	128	6 12 2 3 7 30 30 2
pushfq	V	13.46	93	63	10	130	16 19 4 10 7 27 27 6
test	V	13.37	92	63	10	129	11 16 2 3 14 32 32 2
rdrand	V	12.3	86	63	10	128	12 14 2 3 9 26 26 2
cpuid	VS	11.52	62	63	15	128	5 11 2 3 7 26 26 2
popfq	V	7.62	62	63	10	128	7 13 2 3 9 19 19 2
mfence	VS	6.45	56	63	10	128	5 11 3 4 7 20 20 2
hlt	VS	6.59	57	63	10	129	6 11 2 3 7 20 20 2
pause	VS	6.84	58	63	10	128	5 11 2 3 8 23 23 2
wbinvd	GPS	6.45	57	64	10	128	5 12 2 3 7 21 21 2
swapgs	GP	7.63	62	63	10	128	5 12 2 3 9 23 23 2
sysexit	GP	8.2	66	64	10	128	7 14 2 3 10 24 24 2
int 3	BP	7.23	61	64	10	128	7 13 2 3 9 20 20 2
ud2	UDA	6.84	59	64	10	128	6 12 2 3 7 22 22 2
rsm	UD64	6.84	59	64	10	128	5 12 2 3 8 22 22 2
salc	UD32	6.84	59	64	10	128	7 12 2 3 7 20 20 2
aaa	UD32	6.84	59	64	10	128	7 12 2 3 7 20 20 2
aad	UD32	6.84	59	64	10	128	7 12 2 3 7 20 20 2
popa	UD32	7.03	59	63	10	128	7 11 2 3 7 20 20 2

Table 4.2: Specpoline results on Intel Broadwell.

instructions is clearly markedly hindered in comparison to the `nop` or empty loop. There are many possible explanations for this: the exception-handling circuitry may add timing or µop overhead (Broadwell can only handle up to a maximum of 192 µops in flight in the pipeline) to execution of the speculoop, resulting in fewer iterations before the branch misprediction is detected; there may be a physical limit to how many exception-generating µops can be in flight at once; or speculative execution may halt immediately at the first occurrence of the faulting µop. Whilst some #GP-faulting instruction such as `sysexit` demonstrate an increased level of speculation, the #UD-faulting instructions demonstrate a uniformly low level of speculation, supporting the observation of [1] and suggesting the instruction circuit of [148] is implemented on Broadwell. However, these results represent only 6 examples of such instructions; whilst they suggest that #UD-faulting instructions do not routinely produce transient effects, as documented instructions these are perhaps the least likely of all #UD-faulting instructions to demonstrate erroneous transient effects, so there may still exist undocumented instructions with detectable transient effects.

These results also demonstrate the transient behaviour indicative of successful transient execution, which enables it to be identifeid in future testing: significantly higher µops issued than retired (the *characteristic ratio*), as all the speculative µops are mispredicted and therefore flushed without being retired, and increased activity on ports 5 and 6. (The primary branch unit for Broadwell is on port 6, so the increased activity here matches with many iterations of the speculative loop occurring; I am unsure why activity appears to always be increased on port 5, as this port handles ALU and fast LEA operations rather than branching, and the only `lea` instruction produced by the disassembler is within the trampoline.) Knowledge of this behaviour can be used to determine under which conditions speculative execution can occur: for example, I replicated the Meltdown-XD result of [1], with all instructions demonstrating the ratio of #UD instructions when the page of the test array is not executable.

I was concerned that calling and returning from the test function normally (rather than via the trampoline) might be producing unexpected behaviour from the specpoline, as this function call uses the indirect branch predictor rather than the more predictable return stack buffer [126], so I attempted encoding the specpoline directly in machine code. By adding code byte by byte into the test function as used for standard undocumented instruction testing, arbitrary instructions can be tested within the specpoline at runtime without the complication of introducing the indirect branch predictor. Inspired by Intel's top-down analysis method [123]

I monitored additional performance counters in an attempt to isolate the stage of the pipeline at the #UD-faulting instructions were either no longer apparent or indistinguishable from one another. These results further supported the observation of [1] that there are no observable transient effects after decoding: behaviour seems to be suppressed (and uniform, suggesting a universal 'illegal' µop or set of µopsas described in [148]) from the instruction decoders onwards, as the `IDQ.DSB_UOPS`, `IDQ.MITE_UOPS` and `IDQ.MS_UOPS` (which indicate the number of µops coming into the instruction decode queue from the three main instruction sources of the decoded icache, legacy decode pipeline and microcode sequencer) indicate uniform values for a range of #UD instructions (including instructions implemented in other modes, which should demonstrate distinct decoding behaviour if they were decoded normally). There appear to be no significant stalls anywhere in the pipeline, so the suppressed speculation is not due to exception µops overwhelming the pipeline; `RESOURCE_STALLS.RS` and `RESOURCE_STALLS.ROB` report no stalls at the reservation station or reorder buffer, and the number of µops not delivered by the instruction decode queue (`IDQ_UOPS_NOT_DELIVERED.CORE`) is the same as for `nop`, although this figure is higher than that of other valid instructions, perhaps suggesting that both the #UD instructions and `nop` are bypassing the execution core.

To conclude, detection of #UD instructions therefore occurs either at or before the instruction decoders in the Broadwell pipeline, and transient behaviour from this point onwards appears to be uniform across a range of tested #UD-faulting instructions. Hypothesis 3 is therefore almost certainly incorrect; given the uniformity of the µops observed at each stage it seems highly improbable that any meaningful distinction could be determined between them by observing transient execution.

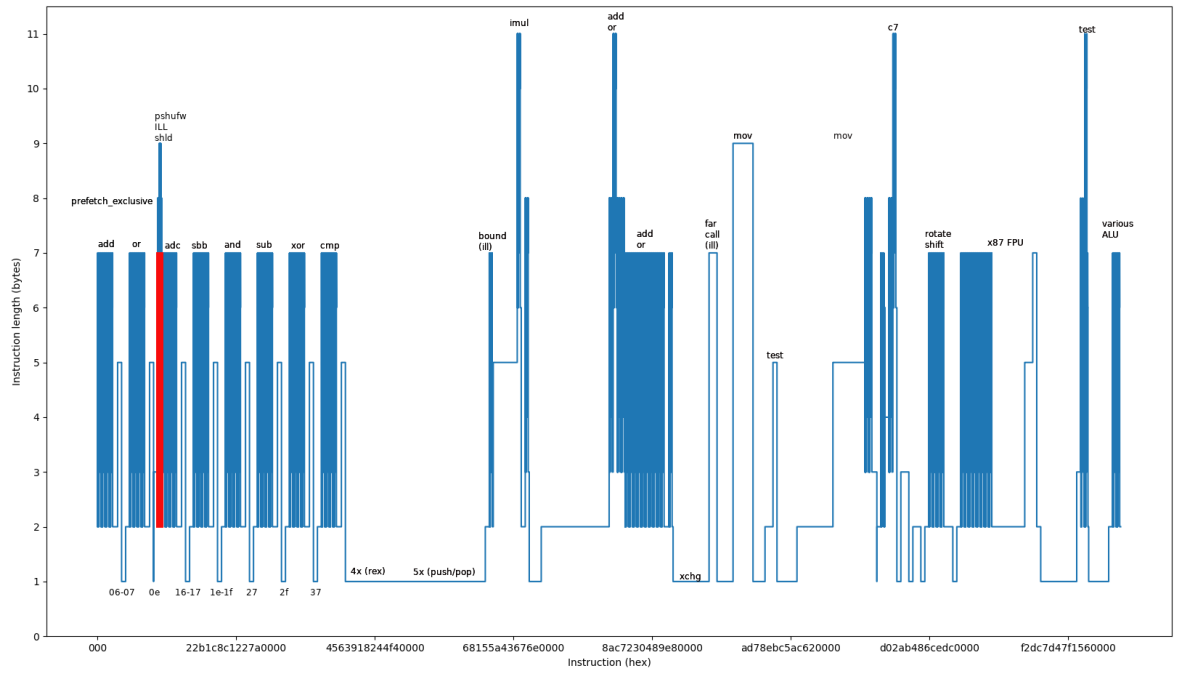


Figure 4.1: Map of one-byte opcode instruction lengths found by the tunnelling algorithm on Intel Broadwell; the labels indicate common mnemonics in each region.

