

Microarchitectural Vulnerabilities in Heterogeneous Computing and Cloud Systems

Zane Weissman



A Dissertation
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for the
Degree of Doctor of Philosophy in
Electrical and Computer Engineering
April 2024

APPROVED:

Professor Berk Sunar
Advisor
Worcester Polytechnic Institute

Professor Thomas Eisenbarth
Committee Member
University of Lübeck

Professor Shahin Tajik
Committee Member
Worcester Polytechnic Institute

Abstract

This dissertation brings together some of the defining trends of early 21st century computing—cloud computing, heterogeneous computing, and the increasingly complex microarchitectures that support them—and analyzes the microarchitectural threat landscapes of these systems while presenting a number of new vulnerabilities which they introduce. Microarchitectural attacks made headlines in 2018 with the disclosures of Spectre and Meltdown, two CPU vulnerabilities affecting devices as diverse as smartphones, laptops, and enterprise servers, and the following wave of research interest has uncovered countless variants and new, unrelated microarchitectural vulnerabilities. These vulnerabilities stem from bugs (and features) of the increasingly complex microarchitectures of modern CPUs that squeeze as much performance as possible out of every transistor. While it remains to be seen if Moore’s law is truly dead, diminishing returns in CPU performance gains and increased interest in specialized computing tasks as diverse as 3D graphics rendering, cryptography, and machine learning have driven the development of GPUs, FPGAs, and other devices that work *heterogeneously* alongside traditional CPUs. CPUs, in turn, have gained new features to tightly integrate these peripherals with their large shared caches and main memory for heterogeneous parallelism. Cloud service providers (CSPs) have lowered the cost of entry to heterogeneous computing by leasing compute time on GPUs and FPGAs, and

even customers who don't specifically rent one of these peripherals will often have their CPU workloads optimized by smart network cards and storage devices that rely on the same technologies. However, the integration of these devices broadens the attack surfaces of several known microarchitectural vulnerabilities and even introduces new ones. This is of particular concern in cloud environments, where service providers share computational resources of all kinds between many users to maximize power and cost efficiency, often in ways that are opaque to their customers and to end users whose private data is handled in shared environments. Therefore, CSPs bear a major responsibility in ensuring that microarchitectural threats are mitigated where possible, and that client workloads are architecturally isolated where mitigations don't exist or are unfeasibly expensive to implement.

First, this work analyzes the architectural features of Intel's Arria 10 GX FPGA platform, a system designed specifically for heterogeneous computing and presents the first FPGA to FPGA, FPGA to CPU, and CPU to FPGA cache timing side-channels. Then, we present Jackhammer, a novel, efficient, and stealthy hardware implementation of the Rowhammer for the Arria 10 GX FPGA. Next, we show that I/O memory management units—intended to ensure proper isolation of peripherals—are the source of a new attack surface between FPGAs, GPUs, or other peripherals. Turning to cloud computing, we investigate the microarchitectural security of the Firecracker virtual machine manager, which powers a significant portion of AWS's compute services. we demonstrate that Firecracker provides negligible defense against major classes

of microarchitectural attacks, uncover holes in AWS’s setup recommendations for the microarchitectural security of Firecracker production hosts, and even identify a variant of the Medusa side-channel attack that works in a Firecracker virtual machine but not outside of it. Finally, we investigate a new fault injection technique enabled by the latest and most sophisticated Rowhammer techniques: adjacent bit memory faults; these faults in combination with a recently discovered lattice attack algorithm enable an incredibly powerful key recovery attack against ECDSA signatures. we establish the existence of such faults in modern hardware and lay out solutions to the practical problems an attacker must address to put together a real-world attack.

In the course of evaluating these vulnerabilities, we also suggest and analyze a variety of countermeasures, both hypothetical and available, to be implemented in hardware, firmware, system software, or user-level software. We highlight the specific challenges of securing heterogeneous and cloud systems against microarchitectural attacks and emphasize the need for defenses at every level. We hope that this work encourages hardware designers, cloud systems engineers, and cloud service developers to reassess threat models and isolation assumptions when developing secure systems with shared and heterogeneous hardware.

Acknowledgments

This work would not have been possible without my advisor, Berk Sunar, who pushed me to take pride in my work and always held the highest respect for my autonomy; Thore Tiemann, whose dependability, intellectual curiosity, and hard work made him the best research partner I could ask for; Thomas Eisenbarth, who always gave measured advice and who lent me his bike while I was in Lübeck; Shahin Tajik, who was always a pleasure to work with as a TA, and who served on my committee with Berk Sunar and Thomas Eisenbarth; and Evan Custodio, without whom the research project that got me interested in (and funded a good part of) my Ph.D. would not have existed.

Thank you to Daniel Moghimi, Andrew Adiletta, Caner Tol, Kemal Derya, Saad Islam, Yarkin Doroz, Kristi Rahman, and all the other students and faculty of CHIPS lab with whom I've exchanged ideas, papers, or code; Alpa Trivedi, Sayak Ray, and Thomas Unterluggauer of Intel for their advice, comments, and conversations; and the anonymous reviewers of all of my conference paper submissions.