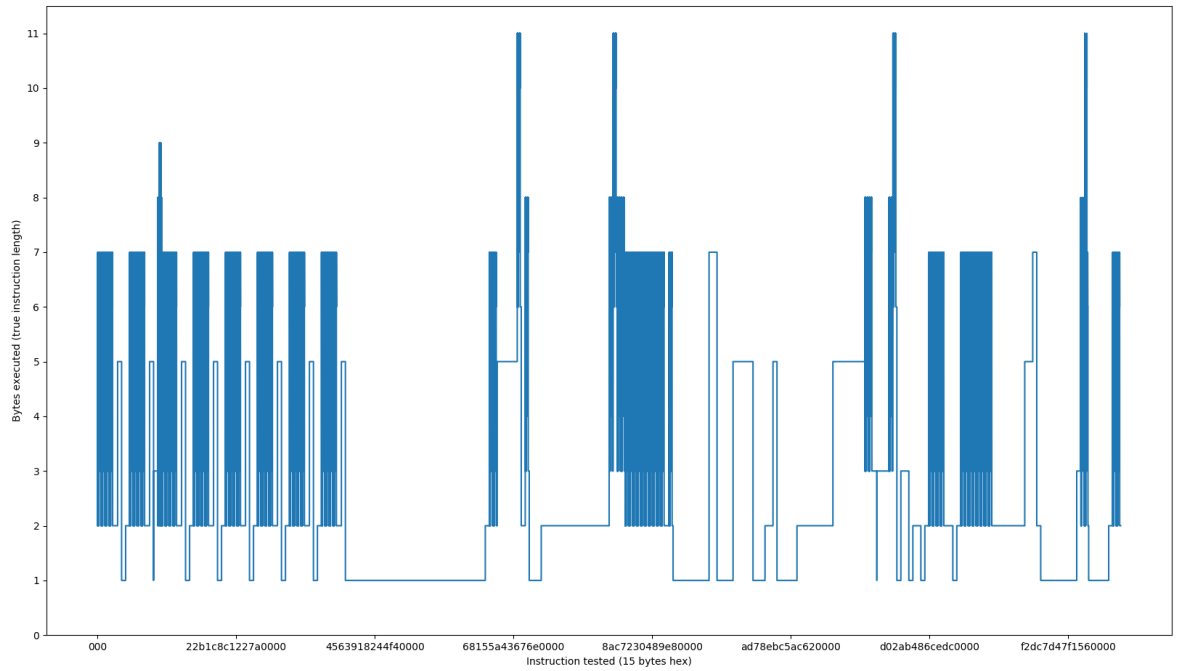


Figure 4.2: Map of one-byte opcode instruction lengths found by the tunnelling algorithm on Intel Bonnell.

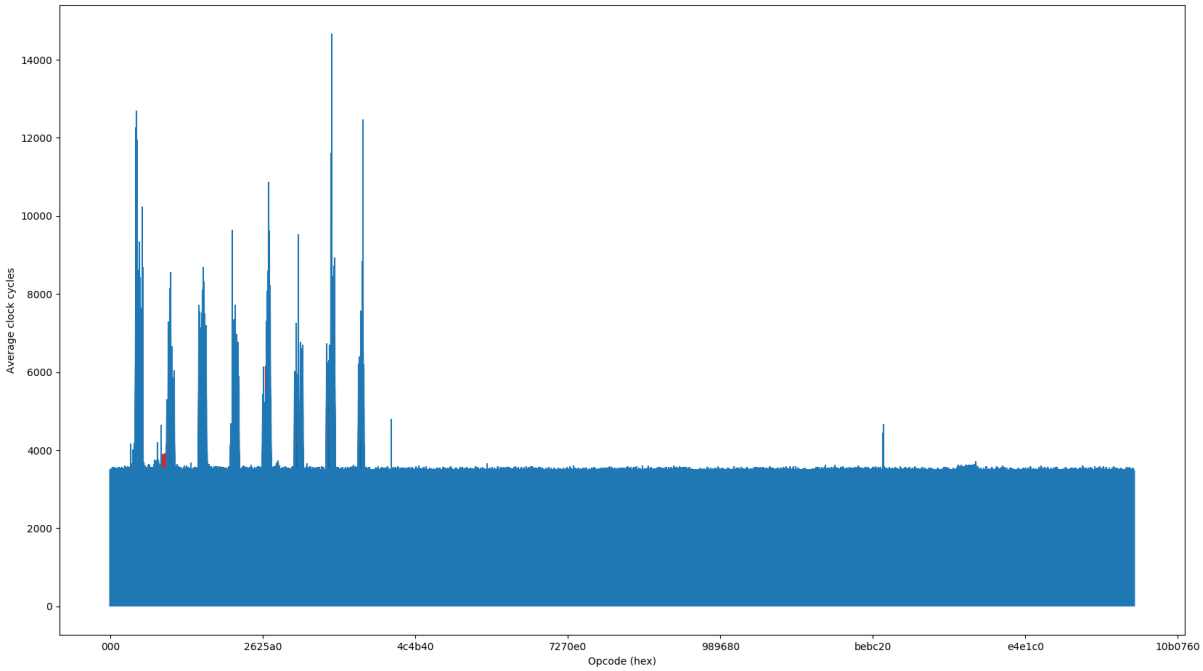


Figure 4.3: Timing attack results on the three-byte encoding space on Intel Broadwell.

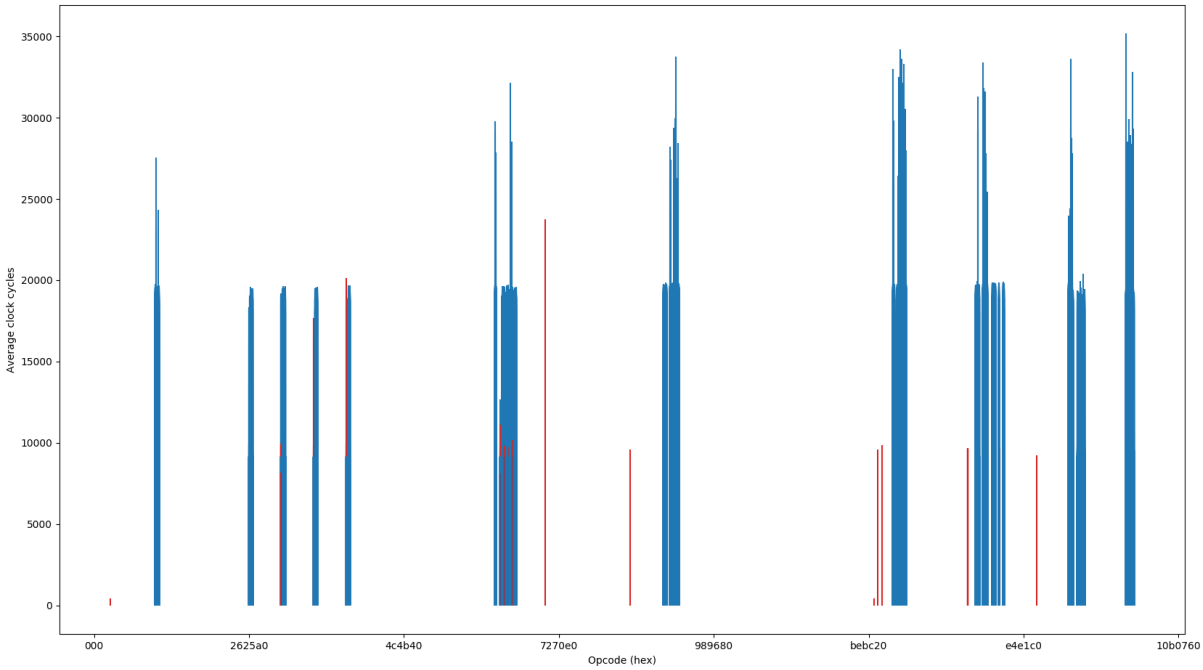


Figure 4.4: Timing attack results on the three-byte encoding space on Intel Bonnell.

Chapter 5

Evaluation and Conclusion

"GCAMTDNBUJ - Garbage Collect All Memory That Does Not Bring User Joy"

@x86instructions, Twitter

5.1 Project Outcomes

Hypothesis 1: undocumented instructions. This proved correct on the HiFive Unleashed and HiFive1, as both exhibit undocumented instructions with the reserved encoding `100xxxxxxxxxxxx00`. These were partially reverse-engineered on the HiFive Unleashed and demonstrated to use two entirely undocumented instruction formats; they exhibit different behaviour on the HiFive1, but reverse-engineering on this platform was unsuccessful. No new undocumented instructions were found on x86, with the focus instead on investigating hypotheses 2 and 3 and facilitating future work on hypothesis 1 via new testing techniques and instruction search strategies; however, the known undocumented `prefetchwt1` instructions were investigated and demonstrated to be aliases of `prefetch`.

Hypothesis 2: undocumented exception and decoding behaviour. Strategies were developed for investigating decoding behaviour via the tunneling algorithm, a timing attack, and the specpoline mechanism. In particular, the first known visualisations of x86 decoding behaviour (beyond the standard opcode maps) were presented, which provide a new method for analysis of decoding behaviour. One instruction was found to have surprising behaviour on Intel Westmere, but unfortunately could not be investigated further due to hardware failure. The undocumented #UD exception behaviour reported in [2] was found to have been corrected in a microcode update, indicating that it was likely a CPU bug. The secrecy surrounding this highlights the problematic opacity of microcode updates; undocumented exception behaviour is a known cause of vulnerabilities, and so any changes in microcode - even if to fix bugs - should not be kept secret by manufacturers.

Hypothesis 3: Transient execution of undocumented instructions. Whilst there was theoretical support for this hypothesis, experimental results with the specpoline mechanism strongly supported that it is incorrect on Intel Broadwell: faulting speculative instructions show uniform behaviour by (at the latest) the decoding stage. This hypothesis remains to be investigated on other speculative microarchitectures, which may demonstrate different behaviour.

Other contributions: All four project aims were met. The experiments described above meet the project's third aim, whilst the code produced for the experiments on x86 and RISC-V has been released on GitHub as v2 of the open-source OpcodeTester tool [2], meeting the project's second aim. The release of this tool will hopefully facilitate instruction fuzzing on other microarchitectures and future research in this area. In addition forks of Sandsifter and NanoBench were released, adding support for disassembly with Intel XED and testing of faulting instructions respectively. Chapter 1 provides an accessible introduction to CPU security, meeting the project's first aim, and Sections 5.4 and 5.5 of this chapter meet the fourth aim of the project by assessing the scope for future work in this area and the outlook for CPU security.

5.2 Critical Evaluation

5.3 Challenges

Variable-length encoding. As discussed in Section 4.3.1, x86's extreme variable-length encoding presents immense challenges for reliable instruction testing. Essentially, on x86 the 1-15 bytes tested cannot be treated as a black-box single instruction which will either execute or fault: they may actually represent a sequence of valid instructions which subsequently crash the program or are falsely identified as an undocumented instruction. In one sense an understanding of x86 encodings is already built into OpcodeTester with its use of Intel XED. However, XED is not infallible and can only decode 15 bytes at a time: an ongoing issue remains that XED may declare a 15-byte sequence invalid because the final instruction is incomplete, whilst at runtime bytes following the test array in memory complete the instruction and the sequence therefore executes. Similarly, the tunnelling algorithm achieves this to some extent by being guided by the decoded instruction length, but an instruction's length does not encode all possible information: there exist instructions of different lengths which are functionally similar enough to treat as identical for coverage purposes, and of the same length which are functionally distinct enough to require separate testing.

It therefore seems advisable for future work on x86 to use more architecture-specific strategies, aiming for coverage of specific architectural features rather than random instruction generation alone (akin to Google's UVM instruction generation for verifying RISC-V RTL [149]). Potential options include manipulating the Mod-R/M and SIB bytes to find undocumented aliases or variants of documented instructions or even synthesising undocumented instructions from the formal semantics of [92] to produce encodings similar to documented instructions. Architecture-specific strategies can still benefit however from complementary approaches such as the timing attack and decoder mapping to avoid overlooking any undocumented instructions which considerably deviate from the documented encodings.

An additional challenge of x86 is that so many of the factors in microarchitectural state (microcode, pipeline details, etc.) are undocumented, and so it is extremely difficult to isolate the cause of suspicious or potentially vulnerable instruction behaviour. I believe there is still considerable value in an application-level tool, as this renders instruction testing more accessible and helps to identify instruction vulnerabilities which are caused by interaction with the operating system (such as the MOV SS/POP SS vulnerability). However, on microarchitectures with complex transient execution effects, such as those of x86, this must be complemented by testing at machine-mode or lower to isolate suspicious instructions' microarchitectural effects. The recently partially reverse-engineered Intel VISA (Visualization of Internal Signals Architecture) debugging interface, which is available on Skylake and newer microarchitectures and permits low-level debugging of the CPU via USB when 'unlocked' using a known vulnerability [150], might enable yet lower-level analysis.

Investigating two architectures. I have greatly appreciated the opportunity to learn about RISC-V and open-source hardware, and believe that my understanding of x86's security, its design choices, and the complex technological and economic context in which it developed has greatly benefited from comparing and contrasting it with RISC-V. After all, it is difficult to propose alternatives to poor design choices if unaware that there *are* alternatives! However, in retrospect I think attempting to investigate two entire architectures was overambitious for the time frame of this project; the workload involved was unsustainable and as a result my experimental findings are more preliminary than I hoped. However, the project's findings and release of the improved OpcodeTester tool have laid groundwork for further research in this area, and I sincerely hope that they will aid other researchers in investigating this under-researched topic.

Hardware failure. The failure of the Westmere laptop and Galileo boards was disappointing. I regret that I was unable to conduct more thorough testing of undocumented instructions on Westmere and to fully debug the cause of the 'HCF' jump, as identifying the cause might have led to a useful Linux kernel or driver patch. Given that the laptop was 9 years old and that events prior to its total failure suggested the graphics card was failing, it seems unlikely that the hardware failure was related to the project's instruction testing. The failure of the Galileo boards meant that it was not possible to conduct instruction testing on either board (hence the lack of discussion of these boards in prior sections). I was unable to communicate with the first board via UART or RS-232, and the board was not recognised by Intel's firmware update tool. Communication was also impossible with a second brand-new board; following suggestions on the Intel forums I attempted a firmware update with Intel's tool, but this failed and bricked the board.

5.3.1 Assumptions and Limitations

Open-source security. A significant assumption throughout this work has been that open-source systems are inherently more secure than closed-source systems. This is widely believed to be the case in the open-source community, with Linus' law stating that *"given enough eyeballs, all bugs are shallow"*, and the substantial case against security by obscurity was presented in Sections 1.2.2 and 2.1. However, there are precedents of vulnerabilities in open-source software which remained unnoticed for decades, such as the Shellshock remote code execution vulnerability present in Bash from 1992-2014 [151]. Penetration testing and auditability of open-source designs increase the likelihood of detecting any vulnerabilities present, but do not guarantee their detection. Moreover, whilst open-source development adds diversity of perspective and more 'eyeballs', its typically highly-distributed nature, with many developers making unfocused contributions, can in itself foster vulnerabilities [152]. Improving open-source verification tooling is therefore also crucial, as it will enable vulnerabilities to be detected earlier in development.

No AMD microarchitectures. Due to lack of other available test platforms, both this project and my prior research project have now investigated x86 solely on Intel microarchitectures. This is unfortunate as Intel's and AMD's microarchitectures and interpretations of the informally-specified x86 ISA differ substantially. I considered using AMD-based cloud computing instances such as AMD EPYC instances on Amazon EC2. However, I was concerned that CPU fuzzing might violate cloud providers' terms of service and affect the availability of other customers' instances on the same server in the event of a crash, and therefore decided against such testing. As OpcodeTester has been made publicly available on GitHub, investigation of AMD CPUs is accessible to others, and I hope that in future work I will also have the opportunity to investigate on AMD platforms. However, until such comparative investigation has been conducted the results presented in this project should be considered Intel-specific.