Finally, thank you to everyone who supported me outside of this work: my parents, my wonderful partner Oscar, his parents, my grandma, my sisters, and my friends. Your love has made this long journey worth the effort.

Research presented in this work was funded in part by the Intel Corporation, the Qatar National Research Fund, and National Science Foundation (NSF) grants CNS-1814406 and CNS-2026913.

Contents

1	Introduction	1
1.1	Heterogeneous Computing in the Cloud	2
1.2	Serverless Cloud Computing	5
1.3	Signature Correction Attacks	8
1.4	Contributions	10
1.5	Previous Publications and Coauthors	11
2	Background	13
2.1	Cache Attacks	13
2.2	Translation Look-Aside Buffers TLBs	17
2.3	Attacks on TLBs	17
2.4	PCIe	18
2.5	IOMMUs	20
2.6	Rowhammer	21
2.7	Meltdown and MDS	25
2.7.1	Basic MDS Variants	26

2.7.2	Medusa	27
2.7.3	TSX Asynchronous Abort	28
2.7.4	MDS Mitigations	28
2.8	Spectre	30
2.8.1	Spectre Mitigations	31
2.9	RSA-CRT Signing	31
2.10	ECDSA Signing	32
2.11	Lattice Attacks on the Hidden Number Problem (HNP)	34
2.12	Bleichenbacher’s Fourier Analysis Based Technique	38
2.13	Serverless Cloud Computing and MicroVMs	39
2.14	AWS Firecracker	40
2.14.1	Firecracker Security Recommendations	42
3	Related Works	43
3.1	Works on FPGA Security	43
3.2	Works on Heterogeneous Microarchitectural Attacks	44
3.3	Attacks on IOMMUs	44
3.4	Works on MicroVM Security	45
4	JackHammer	47
4.1	Introduction	47
4.1.1	Contributions	47
4.1.2	Experimental Setup	49
4.2	Analysis of Intel FPGA-CPU Systems	50

4.2.1	Intel FPGA Platforms	51
4.2.2	Intel’s FPGA-CPU Compatibility Layers	52
4.2.3	Cache and Memory Architecture on the Intel FPGAs .	54
4.3	JackHammer Attack	57
4.3.1	JackHammer: Our FPGA Implementation of Rowhammer	58
4.3.2	JackHammer on the FPGA PAC vs. CPU Rowhammer	59
4.3.3	JackHammer on the Integrated Arria 10 vs. CPU Rowhammer	63
4.3.4	The Effect of Caching on Rowhammer Performance . .	66
4.4	Fault Attack on RSA using JackHammer	68
4.4.1	RSA Fault Injection Attacks	69
4.4.2	Our Attack	71
4.4.3	Performance of the Attack	73
4.5	Cache Attacks on Intel FPGA-CPU Platforms	75
4.5.1	Cache Attacks from FPGA PAC to CPU	77
4.5.2	Cache Attacks from Integrated Arria 10 FPGA to CPU	79
4.5.3	Cache Attacks from CPU to Integrated Arria 10 FPGA	83
4.5.4	Intra-FPGA Cache Side-Channels	85
4.6	Countermeasures	86
5	IOTLB-SC	89
5.1	Introduction	89
5.2	Identifying IOTLB Side-Channels	91

5.2.1	System Setup	92
5.2.2	IOTLBs Cause Timing Behavior	94
5.2.3	Tools for Testing IOMMU Behavior	95
5.2.4	Threat Models	97
5.3	Constructing Eviction Sets	100
5.3.1	Initial IOTLB Organization Hypothesis	100
5.3.2	A New Approach to Eviction Set Construction	102
5.4	Analysis of Side-Channel Leakages	108
5.4.1	Web Access Leakage	109
5.4.2	GPU-Accelerated SQL Database Leakage	111
5.4.3	Side-Channel Impact	113
5.5	Covert Channels	114
5.5.1	Covert Channel between Peripherals	115
5.5.2	Covert Channel from CPU to Peripheral	118
5.6	Countermeasures	120
5.6.1	Securing Existing Systems	121
5.6.2	Securing Future IOMMUs	122
6	Firecracker	132
6.1	Introduction	132
6.1.1	Responsible Disclosure	133
6.2	Threat Models	134
6.3	Analysis of Firecracker’s Containment Systems	135

6.4	Analysis of microarchitectural attacks and defenses in Firecracker microVMs	138
6.4.1	Medusa	139
6.4.2	RIDL and More	143
6.4.3	Spectre	145
6.5	Impact	150
6.6	Related Works	151
7	DoubleHammer	152
7.1	Introduction	152
7.1.1	Our Contributions	154
7.2	Threat Model	155
7.3	Profiling	156
7.3.1	OpenSSL Profiling	156
7.3.2	DRAM Profiling	164
7.3.3	Hammering Techniques	165
7.4	Collection	168
7.4.1	Physical Co-location	169
7.4.2	Allocation of a Target Page	169
7.5	Analysis	170
7.5.1	Signature Correction on Adjacent Nonce Faults	170
7.6	Countermeasures	173
7.6.1	Hardware Countermeasures	174

7.6.2 Software Countermeasures	176
8 Conclusion	178
Appendix A Mean Spectral Data Per Site	209
Appendix B Extended RIDL Mitigations Table	211

List of Tables

4.1	Overview of the caching hints configurable over CCI-P on an integrated FPGA.	55
4.2	Performance of our JackHammer exploit compared to a standard software CPU Rowhammer with various eviction intervals.	75
4.3	Summary of our cache attacks analysis.	76
5.1	Overview of the system setups used in this work.	92
5.2	Notation used in algorithms.	102
5.3	Comparison of eviction set finding algorithms on the IOTLB.	107
5.4	Throughput and error rate for the covert channels tested on the <i>a10l</i> system.	115
6.1	Overview of discovered microarchitectural vulnerabilities not fully prevented by the recommended production host settings for AWS Firecracker prior to our disclosure.	139
6.2	Presence of Medusa side-channels with all microarchitectural defense kernel options disabled.	140

6.3	Mitigations necessary to protect the host vs. Firecracker victims from Medusa attacks.	141
6.4	Mitigations necessary to protect the host vs. Firecracker victims from RIDL and other microarchitectural data sampling (MDS) attacks.	142
6.5	Spectre PoCs run with Amazon Web Services (AWS) Firecracker recommended countermeasures.	147
7.1	Virtual addresses of the nonce throughout operation of the SSL/TLS server.	157
7.2	Results of page baiting experiment.	161
7.3	Influence of data pattern on occurrence of double bit flips. . .	168
7.4	Comparison of lattice reduction cost estimates for 2-bit nonce leakage.	173

List of Figures

2.1	Major MDS attack pathways and variant names on Intel CPUs.	26
2.2	Fsrecker threat containment diagram—adapted from AWS.	41
4.1	Overview of the architecture of Intel FPGAs.	51
4.2	Distributions of hammering rates on FPGA PAC and i7-3770.	60
4.3	Distributions of flip rates on FPGA PAC and i7-3770.	61
4.4	Distributions of total flips after 200 million to 2 billion hammers on PAC.	62
4.5	Distributions of total flips after 200 million to 2 billion hammers on i7-3770.	63
4.6	Time series plotting number of flips on a row-by-row basis.	64
4.7	Distributions of hammering rates on integrated Arria 10 and Xeon E5-2600 v4.	65
4.8	Distributions of hammering rates with cachable and uncachable memory.	65
4.9	WolfSSL RSA Fault Injection Attack.	68
4.10	Mean number of signatures to fault at various eviction intervals.	76

4.11 Latency for PCIe read requests on an FPGA PAC served by the CPU's LLC or main memory.	77
4.12 Latency for UPI read requests on an integrated Arria 10 served by the FPGA's local cache, CPU's LLC, or main memory.	78
4.13 Covert channel measurements and decoding.	80
4.14 Memory access and <code>flush</code> latency from the CPU.	84
5.1 Comparison of threat models.	124
5.2 Stack diagram of the network card side-channel test.	125
5.3 Variation in caching behavior of network card across system reboots.	126
5.4 Number and size of constructed eviction sets.	127
5.5 Stack diagram of the web access fingerprinting attack.	128
5.6 Stack diagram of the CPU to peripheral and peripheral to peripheral covert channel and side-channel tests.	128
5.7 Measurements for the conducted experiments with the SQL database.	129
5.8 Peripheral to peripheral covert channel transmissions	130
5.9 Stack diagram of the GPU accelerated SQL database covert channel across virtual machines.	131
6.1 Spectre mitigations enabled in the host and guest kernel during the Spectre tests.	145

7.1	Distribution of 12-bit memory page offsets of the nonce k in OpenSSL on DDR4 system.	159
7.2	Attack window hit rates with SMC denial of service.	163
7.3	Using a sliding window approach to finding adjacent bit flips. .	166
7.4	Average number of bit flips on an 128MB buffer vs the number of sides in a multi-sided Rowhammer attack.	167

Chapter 1

Introduction

Microarchitectural vulnerabilities have been studied for decades, but the most drastic change in the known threat landscape occurred only a few years ago when Spectre and Meltdown were both disclosed in 2018. These speculative execution attacks exploit bugs in out-of-order execution, a feature crucial for performance and deeply integrated into the microarchitecture of modern CPUs, to leak data directly across CPU cores and threads. Spectre and Meltdown were soon followed by many more speculative execution attacks, many of which proved costly to mitigate, requiring simultaneous multi-threading to be disabled. Other, older classes of microarchitectural attacks have also seen steady improvements in this time. Cache timing side-channel techniques have been expanded to meet increasingly large and complex cache systems. Machine learning techniques empower attackers to recover secret data out of noisier, more subtle channels. New strategies in crafting Rowhammer attacks,

which use rapid memory accesses to cause electrical faults, have thwarted memory vendors’ defenses time after time. As computer architectures and microarchitectures evolve, so too do the vulnerabilities present in them.