Hardware performance counters. The analysis in Section 4.4.2 is highly dependent on the reliability of the hardware performance counters. As discussed in [153], the hardware counters on across x86 microarchitectures are known to suffer from inaccuracies (particularly overcounting) and seemingly non-deterministic behaviour, and as described in Section 2.3.3 Broadwell suffers from numerous performance counter errata. The counts I observed may well have been inaccurate; given the proprietary nature of the microarchitectural interactions under investigation, it is impossible to be certain. However, they certainly did not appear to be non-deterministic: the counter values were remarkably consistent across reboots and with varying background load, with a typical variance of just 0-2 µops per value. Furthermore, the characteristic ratios determined from observation are tolerant to slight variance and are dependent only on the counters reporting reasonably consistent values rather than being truly accurate. I did not apply per-process filtering to my results as recommended in [153], as the consistency of the values suggested the process was very rarely being descheduled during the extremely brief performance monitoring period. However, repeated descheduling (perhaps due to very high system load) could potentially lead to false positives; in developing the tool further I would seek to prevent this. The most significant limitation of the transient execution analysis is that testing could only be conducted on a single microarchitecture: by this point in the project the Westmere platform had failed, whilst Bonnell is an in-order microarchitecture with no capacity for transient execution. It would certainly be worthwhile continuing this investigation across multiple microarchitectures, as such low-level pipeline details are likely to vary (particularly on Cascade Lake and subsequent generations, given their hardware mitigations against certain transient execution attacks).

5.4 Future Work

5.4.1 Porting Fuzzing to Other Architectures

> *"Intel products are not intended for use in medical, life saving, life sustaining applications. Unless otherwise agreed in writing by Intel, the Intel products are not designed nor intended for any application in which the failure of the Intel product could create a situation where personal injury or death may occur"* - Intel disclaimer [154])

> *"Some critical medical devices/equipment still use Microsoft XP software supplied by third parties and were affected, including for example, MRI scanners and blood test analysis devices"* [37]

With the exception of the HiFive1 microcontroller, the devices under test for this project are all desktop CPUs. Desktop PC security is undeniably important: whilst desktop CPU manufacturers wash their hands of

responsibility for security with disclaimers that their products should not be used in safety-critical systems, digital infrastructure is now so pervasive that desktops play a critical role in many aspects of modern life. From controlling medical equipment (with alarmingly outdated software, as the quote above highlights) to communicating with industrial programmable logic controllers (PLCs) and coordinating the just-in-time supply chains which ensure shops are stocked with food and medicine, desktop CPUs *are* critical for keeping people safe. However, there are also many other contexts in which CPUs are used which were not considered in this project. Servers store vast quantities of sensitive data; x86 server CPUs can be easily tested with Opcode-Tester, but testing Arm servers would require porting the tool. Similarly, many mobile and embedded devices are powered by Arm CPUs, so porting the tool to this architecture seems highly worthwhile for future work. There are a range of other ISAs including PowerPC, MIPS, and Atmel AVR, but with Arm's dominance in this market [50] targeting its architecture seems most worthwhile.

All CPUs typically also work in tandem with many other processors which may have undocumented behaviour, for example the network card, graphics card, audio card, disk controllers, and security coprocessors. With increasing adoption of machine learning systems in safety-critical or sensitive contexts (such as autonomous driving in cars and facial recognition for airport security and border control), support for fuzzing of graphics cards and domain-specific machine learning processors is particularly crucial. Fuzzing trusted coprocessors such as Intel ME and the AMD PSP for undocumented behaviour also seems highly worthwhile, as their comprehensive access to the main CPU makes them excellent sites for a backdoor. Finally, there are industrial PLCs and other control systems processors used in contexts such as manufacturing and the energy industry. These are perhaps the most security critical of all, as such systems can be extremely dangerous if control cannot be maintained. Nuclear facilities in particular have been subject to sophisticated and persistent attacks such as Stuxnet and TRITON [39] [155]. However, gaining access to such systems to conduct testing is, understandably, challenging, and each must be evaluated within its highly specialised context, likely precluding development of a general-purpose auditing tool.

5.4.2 Instruction Space Search Strategies

Whilst this project proposed one novel strategy for searching the instruction space, there are many others which seem worthy of investigation. As discussed in Section 2.3, it is impossible to achieve full coverage of all possible x86 instruction encodings, and even with the 32-bit RISC-V space a brute-force search is prohibitively slow. This 'search' challenge (where we are aiming to explore or cover the space, rather than search for a specific value) is also known as *state space exploration* and is a problem far from unique to CPU fuzzing. Algorithms and heuristics for the problem are broadly applicable in a range of domains, from fuzzing and verification of systems to artificial intelligence, genetics, drug discovery, and dynamical systems in general across physics, engineering and mathematics. This further motivates investigation of search strategies in the context of CPU fuzzing and verification: such strategies might be generalisable to other domains, leading to benefits far beyond improved CPU security. Evolutionary algorithms such as genetic algorithms are a common class of techniques used for this problem in other domains [156] but have not yet been successfully applied to instruction fuzzing (Domas began but never completed implementing a genetic approach [5]).

A crucial challenge in applying these strategies is creating a meaningful optimisation metric (to determine the distance between states or optimality of a given state). Intuitively, for coverage purposes the distance between two instructions would be how functionally distinct they are; this could be quantified by combining performance counter measurements with observed register state changes. This then presents us with a classic exploration vs. exploitation problem [157]: at what distance is an instruction similar enough to justify skipping testing of it? Defining optimality is harder; one approach would be to seek to optimise the µops issued to µops retired ratio measured with the specpoline mechanism across all instructions unrecognised by the disassembler (see Section 4.4.2). A further promising strategy is the rapidly-exploring random tree algorithm [158]. The algorithm uniformly samples from the search space, attempting to add a selected state to its tree by connecting the state to the closest state already in its tree, subject to any relevant constraints for the problem domain. This uniform sampling biases it towards large Voronoi regions, resulting in the algorithm preferentially expanding into larger unsearched areas. It has primarily been applied so far to contexts with complex physical dynamics, such as motion planning in robotics; CPU state is similarly complex but may not be equally mathematically predictable, so it remains to be seen how effective the algorithm would be.

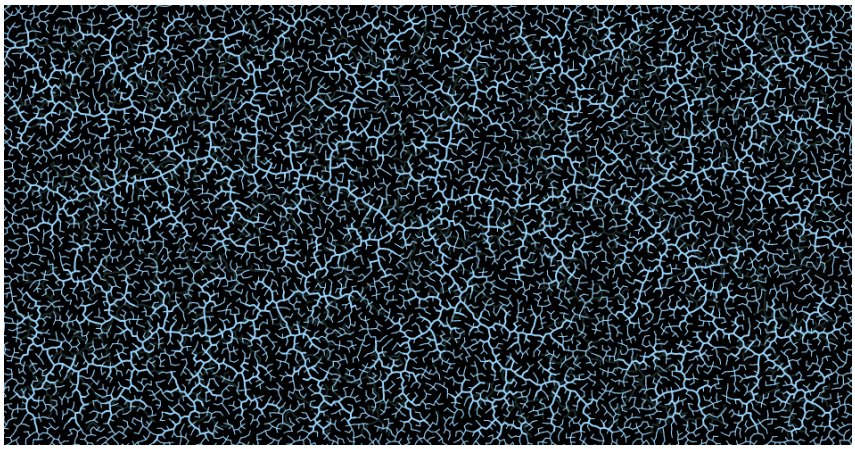


Figure 5.1: A visualisation of the coverage produced by a rapidly-exploring random tree algorithm. The fractal pattern is an artefact of the test data and constraints used, but recalls to mind biologically-evolved efficient coverage strategies in nature, such as formation of neural systems. Reproduced from [159].