This work explores microarchitectural vulnerabilities with a focus on two important and rapidly changing areas of computing: heterogeneous computing and cloud computing. Heterogeneous computing systems use two or more different processors or other computing cores in tandem, allowing for more specialized hardware to perform certain tasks. Cloud systems—which are increasingly heterogeneous themselves—have shaken up the economics of server computing by offering a wide variety of computational resources for rent. This work introduces new microarchitectural vulnerabilities, threat models, and defenses native to these two paradigms and the many new and old technologies that underly them. Beyond presenting these examples, we aim to guide the reader in reconceptualizing microarchitectural security broadly to meet the realities of the present moment and anticipate the trends of the future. We emphasize the necessity of defense in depth against microarchitectural security, and hope that this work aids readers with a variety of backgrounds and interests in understanding and implementing secure systems.

1.1 Heterogeneous Computing in the Cloud

Modern server-grade computing infrastructures are becoming more heterogeneous: computational needs are spread over fast and flexible CPUs as well as

powerful peripherals such as smart storage, GPUs, smart NICs and FPGAs. CSPs like Amazon Web Services [8] and Alibaba Cloud [6] already offer FPGA instances with ultra-high performance Xilinx Virtex UltraScale+ and Intel Arria 10 GX FPGAs to the consumer market. Furthermore, major CSPs have started to shift tasks such as networking, memory management and VM management into more specialized hardware peripherals [7, 9, 48], freeing up precious CPU time that is rented to more tenants who share the same hardware. Multi-tenant, peripheral-heavy cloud systems rely on increasingly interlinked memory systems to provide high throughput for shared, scalable and parallelized cloud infrastructure. Technologies like VT-d, DDIO, and CXL allow peripherals to not only directly read and write to the memory of a virtual machine, but to also use a CPU’s shared cache to speed up repeated reads and writes. Among all heterogeneous computational devices, FPGAs are particularly interesting for cloud computing applications, as they can be reconfigured for the needs of different users at different times. These FPGAs are designed for high I/O bandwidth and high compute capacity, making them ideal for server workloads. New Intel FPGAs offer cache-coherent memory systems for even better performance when data is being passed back and forth between CPU and FPGA.

The flexibility of FPGA systems can also open up new attack vectors for malicious users in public clouds or more efficiently exploit existing ones. *Integrated FPGA platforms* connect the FPGA right into the processor bus interconnect giving the FPGA direct access into cache and memory [78].

Similarly, high-end FPGAs can be integrated into a server as an accelerator, e.g. connected via PCIe interface [85, 214]. Such combinations provide unprecedented performance over a high-throughput and low-latency connection with the versatility of a reprogrammable FPGA infrastructure shared among cloud users. However, the tight integration may also expose users to new adversarial threats.

On a logic layer, input-output memory management units (IOMMUs) enforce memory isolation between these peripherals and guest VMs running on CPUs, making IOMMUs a key component for ensuring security of the cloud infrastructure [15, 92, 129]. The IOMMU ensures that accesses to virtual memory spaces are isolated and appropriately virtualized: e.g., devices may handle only I/O-specific virtual addresses and not the CPU-side virtual addresses or the underlying system’s physical addresses; in addition, devices may only access memory with the appropriate permissions set.

However, when many tenants share the same hardware, side effects in these complex shared memory systems weaken the security promises of virtualization that make highly scalable multi-tenant cloud computing possible. These side effects of shared hardware are exploited by microarchitectural attacks, most prominently cache attacks. Cache attacks exploit the measurable difference in access times to the many tiers of modern caches to overcome the sophisticated memory isolation mechanisms that protect tenants’ data and computation from each other. Besides cache attacks, which have been successfully applied in commercial cloud settings [82, 131], microarchitectural

attacks like Meltdown [123] and related MDS attacks [29, 182, 201] as well as Rowhammer attacks pose a real threat in shared cloud environments. One malicious tenant may, after successful co-location [172, 223], use these microarchitectural side effects to glean sensitive information from co-located VMs.

While most research in microarchitectural attacks has focused on attacks from core to core on CPUs, caches are no longer only accessible by CPUs. Intel’s DDIO technology, present on all recent Intel server architectures, allows high speed peripherals to directly access a CPU’s shared cache without interrupting CPU execution [90]. Cloud users may rent peripherals such as purpose-specific GPU or FPGA cloud instances for higher performance in particular workloads. In such heterogeneous compute environments, security is even more challenging, as tenants are no longer confined to virtual machines (VMs) on the CPU, but may additionally have control over peripherals. With CSPs renting instances that grant tenants full access to FPGAs designed specifically for heterogeneous computation [6, 8, 55], it becomes trivial for attackers to gain sufficient control over peripherals in the cloud that are more than capable of exploiting microarchitectural vulnerabilities.

1.2 Serverless Cloud Computing

Outside of the world of heterogeneous computing, the CPU services offered by cloud providers are changing rapidly too, requiring new system infrastructure

and presenting new security challenges. Serverless computing is an emerging trend in cloud computing where CSPs serve runtime environments to their customers. This way, customers can focus on maintaining their function code while leaving the administrative work related to hardware, operating system (OS), and sometimes runtime to the CSPs. Since individual functions are typically small, CSPs aim for fitting as many functions on a single server as possible to minimize idle times and, in turn, maximize profit. This must be achieved while preserving security for both the provider and clients. Containers offer some isolation at the operating system level with relatively little performance overhead, but they are unable to take advantage of some of the powerful virtualization features of modern CPUs and OSs that provide deeper isolation at an architectural level. To use these features with performance suitable for serverless computing, developers slimmed down the virtual machines (VMs) used for heavier cloud computing, creating what are now called MicroVMs.

Firecracker [2] is a virtual machine monitor (VMM) designed to run microVMs with memory overhead and start times comparable to those of common container systems. Firecracker is actively developed by AWS and has been used in production for serverless compute services since 2018 [2]. AWS’s design paper [2] describes the features of Firecracker, how it diverges from more traditional virtual machines, and the intended isolation model that it provides: safety for *“multiple functions run[ning] on the same hardware, protected against privilege escalation, information disclosure, covert*

channels, and other risks” [2]. Furthermore, AWS provides production host setup recommendations [13] for securing parts of the CPU and kernel that a Firecracker VM interacts with. In this paper, we challenge the claim that Firecracker protects functions from covert and side-channels across microVMs. We show that Firecracker itself does not add to the microarchitectural attack countermeasures but fully relies on the host and guest Linux kernels and CPU firmware/microcode updates.

Microarchitectural attacks like the various Spectre [36, 77, 113, 116, 128, 208] and MDS [37, 138, 182, 201] variants pose a threat to multi-tenant systems as they are often able to bypass both software and architectural isolation boundaries, including those of VMs, when CPU core resources are shared. In serverless environments, resources are expected to be overcommitted, which leads to multiple functions competing for and sharing compute resources on the same hardware. This directly leads to an increased microarchitectural attack surface for serverless environments. The attack surface is the greatest if simultaneous multi-threading (SMT) is enabled—as is the compute power of a CPU, as SMT increases performance by up to 30% [130]. But even with SMT disabled, functions sharing compute resources in a time-sliced way remain vulnerable to some microarchitectural attacks.

1.3 Signature Correction Attacks

One powerful application of fault injection vulnerabilities like Rowhammer is signature correction attacks. Digital signature schemes, e.g. DSA, ECDSA, and RSA, are widely deployed to protect the integrity of security protocols like TLS, SSH, and IPSec. For instance, in TLS, RSA and ECDSA and DSA are used to sign the state of the agreed upon protocol parameters during the handshake phase. Consequently, DSA implementations have been targeted by powerful side-channel attacks, requiring cryptographic libraries to be patched repeatedly over the past two decades. Developers have come a long way in protecting against side-channel attacks, especially in hardening crypto libraries against timing attacks via so-called constant time implementations [99]. While most (basic) leakages have been fixed, cryptanalytic techniques for recovery have advanced to the point where extremely subtle leakages as small as a bit have been exploited to yield full key recovery [4, 19]. Hence, it becomes important to reassess cryptographic implementations with a more critical eye.

Signature Correction Algorithms (SCA) A particularly effective software-only technique to recover leakage from signature implementations was discovered by Mus et al. [144]. The attack works by injecting faults via Rowhammer and then tracing the fault to the faulty signature output, recovering a fraction of the internal secret bits in the process. A more enhanced version, called the signature correction attack, was developed by Islam et al. [98] to target CRYSTALS-Dilithium, the finalist of the NIST post-quantum

signature competition. The attack again injects faults during signing, and subsequently corrects faulty signatures to deduce the error patterns and subsequently the key bits of CRYSTALS-Dilithium. Most relevant to our work, is Jolt [143] where Mus et al. demonstrated that signature schemes in common cryptographic libraries are still vulnerable to software-only fault injection attacks. They employed Rowhammer and showed how **signature correction** can be adapted to work with (EC)DSA, Schnorr and RSA signatures to achieve full key recovery. Jolt takes advantage of the fact that the private key (unlike the nonce), which remains unchanged across sessions, slowly recovering scattered bits. The remaining bits are recovered via a costly computation of a modified version of Shank’s Algorithm. For instance, if all but 100 bits are recovered the cost of Shank’s Algorithm is 2^{50} in time and space. Hence for Jolt to be practical most bits must be recovered via Signature Correction.

Lattice Based Cryptanalysis and the Hidden Number Problem

Lattice reduction algorithms first found their uses in cryptanalysis [53, 54, 121, 179, 180] Later lattice-based attacks include Coppersmith’s approach, factoring RSA keys with a partial understanding of the secret key [147, 153] and side-channel attacks where lattices are used to solve the Hidden Number Problem (HNP) and break (EC)DSA and Diffie-Hellman. [3, 5, 24, 43, 74]. Depending on key size and the number of leaked nonce bits, a HNP can be formulated as a Closest Vector Problem (CVP). Lattice-based approaches, e.g. [127], [149], [139] enable effective key recovery for modest signature

sizes, such as 2-bit leakage for 160-bit signatures and at least 4-bit leakage for 256-bit signatures, with only a few hundred faulty signatures.