5.5 Outlook for CPU Security

Current context. As discussed in Chapter 1, the current technical, economic, and political context provides a compelling case for the necessity of improved CPU security and public CPU auditing. CPU designs and the dominant x86 ISA have grown so complex that it is impossible to comprehensively verify them, even with state-of-the-art industry tooling. Said designs have been found to be vulnerable to an ever-growing set of transient execution attacks and to feature other software-exploitable hardware bugs. The incumbent x86 manfacturers have a monopoly on the industry and have little economic incentive to improve security or attempt to radically innovate on their current flawed microarchitectures. Nation-states are placing increasing focus on cyberwarfare, beginning to invest the resources necessary to conduct large-scale hardware compromise, and we have seen cyber attacks compromising critical national infrastructure and causing societal disruption, whilst governmental security compromise of commodity devices for mass surveillance is ongoing. However, RISC-V and open-source hardware design in general are gaining momentum. A return to the RISC mindset across the industry could help mitigate the current complexity explosion, and RISC-V's free licensing may stimulate innovation in microarchitectural design. Manufacturers such as Western Digital appear to be embracing greater openness in CPU design [54] thanks to RISC-V, which is a hugely beneficial trend for security. However, no ISA can be a panacea, and the undocumented instructions found in this project suggest either inadequate verification or deliberate concealment of undocumented behaviour by the predominant RISC-V manufacturer, SiFive. Both alternatives are troubling and demonstrate why CPU auditing tools will remain necessary in the future.

Security at any cost? Despite the risks posed by CPU compromise, it is important to maintain perspective on the issue. There are substantial costs involved in improving CPU security which cannot be ignored; computer architecture is, after all, a science of trade-offs [160]. Mitigating a CPU vulnerability involves direct economic and organisational costs: substantial developer time is required to implement and deploy a mitigation, and there is a resulting opportunity cost in that this time could instead be spent on vulnerabilities more likely to affect the given organisation. Furthermore, the very same design choices and optimisations which this project has been so critical of from a security perspective also help to improve energy efficiency and boost performance. Improved performance is crucial to a wide range of compute-intensive scientific domains, and security mitigations which impose substantial performance costs - such as the Spectre and Meltdown patches [161] - therefore have knock-on effects on scientific progress. Moreover, with 'unprecedented' systemic transitions and "deep emission reductions in all sectors" now necessary in the next two decades to limit global warming to 1.5°C [162], we may no longer have the luxury of debating whether a security patch is worth the resulting energy inefficiency.

Secure by design. I discuss these trade-offs not with the aim of dismissing the value of CPU security, but to highlight the cost of blindly continuing with the 'penetrate and patch' mentality, with insecure products being released without penalty and bugs being patched over only when they are publicly found to be vulnerable. Whilst this project has approached CPU security from an auditing perspective - and such auditing

is likely to continue to play an important role in CPU security - it should be a last resort in our arsenal of vulnerability detection methods. The wide-ranging costs of subsequent patching when a vulnerability is found are not incurred if the vulnerability is detected prior to manufacture, or if better still it never makes it into the design. The reason so *many* costly patches have recently been necessary is manufacturers' failure to create CPUs which are *secure by design*. Whilst we should still aim for the entirely open-source and fully auditable systems envisioned in Section 1.2.3, this alone is insufficient: we must work towards fully verifiable systems, with formally specified and verified implementations at every abstraction layer [163]. Crucially, this must include system-level verification covering the interactions between layers to detect issues such as cross-layer software-exploitable hardware vulnerabilities, the impact of which has only recently been made apparent [1] [164]. The ISA has a crucial role to play in this: as [165] argues, we should "rethink the ISA" to expose relevant microarchitectural state at the ISA level, thus enabling verification of the correctness of microarchitectural state changes and greater security guarantees. By fundamentally changing our design practices to embrace hardware-software co-design, we can bridge the divide between hardware and software that our tower of leaky abstractions has created, and work towards CPUs - and systems - which are secure by design.

Bibliography

[1] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. *A Systematic Evaluation of Transient Execution Attacks and Defenses*. arXiv preprint arXiv:1811.05441v2, 2019.

[2] C. Easdon. *Automated Analysis of Undocumented Intel x86-64 Instructions and the Case for Public CPU Auditing Tools*. TU Graz, 2018. Paper, presentation slides and v1 OpcodeTester tool available at https://github.com/cattius/opcodetester.

[3] Intel. *Intel X86 Encoder Decoder (Intel XED)*. GitHub repository, 2019. Available at https://github.com/intelxed/xed.

[4] M. J. Clark. *RISC-V Disassembler*. GitHub repository, 2018. Available at https://github.com/michaeljclark/riscv-disassembler.

[5] C. Domas. *Breaking the x86 ISA*. BlackHat USA, 2017. Sandsifter tool available at https://github.com/xoreaxeaxeax/sandsifter/.

[6] A. K. Dewdney. Computer Recreations: Of worms, viruses and Core War. *Scientific American*, March 1989.

[7] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach (Sixth Edition)*. Morgan Kaufmann, 2019.

[8] Gerard O'Regan. *A Brief History of Computing*. Springer Science & Business Media, 2008.

[9] F. Fritz. *Public domain image of Intel i7-7820X*. Flickr, 2018. Available at https://www.flickr.com/photos/130561288@N04/39793547952/.

[10] P. Gelsinger, D. Kirkpatrick, A. Kolodnyand, and G. Singer. *Such a CAD! Coping with the Complexity of Microprocessor Design at Intel*. IEEE Solid-State Circuits Magazine, Volume 2, Issue 3, 2010.

[11] R. Courtland. *The End of the Shrink*. IEEE Spectrum, Volume 50, Issue 11, 2013.

[12] A. Frumusanu. *The iPhone XS & XS Max Review: Unveiling the Silicon Secrets*. AnandTech, 2018. Available at https://www.anandtech.com/show/13392/the-iphone-xs-xs-max-review-unveiling-the-silicon-secrets.

[13] P. M. Hansen, M. A. Linton, R. N. Mayo, M. Murphy, and D. A. Patterson. *A performance evaluation of the Intel iAPX 432*. ACM SIGARCH Computer Architecture News, Volume 10, Issue 4, 1982.

[14] D. May. *CSP, occam and Transputers*. Communicating Sequential Processes. The First 25 Years: Symposium on the Occasion of 25 Years of CSP. Springer Science & Business Media, 2005.

[15] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. *Dark Silicon and the End of Multicore Scaling*. Proceedings of the 38th International Symposium on Computer Architecture, 2011.

[16] T. Dullien. *Security, Moore's law, and the anomaly of cheap complexity*. 10th International Conference on Cyber Conflict, 2018.

[17] E. Sperling. *Quantum Effects At 7/5nm And Beyond*. Semiconductor Engineering, 23rd May 2018. Available at https://semiengineering.com/quantum-effects-at-7-5nm/.

[18] A. Baumann. *Hardware is the new software*. Proceedings of the 16th Workshop on Hot Topics in Operating Systems, 2017.

[19] A. Fog. *Stop the instruction set war*. Agner's CPU Blog, 2009. Available at https://www.agner.org/optimize/blog/read.php?i=25.

[20] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. *Meltdown: Reading Kernel Memory from User Space*. Proceedings of the 27th USENIX Security Symposium, 2018.

[21] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. *Spectre Attacks: Exploiting Speculative Execution*. Proceedings of the 40th IEEE Symposium on Security and Privacy, 2019.

[22] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard. *Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript*. International Conference on Financial Cryptography and Data Security, 2017.

[23] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss. *NetSpectre: Read Arbitrary Memory over Network*. arXiv preprint arXiv:1807.10535, 2018.

[24] O. Sibert, P. A. Porras, and R. Lindell. *The Intel 80x86 Processor Architecture: Pitfalls for Secure Systems*. Proceedings of the 1995 IEEE Symposium on Security and Privacy, 1995.

[25] Jaseg. *Untrusting the CPU: a proposal for secure computing in an age where we cannot trust our CPUs anymore*. 33rd Chaos Communication Congress, 2016.

[26] *Task Force Report: Resilient Military Systems and the Advanced Cyber Threat*. Defense Science Board, US Department of Defense, 2013.

[27] *Global Manufacturing at Intel*. Intel, 2019. Available at https://www.intel.com/content/www/us/en/architecture-and-technology/global-manufacturing.html.

[28] A. Shilov. *Intel considers $11 billion fab in Israel*. AnandTech, 29th January 2019. Available at https://www.anandtech.com/show/13914/intel-considers-11-billion-fab-in-israel.

[29] *TSMC says latest chip plant will cost around $20 bln*. Reuters, 7th December 2017. Available at https://www.reuters.com/article/tsmc-investment/tsmc-says-latest-chip-plant-will-cost-around-20-bln-idUSL3N1O737Z.

[30] J. Villasenor. *Compromised By Design? Securing the Defense Electronics Supply Chain*. Center for Technology Innovation, The Brookings Institute, 2013.

[31] S. Bhunia and M. Tehranipoor. *Hardware Security: A Hands-on Learning Approach*. Morgan Kaufmann, 2018.

[32] G. T. Becker, F. Regazzoni, C. Paar, and W. P. Burleson. *Stealthy Dopant-Level Hardware Trojans*. International Workshop on Cryptographic Hardware and Embedded Systems. Springer, 2013.

[33] T. Hudson. *Modchips of the State: Hardware implants in the supply-chain*. 35th Chaos Communication Congress, 2018.

[34] J. Robertson and M. Riley. *The Big Hack: How China Used a Tiny Chip to Infiltrate U.S. Companies*. Bloomberg Businessweek, 8th October 2018.

[35] J. Naughton. *The tech giants, the US and the Chinese spy chips that never were... or were they?* The Guardian, 13th October 2018. Available at https://www.theguardian.com/commentisfree/2018/oct/13/tech-giants-us-chinese-spy-chips-bloomberg-supermicro-amazon-apple.

[36] R. S. Mueller III. *Report On The Investigation Into Russian Interference In The 2016 Presidential Election, Volume I of II*. US Department of Justice, 2019.

[37] F. Bamber, A. Bowyer, N. Leung, F. Lopes, L. Mills, D. Williams, and R. White. *Investigation: WannaCry cyber attack and the NHS*. National Audit Office (UK), 2018. Available at https://www.nao.org.uk/wp-content/uploads/2017/10/Investigation-WannaCry-cyber-attack-and-the-NHS.pdf.

[38] J. Styczynski, N. Beach-Westmoreland, and S. Stables. *When The Lights Went Out: A Comprehensive Review of the 2015 Attacks On Ukrainian Critical Infrastructure*. Booz Allen Hamilton, 2016. Available at https://boozallen.com/content/dam/boozallen/documents/2016/09/ukraine-report-when-the-lights-went-out.pdf.

[39] J. P. Farwell and R. Rohozinski. *Stuxnet and the Future of Cyber War*. Survival: Global Politics and Strategy, 2011.

[40] A. Foxall. *Putin's Cyberwar: Russia's Statecraft in the Fifth Domain*. Russia Studies Centre & NATO Strategic Communications Centre of Excellence, 2016.

[41] A. Huang. *Supply Chain Security: "If I were a Nation State..."*. BlueHat IL, 2018.

[42] H. Abelson, R. Anderson, S. M. Bellovin, J. Benaloh, M. Blaze, W. Diffie, J. Gilmore, M. Green, S. Landau, P.G. Neumann, R. L. Rivest, J. I. Schiller, B. Schneier, M. Specter, and D. J. Weitzner. *Keys under doormats: mandating insecurity by requiring government access to all data and communications*. Journal of Cybersecurity, Volume 1, Issue 1, Oxford University Press, 2015.

[43] M. Moon. *Anyone can now print out all TSA master keys*. Engadget, 28th July 2016. Available at https://www.engadget.com/2016/07/28/tsa-master-key-3d-models/.

[44] N. Perlroth, J. Larson, and S. Shane. *N.S.A. Able to Foil Basic Safeguards of Privacy on Web*. The New York Times, 5th September 2013. Available at https://www.nytimes.com/2013/09/06/us/nsa-foils-much-internet-encryption.html.

[45] *Secret Documents Reveal N.S.A. Campaign Against Encryption : Excerpt from 2013 Intelligence Budget Request*. The New York Times, 5th September 2013. Available at https://archive.nytimes.com/www.nytimes.com/interactive/2013/09/05/us/documents-reveal-nsa-campaign-against-encryption.html.

[46] E. Silfversten, W. Phillips, G. P. Paoli, and C. Ciobanu. *Economics of Vulnerability Disclosure*. European Union Agency for Network and Information Security (ENISA), 2018.

[47] *Telecommunications and Other Legislation Amendment (Assistance and Access) Bill 2018*. Parliament of Australia, 2018.

[48] K. Opsahl. *Warrant Canary Frequently Asked Questions*. Electronic Frontier Foundation, 2014. Available at https://www.eff.org/deeplinks/2014/04/warrant-canary-faq.

[49] *Product Security at Intel*. Intel, 2019. Available at https://www.intel.com/content/www/us/en/corporate-responsibility/product-security.html.

[50] *Arm Limited : Q3 2018 Roadshow Slides*. Arm, 2018.

[51] A. L. Shimpi. *The ARM Diaries, Part 1: How ARM's Business Model Works*. AnandTech, 2013. Available at https://www.anandtech.com/show/7112/the-arm-diaries-part-1-how-arms-business-model-works.

[52] G. Tang (ed. I. W. Brown). *Intel and the x86 Architecture: A Legal Perspective*. JOLT Digest (Digest of The Harvard Journal of Law & Technology), 2011. Available at https://jolt.law.harvard.edu/digest/intel-and-the-x86-architecture-a-legal-perspective.

[53] L. Hively, F. Sheldon, and A. Squicciarini. *Toward Scalable Trustworthy Computing Using the Human-Physiology-Immunity Metaphor*. IEEE Security & Privacy, Volume 9, Issue 4, 2011.

[54] A. Nath et al. *SweRV RISC-V Core from Western Digital*. GitHub repository., 2019. Available at https://github.com/westerndigitalcorporation/swerv_eh1.

[55] A. Shilov. *Western Digital to Use RISC-V for Controllers, Processors, Purpose-Built Platforms*. AnandTech, 14th December 2017. Available at https://www.anandtech.com/show/12133/western-digital-to-develop-and-use-risc-v-for-controllers.

[56] J. Xie. *NVIDIA RISC-V Evaluation Story*. 4th RISC-V Workshop, 2016.

[57] *CHIPS (Common Hardware for Interfaces, Processors and Systems) Alliance*. Linux Foundation Projects, 2019. Available at https://chipsalliance.org/.

[58] *Coreboot: fast, secure and flexible OpenSource firmware*. 2019. Available at https://www.coreboot.org/.

[59] *CoreGuard*. Dover Microsystems, 2019. Available at https://www.dovermicrosystems.com/solutions/coreguard/.

[60] *CryptoManager Root of Trust*. Rambus, 2019. Available at https://www.rambus.com/security/cryptomanager-platform/root-of-trust/.

[61] B. Marshall. *On Hardware Verification in an Open Source Context*. Workshop on Open Source Design Automation, 2019.

[62] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard. *Systematic Classification of Side-Channel Attacks: A Case Study for Mobile Devices*. IEEE Communications Surveys & Tutorials, Volume 20, Issue 1, 2010.

[63] J. Synnott, D. Dietzel, and M. Ioannou. *A review of the polygraph: history, methodology and current status*. Crime Psychology Review, Volume 1, Issue 1, 2015.

[64] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri. *Port Contention for Fun and Profit*. IACR Cryptology ePrint Archive, 2018.

[65] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. *Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks*. 27th USENIX Security Symposium, 2018.

[66] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus. *SMoTherSpectre: exploiting speculative execution through port contention*. arXiv preprint arXiv:1903.01843v1, 2019.

[67] J. Van Bulck, F. Piessens, and R. Strackx. *Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic*. Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018.

[68] J. Stecklina and T. Prescher. *LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels*. arXiv preprint, arXiv:1806.07480v1, 2018.