1.4 Contributions

This work presents a number of novel contributions to the fields of microarchitectural security, heterogeneous computing, cloud security, and fault injection attacks. Chapter 4, JackHammer, introduces a hardware Rowhammer design for FPGAs that runs twice as fast as a comparable CPU attack and causes faults that CPU Rowhammer cannot replicate. We tested JackHammer in a fault injection attack against a newly discovered vulnerability in WolfSSL’s implementation of RSA (CVE-2019-19962 [134]) and found that it performs up to 200% faster than CPU Rowhammer with common countermeasures employed. This section also presents the first evidence of cache timing side-channels in heterogeneous FPGA-CPU systems and a reliable FPGA to CPU covert channel. Chapter 5, IOTLB-SC, demonstrates the first microarchitectural vulnerability internal to IOMMUs—a timing side-channel in shared translation look-aside buffers, analyzes the threat this channel poses, and measures the performance of the channel with a GPU to FPGA covert channel. Chapter 6, Microarchitectural Vulnerabilities in AWS Firecracker, provides a comprehensive analysis of the security features, system setup recommendations, and threat model of AWS’s MicroVM, Firecracker. We test Firecracker’s defenses against a wide range of microarchitectural attack proof

of concepts in multiple major classes of attacks and find that Firecracker provides negligible defenses in most cases, and in one case even opens a new vulnerability. We also identify poorly-mitigated Spectre variants of particular concern. We reported our findings to AWS along with our concerns about Firecracker’s documentation. Following our disclosure, AWS updated its security documentation to emphasize that Firecracker itself does not protect against microarchitectural attacks and direct system administrators to the CPU and OS vendor websites that provide full documentation of available system level microarchitectural mitigations and the most up-to-date microarchitectural security advisories and recommendations. These contributions are elaborated in more detail at the start of each section.

1.5 Previous Publications and Coauthors

I have published most of the contributions of this work in peer reviewed journals, and most of this work is comprised of portions of my previously written papers, the names of which correspond to the names of chapters 2–5. JackHammer was coauthored with Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar and published in TCHES 2020 [207] IOTLB-SC was coauthored with Thore Tiemann, Thomas Eisenbarth, and Berk Sunar and published in AsiaCCS 2023 [193] Contributions from JackHammer and IOTLB-SC were also included in a chapter for *Security of FPGA-Accelerated Cloud Computing Environments*, edited by Jakub Szefer

and Russell Tessier and published by Springer [194]. The work on Firecracker, coauthored with Thore Tiemann, Thomas Eisenbarth, and Berk Sunar is presently available in pre-print on arXiv [205]. DoubleHammer is unpublished at the time of writing, but the writing in this dissertation is drawn from a draft coauthored with Kristi Rahman and Berk Sunar.

Chapter 2

Background

2.1 Cache Attacks

Cache attacks have been proposed attacking various applications [23, 32, 67, 69, 154, 197]. In general, cache attacks use timing of the cache behavior to leak information. Modern cache systems use a hierarchical architecture that includes smaller, faster caches and bigger, slower caches. Measuring the latency of a memory access can often confidently determine which levels of cache contain a certain memory address (or if the memory is cached at all). Many modern cache subsystems also support coherency, which ensures that whenever memory is overwritten in one cache, copies of that memory in other caches are either updated or invalidated. Cache coherency may allow an attacker to learn about a cache line that is not even directly accessible [95]. Cache attacks have become a major focus of security research

in cloud computing platforms where users are allocated CPUs, cores, or virtual machines which, in theory, should offer perfect isolation, but in practice may leak information to each other via shared caches [82]. Timing side-channel attacks against the CPU’s cache are widely studied and well understood: researchers have crafted several variants [45, 66, 126, 164, 217], used them as part of more complicated microarchitectural attacks [113, 123], and built defenses against them [61, 125, 219]. An introduction to various cache attack techniques is given below:

Flush+Reload Flush+Reload (F+R) [217] has three steps: **1)** The attacker uses the `clflush` instruction to flush the cache line that is to be monitored. After flushing this cache line, **2)** she waits for the victim to execute. Later, **3)** she reloads the flushed line and measures the reload latency. If the latency is low, the cache line was served from the cache hierarchy, so the cache line was accessed by the victim. If the access latency is high, the cache line was loaded from main memory, meaning that the victim did not access it. F+R can work across cores and even across sockets, as long as the LLC is coherent, as is the case with many modern multi-CPU systems. Flush+Flush (F+F) [66] is similar to F+R, but the third step is different: the attacker flushes the cache line again and measures the execution time of the flush instruction instead of the memory access.

Orthogonal to F+R, if the attacker does not have access to an instruction to flush a cache line, she can instead evict the desired cache line by accessing

cache lines that form an eviction set in an Evict+Reload (E+R) [122] attack. Eviction sets are described shortly. E+R can be used if the attacker shares the same CPU socket (but not necessarily the same core) as the victim and if the last-level cache (LLC) is inclusive.¹ F+R, F+F, and E+R are limited to shared memory scenarios, where the victim and attacker share data or instructions, e.g. when memory de-duplication is enabled.

Prime+Probe Prime+Probe (P+P) gives the attacker less temporal resolution than the aforementioned methods since the attacker checks the status of the cache by probing a whole cache set rather than flushing or reloading a single line. However, this resolution is sufficient in many cases [93, 122, 135, 154, 155, 172, 222]. P+P has three steps: **1)** The attacker primes the cache set under surveillance with dummy data by accessing a proper eviction set, **2)** she waits for the victim to execute, **3)** she accesses the eviction set again and measures the access latency (probing). If the latency is above a certain threshold, some parts of the eviction set was evicted by the victim process, meaning that the victim accessed cache lines belonging to the cache set under surveillance [126]. Unlike F+R, E+R, and F+F, P+P does not rely on shared memory. However, it is noisier, only works if the victim is located on the same socket as the attacker, and relies on inclusive caches. An alternative attack against non-inclusive caches is to target the cache directory structure [216].

¹A lower-level cache is called inclusive of a higher-level cache if all cache lines present in the higher-level cache are always present in the lower-level cache.

Evict+Time In scenarios where the attacker can not probe the target cache set or line, but she can still influence the target cache line, an Evict+ Time (E+T) is still possible depending on the target application. In an E+T attack, the attacker only evicts the victim’s cache line and measures the aggregate execution time of the victim’s operation, hoping to observe a correlation between the execution time of an operation such as a cryptographic routine and the cache access pattern.

Eviction Sets Caches store data in units of cache lines that can hold 2^b bytes each (64 bytes on Intel CPUs). Caches are divided into 2^s sets, each capable of holding w cache lines. w is called the way-ness or associativity of the cache. An eviction set is a set of congruent cache line addresses capable of filling a whole cache set. Two cache lines are considered congruent if they belong to the same cache set. Memory addresses are mapped to cache sets depending on the s bits of the physical memory address directly following the b cache line offset bits, which are the least significant bits. Additionally, some caches are divided into n slices, where n is the number of CPU cores. In the presence of slices, each slice has 2^s sets with w ways each. Previous work has reverse-engineered the mapping of physical address bits to cache slices on some Intel processors [94]. A minimal eviction set contains w addresses and therefore fills an entire cache set when accessed.

2.2 Translation Look-Aside Buffers TLBs

While a cache stores data for faster access, a translation look-aside buffer (TLB) is technically just another cache, though rather than caching the data or instructions stored at an address, it caches an address translation. However, throughout this paper we will refer to memory caches as simply “caches.” Intel’s documentation [91] and several works reverse-engineering cache architectures [79,94,126,154] and TLB architectures [58,192] reveal that TLBs on modern Intel CPUs are organized very similarly to modern CPU memory caches. Modern TLBs and caches are typically organized into **sets** and **ways**. The number of ways is the number of entries each set can contain. For TLBs, each virtual address is mapped to one set, but can occupy any way within that set. When a set is full, old entries may be evicted to make room for new ones. A set of addresses which reliably causes the eviction of all other entries in a set when accessed is called an **eviction set**. A minimal eviction set contains as many addresses as there are ways in the cache/TLB and therefore fills an entire cache set when accessed [203].

2.3 Attacks on TLBs

In 1995, Silbert et al. remarked in a security analysis of Intel CPU architectures that “all 80x86 [now more commonly called x86] processors have a translation look-aside buffer (TLB) that ... has potential for use as a covert timing channel” [188]. In 2013, Hund et al. [79] demonstrated that a TLB

timing side-channel on then-modern Intel CPUs could reveal if a page was mapped by the operating system even if the user does not have permission to access the page directly. They demonstrated that this exploit could be used to identify the pages used by the kernel, even when the addresses of the pages were randomized (a common defense against side-channel attacks of many types). In 2017, Gras et al. crafted an attack that uses a cache side-channel to identify TLB evictions. This was a robust attack that can be mounted even from JavaScript to de-randomize kernel pages [59]. Gras et al.’s “TLBleed” in 2018 [58] showed that TLBs in modern Intel CPUs were vulnerable to timing side-channel attacks of the sort that are typically used on CPU memory caches, and can be used for similarly complex attacks: with the help of some machine learning, the TLB side-channels on Skylake, Broadwell, and Coffeelake CPUs can be used to recover a key from an Edward-curve cryptographic function.

2.4 PCIe

Peripheral Component Interconnect Express (PCIe) [159] is the backbone of modern desktop and server systems. While often referred to as a bus, PCIe uses a high-speed point-to-point topology with devices being connected to switches or directly to a root port via serial links. The root complex connects the PCIe network to the CPU and the main memory. On a PCIe network, all devices can send memory requests to each other and to the main memory.

An IOMMU can be used to virtualize addresses used by PCIe devices and to implement access restrictions. If supported, each root port of a root complex may define access rules for inter-device communication and implement them in the PCIe switches.

Two recent works [106, 189] describe covert- and side-channel attacks that rely on PCIe bus contention. A preliminary is that the two devices involved share the same PCIe switch. In contrast, our work assumes the two devices to share a PCIe root port. Our assumption is less restrictive as any two PCIe devices sharing a switch share a root port, but devices sharing a root port do not necessarily share a switch, as root ports can have many lanes to support multiple devices without sharing a physical bus [159].