[69] S. Islam, A. Moghimi, I. Bruhns, M. Krebbel, B. Gulmezoglu, T. Eisenbarth, and B. Sunar. *SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks*. arXiv preprint arXiv:1903.00446v1, 2019.

[70] T. Garfinkel and A. Warfield. *What virtualization can do for security*. The USENIX Magazine, Volume 32, Issue 6, 2007.

[71] C. Bowman, A. Gesher, J. K. Grant, and D. Slate. *The Architecture of Privacy: On Engineering Technologies that Can Deliver Trustworthy Safeguards*. O'Reilly Media, 2015.

[72] X. Ruan. *Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*. Apress, 2014.

[73] *INTEL-SA-00086*. Intel, 2018. Available at https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00086.html.

[74] *Intel CSME, Server Platform Services, Trusted Execution Engine and Intel Active Management Technology 2018.4 QSR Advisory*. Intel, 2019. Available at https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00185.html.

[75] T. Claburn. *Security hole in AMD CPUs' hidden secure processor code revealed ahead of patches*. The Register, 6th January 2018. Available at https://www.theregister.co.uk/2018/01/06/amd_cpu_psp_flaw/.

[76] *Severe Security Advisory on AMD Processors*. CTS Labs, 2018. Available at https://safefirmware.com/amdflaws_whitepaper.pdf.

[77] A. Ermolov, D. Evdokimov, and M. Malyutin. *Intel AMT Stealth Breakthrough*. BlackHat USA, 2017.

[78] *Source Code Analysis Tools*. Open Web Application Security Project (OWASP), 2019. Available at https://www.owasp.org/index.php/Source_Code_Analysis_Tools.

[79] Y. Alaoui. *Deep dive into Intel Management Engine disablement*. Purism, 2017. Available at https://puri.sm/posts/deep-dive-into-intel-me-disablement/.

[80] J. Rutkowska. *x86 Considered Harmful*. Invisible Things Labs, 2015. Available at https://blog.invisiblethings.org/papers/2015/x86_harmful.pdf.

[81] M. Necasek. *Undocumented 8086 Opcodes*. OS/2 Museum Blog, 2013. Available at http://www.os2museum.com/wp/undocumented-8086-opcodes/.

[82] J. Bayko. *Great Microprocessors of the Past and Present, Version 10.0.0*. 1997. Available at https://www.cl.cam.ac.uk/teaching/2006/CompArch/documents/all/trends/cpu_history.html.

[83] A. Albertini. *Corkami documentation: x86 oddities*. GitHub repository, 2017. Available at https://github.com/corkami/docs/blob/master/x86/x86.md.

[84] R. Collins. *The LOADALL Instruction*. Available at http://www.rcollins.org/articles/loadall/tspec_a3_doc.html.

[85] B. Crothers. *Intel changing its tune on revealing chip flaws*. InfoWorld magazine, 9th January 1995.

[86] V. Pratt. *Anatomy of the Pentium bug*. Theory and Practice of Software Development (TAPSOFT '95), 1995.

[87] *Mobile/Desktop 5th Generation Intel Core Processor Family Specification Update, Revision 019*. Intel, July 2017.

[88] *6th Generation Intel Processor Family Specification Update*. Intel, November 2018.

[89] *7th Generation Intel Processor Family and 8th Generation Intel Processor Family for U Quad Core Platforms*. Intel, August 2017.

[90] N. Peterson and N. Mulasmajic. *POP SS/MOV SS Vulnerability*. 2018. Available at https://everdox.net/popss.pdf.

[91] *Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*. Intel, 2019.

[92] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Rosu. *A Complete Formal Semantics of x86-64 User-Level Instruction Set Architecture*. ACM SIGPLAN Conference on Programming Language Design and Implementation (to appear), 2019.

[93] S. Heule, E. Schkufza, R. Sharma, and A. Aiken. *Stratified Synthesis: Automatically Learning the x86-64 Instruction Set*. Proceedings of the 37th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation, 2016.

[94] T. P. Morgan. *Intel's answer to ARM: Customisable x86 chips with HIDDEN POWERS*. The Register, 20th May 2013. Available at https://www.theregister.co.uk/2013/05/20/intel_chip_customization/.

[95] B. Case. *Intel Reveals Pentium Implementation Details*. Microprocessor Report, Volume 7, Number 4, March 1993.

[96] D. Goodin. *'Super-secret' debugger discovered in AMD CPUs*. The Register, 15th November 2010.

[97] M. Seaborn. *Issue 2010: Escape from x86-64 inner sandbox using BSF instruction*. Monorail, the issue tracking tool for chromium-related projects, 2011. Available at https://bugs.chromium.org/p/nativeclient/issues/detail?id=2010.

[98] G. Dunlap. *The Intel SYSRET privilege escalation*. Xen Project, 2012. Available at https://xenproject.org/2012/06/13/the-intel-sysret-privilege-escalation/.

[99] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. *Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR*. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016.

[100] C. Domas. *Hardware Backdoors in x86 CPUs*. BlackHat USA, 2018.

[101] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. *Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors*. Proceedings of the 41st Annual International Symposium on Computer Architecture, 2014.

[102] D. Gruss, C. Maurice, and S. Mangard. *Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript*. Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, 2016.

[103] V. J. M. Manés, H.S. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. *The Art, Science, and Engineering of Fuzzing : A Survey*. arXiv preprint, arXiv:1812.00140v4, 2019.

[104] J. Li, B. Zhao, and C. Zhang. *Fuzzing: a survey*. Cybersecurity, Springer, 2018.

[105] M. S. Hrishikesh, M. Rajagopalan, S. Sriram, and R. Mantri. System Validation at ARM : Enabling our Partners to Build Better Systems, 2016.

[106] C. Domas. *The ring 0 facade: awakening the processor's inner demons*. DEF CON 26, 2018.

[107] J. Zhu, W. Song, Z. Zhu, J. Ying, B. Li, B. Tu, G. Shi, R. Hou, and D. Meng. *CPU Security Benchmark*. Proceedings of the 1st Workshop on Security-Oriented Designs of Computer Architectures and Processors, 2018.

[108] C. Domas. *Continuing to Break the x86 Instruction Set*. Shakacon X, 2018. Conference presentation.

[109] Blitz. *A bare-metal x86 instruction set fuzzer a la Sandsifter*. GitHub, 2019. https://github.com/blitz/baresifter/.

[110] *VIA C3 Nehemiah Processor Datasheet, Revision 1.13*. Via Technologies, 2004.

[111] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems (Second Edition)*. Wiley, 2008.

[112] A. C. Hobbs (ed. C. Tomlinson). *Locks and Safes: The Construction of Locks*. Virtue and Co., 1868.

[113] A. M. Matwyshyn, A. Cui, A. D. Keromytis, and S. J. Stolfo. *Ethics in Security Vulnerability Research*. IEEE Security & Privacy, Volume 8, Issue 2, 2010.

[114] R. Gevers, M. Rieback, M. R. van Geelen, V. Gevers, E. van Andel, and Z. Done. *Coordinated Vulnerability Disclosure: The Guideline*. National Cyber Security Centre (NL), 2018.

[115] T. F. Dullien. *Weird machines, exploitability, and provable unexploitability*. IEEE Transactions on Emerging Topics in Computing, 2017.

[116] P. G. Neumann. *Fundamental Trustworthiness Principles*. SRI International, 2017.

[117] *SiFive FU540-C000 Manual v1p0*. SiFive, 2018.

[118] Y. Yarom and K. Falkner. *FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack*. 23rd USENIX Security Symposium, 2014.

[119] A. Waterman and K. Asanović. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10 (draft specification)*. The RISC-V Foundation, 2017.

[120] D. Kaplan. *When hardware must "just work": An inside look at x86 CPU design*. 32nd Chaos Communication Congress, 2015.

[121] M. Mushtaq, A. Akram, M. K. Bhatti, M. Chaudhry, V. Lapotre, and G. Gogniat. *NIGHTs-WATCH: A Cache-Based Side-Channel Intrusion Detector using Hardware Performance Counters*. Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy, 2018.

[122] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon. *Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters*. Proceedings of the 18th International Symposium of Research in Attacks, Intrusions, and Defenses, 2015.

[123] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel, April 2019.

[124] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, and A. Naik. *Replacing Testing with Formal Verification in Intel Core i7 Processor Execution Engine Validation*. Proceedings of the 21st International Conference on Computer Aided Verification, 2009.

[125] *AMD64 Architecture Programmer's Manual, Volumes 1-5*. AMD, 2019.

[126] *Deep Dive: Retpoline: A Branch Target Injection Mitigation*. Intel, 2019. Available at https://software.intel.com/security-software-guidance/insights/deep-dive-retpoline-branch-target-injection-mitigation.

[127] A. Waterman and K. Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. The RISC-V Foundation, 2017.

[128] T. Newsome. *RISC-V External Debug Support Version 0.13*. SiFive, 2017.

[129] A. S. Waterman. *Design of the RISC-V Instruction Set Architecture*. UC Berkeley, 2016.

[130] A. Fog. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*. 2018.

[131] *Mobile Intel Atom Processor N270 Single Core Datasheet*. Intel, May 2008.

[132] *Intel Atom Processor N270 Series Specification Update, Revision 009*. Intel, July 2014.

[133] *SiFive FE310-G000 Manual v2p3*. SiFive, 2017.

[134] S. Krishnan. *Red Hat Bugzilla Bug 1202858*. 2015. Available at https://bugzilla.redhat.com/show_bug.cgi?id=1202858.

[135] *Booting Debian demo image*. SiFive Forums, 2019. Available at https://forums.sifive.com/t/booting-debian-demo-image/2213.

[136] *SIGNAL(7): Linux Programmer's Manual*. man-pages Project. Available at http://man7.org/linux/man-pages/man7/signal.7.html.

[137] R. McElrath, R. Seacord, D. Svoboda, V. Patel, M. Thompson, S. Haas, A. Singhal, C. J. Lallier, A. Ballman, J. Loo, A. Gale, W. Snavely, L. Whiting, A. Hicken, R. Cherukuri, G. A. Campbell, M. Rozenau, and A. Gangopadhyay. *MSC22-C. Use the setjmp(), longjmp() facility securely*. SEI CERT C Coding Standard Wiki, 2018. Available at https://wiki.sei.cmu.edu/confluence/display/c/MSC22-C.+Use+the+setjmp%28%29%2C+longjmp%28%29+facility+securely.

[138] C. Wolf. *End-to-end formal ISA verification of RISC-V processors with riscv-formal*. 34th Chaos Communication Congress, 2017.

[139] C. Easdon and C. Domas. *Sandsifter: XED fork*. GitHub repository, 2019. Available at https://github.com/cattius/sandsifter.

[140] C. Easdon and A. Abel. *NanoBench fork with exception handling and support for abitrary machine code*. GitHub repository, 2019. Available at https://github.com/cattius/nanoBench.

[141] A. Abel and J. Reineke. *uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures*. Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 2019.

[142] *Ubuntu Manpage: arc4random, arc4random_buf, arc4random_uniform, arc4random_stir, arc4random_addrandom - arc4*. Canonical, 2019. Available at http://manpages.ubuntu.com/manpages/disco/en/man3/arc4random.3bsd.html.

[143] G. Paoloni. *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures*. Intel, 2010.

[144] H. Wong. *The Microarchitecture Behind Meltdown*. 2018. Available at http://blog.stuffedcow.net/2018/05/meltdown-microarchitecture/.

[145] L. Torvalds et al. *traps.c*. Linux kernel GitHub Repository., 2019. Available at https://github.com/torvalds/linux/commits/master/arch/x86/kernel/traps.c.

[146] S. D. Rodgers, R. Vidwans, J. Huang, M. A. Fetterman, and K. Huck. *Method and apparatus for generating event handler vectors based on both operating mode and event type*. Intel. US Patent 5889982, 1999.

[147] C. P. Roth, R. P. Singh, and G. A. Overkamp. *Exception handling using an exception pipeline in a pipelined processor*. Intel. US Patent 6823448, 2000.

[148] G. L. Brown and R. G. Prasadh. *System for inserting a supplemental micro-operation flow into a macroinstruction-generated micro-operation flow*. Intel. US Patent 5867701, 1999.

[149] T. Liu, R. Ho, and U. Jonnalagadda. *UVM-based RISC-V processor verification platform*. RISC-V Summit, Santa Clara, 2018.

[150] M. Ermolov and M. Goryachy. *Intel VISA: Through the Rabbit Hole*. BlackHat Asia, 2019.

[151] N. Perlroth. *Security Experts Expect 'Shellshock' Software Bug in Bash to Be Significant*. The New York Times, 25th September 2014. Available at https://www.nytimes.com/2014/09/26/technology/security-experts-expect-shellshock-software-bug-to-be-significant.html.

[152] A. Meneely and L. Williams. *Secure Open Source Collaboration: An Empirical Study of Linus' Law*. Proceedings of the 16th ACM Conference on Computer and Communications Security, 2009.

[153] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose. *SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security*. Proceedings of 40th IEEE Symposium on Security and Privacy, 2019.

[154] *Intel Terms of Use*. Intel, 2019. Available at https://www.intel.com/content/www/us/en/legal/terms-of-use.html.

[155] B. Johnson, D. Caban, M. Krotofil, D. Scali, N. Brubaker, and C. Glyer. *Attackers Deploy New ICS Attack Framework "TRITON" and Cause Operational Disruption to Critical Infrastructure*. FireEye Threat Research, 2017. Available at https://www.fireeye.com/blog/threat-research/2017/12/attackers-deploy-new-ics-attack-framework-triton.html.

[156] D. Corne and M. A. Lones. *Evolutionary Algorithms*. Handbook of Heuristics, Springer, 2018.

[157] M. Črepinšek, S. Liu, and M. Mernik. *Exploration and Exploitation in Evolutionary Algorithms: A Survey*. ACM Computing Surveys, Volume 45, Issue 3, 2013.

[158] S. M. Lavalle. *Rapidly-Exploring Random Trees: A New Tool for Path Planning*. Department of Computer Science, Iowa State University, 1998.

[159] J. Davies. *Rapidly-Exploring Random Tree*. 2014. Available at https://www.jasondavies.com/rrt/.

[160] Y. Patt. *Requirements, Bottlenecks, and Good Fortune: Agents for Microprocessor Evolution*. Proceedings of the IEEE, Volume 89, Issue 11, 2001.

[161] M. Larabel. *The Current Spectre / Meltdown Mitigation Overhead Benchmarks On Linux 5.0*. Phoronix, 2019. Available at https://www.phoronix.com/scan.php?page=article&item=linux50-spectre-meltdown.

[162] Edited by: V. Masson-Delmotte, P. Zhai, H. Pörtner, D. Roberts, J. Skea, P.R. Shukla, A. Pirani, W. Moufouma-Okia, C. Péan, R. Pidcock, S. Connors, J.B.R. Matthews, Y. Chen, X. Zhou, M.I. Gomis, E. Lonnoy, T. Maycock, M. Tignor, and T. Waterfield. IPCC Summary for Policymakers 2018. *Global Warming of 1.5°C. An IPCC Special Report on the impacts of global warming of 1.5°C above pre-industrial levels and related global greenhouse gas emission pathways, in the context of strengthening the global response to the threat of climate change, sustainable development, and efforts to eradicate poverty*, 2018.

[163] D. May, G. Barrett, and D. Shepherd. *Designing chips that work*. Philosophical Transactions of the Royal Society of London. Series A: Physical and Engineering Sciences, Volume 339, Issue 1652, 1992.

[164] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A. Sadeghi, and J. Rajendran. *When a Patch is Not Enough - HardFails: Software-Exploitable Hardware Bugs*. ArXiv preprint arXiv:1812.00197v1, 2018.

[165] J. Lowe-Power, V. Akella, M. K. Farrens, S. T. King, and C. J. Nitta. *Position Paper: A Case for Exposing Extra-Architectural State in the ISA*. Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy, 2018.