Currently, PCIe 3.0 is the prevailing PCIe specification for commodity hardware. After a short period of CPUs supporting PCIe 4.0, PCIe specification 5.0 is the upcoming standard for the next generations of server-grade CPUs. CPUs supporting PCIe 5.0 are scheduled for November 2022 and January 2023, respectively [16, 186]. PCIe 5.0 doubles transfer rates compared to PCIe 4.0, making the interconnect compete with main memory speeds. As a result, PCIe 5.0 physical layer is also used by a new protocol named Compute Express Link (CXL) [185]. CXL supports three sub-protocols: *CXL.io* is based on PCIe and enables CXL devices to share the PCIe infrastructure with PCIe devices unaware of CXL. With *CXL.cache*, devices are enabled to cache data from main memory while maintaining coherency between the main memory, the CPU caches and the accelerator cached copy. *CXL.mem* is

used by a host CPU to access CXL device memory and manage its coherent usage.

2.5 IOMMUs

Input-Output Memory Management Units (IOMMUs) are located between PCIe devices and the main memory. Usually, they are implemented as part of the root complex. Modern server systems feature one IOMMU per root port. Similar to MMUs in the CPU, IOMMUs provide address translation and protection for memory regions that are made accessible to PCIe devices [15,92]. Address virtualization allows to isolate or virtualize such devices. Also, it allows 32-bit peripherals to use memory regions above 4 GB.

The translation process of the IOMMU works very similar to the process in a CPU's MMU. Modern IOMMUs map PCIe devices to IOMMU groups or domains. The operating system, hypervisor, or VMM maintains a page table with all address mappings per group/domain. The page table is organized in a tree structure. Its depth depends on the width of the I/O virtual addresses (IOVAs) supported by the IOMMU. For IOVAs referencing 4 KB pages, the 12 least significant address bits (page offset) remain untranslated. Accordingly, the 21/30 least significant bits of IOVAs pointing to 2 MB/1 GB pages remain untranslated.

IOVAs are translated to physical addresses (PAs) by the IOMMU performing a page table walk. Since this is quite time consuming, modern IOMMUs

feature a translation look-aside buffer called IOTLB. This cache is used to store translated IOVA→PA mappings and is shared by all devices managed by the IOMMU.

2.6 Rowhammer

In a Rowhammer attack, rapid accesses to carefully chosen memory addresses cause a memory fault of a single bit (referred to as a bit flip) in memory that was *not* directly accessed. Synchronous Dynamic Random-Access Memory (SDRAM) interfaces like DDR3, DDR4, and DDR5 are designed around the concept of *memory rows*, the smallest unit of memory that is loaded at one time from the main storage chips on the memory module. As silicon manufacturing techniques have improved, the memory cells that make up each row run at lower voltages and are smaller and more densely packed, improving speed, reducing power consumption, and increasing available memory size. However, these features also reduce the reliability of the memory. Lower voltage circuits have smaller margins of error between the encodings of “0” and “1” and smaller electrical components are prone to variation in the manufacturing process. Most importantly, high density between cells increases electromagnetic crosstalk, which is the fundamental reason for the Rowhammer error. Under normal operation, the effect of row-to-row crosstalk is not enough to cause faults, but rapid, repeated accesses can drain or charge nearby memory cells enough to flip a bit from 0 to 1 or 1 to 0. [145]

Since the first widely released findings in 2014, Rowhammer faults have been observed on DDR3 [109] and DDR4 [51, 63], on mobile platforms [199], and on SDRAMs with Error-Correcting Codes [42]. Rowhammer attacks have been mounted from JavaScript inside a web browser [64], across networks [124, 191], and from FPGAs [206]. Researchers have demonstrated a wide variety of practical exploits based on Rowhammer, including privilege escalation [63, 184], cross-virtual-machine attacks [213], and fault injection attacks against machine learning models [195] and cryptographic schemes [25, 97, 143, 144].

Multi-Sided Rowhammer

The first widely distributed research on Rowhammer [109] observed bit flips triggered by alternating memory accesses to two addresses in two rows, now referred to as a “double-sided” Rowhammer attack. However, double-sided Rowhammer attacks are mostly prevented by target row refresh (TRR), a broad class of Rowhammer defenses implemented in the majority of DDR4 memory modules and some memory controllers. TRR aims to identify rows that are targeted by a Rowhammer attack in real time and refresh their values before a fault can occur. Later Rowhammer variants have bypassed TRR detection mechanisms by increasing the number of addresses or “sides” and complicating the access patterns (in both order and timing) [44, 51, 100]. Since TRR implementations vary, much of the later Rowhammer research has focused on methods for dynamically finding effective hammering patterns.

Finding Rowhammer Targets

Since Rowhammer attacks rely on the physical layout of a system’s memory, techniques for reverse engineering properties of physical address layouts greatly assist attackers in finding successful attack locations. SPOILER [96] showed that row conflicts in the DRAM can be detected through timing side-channel and used to quickly and reliably find large blocks of memory with contiguous physical addresses without root access or ever directly viewing the addresses themselves. This allows an attacker to select pages with nearby physical addresses, which are in turn likely to be stored near to each other in the DRAM chip, a necessary condition for a Rowhammer attack. Pessl et al. demonstrated how the DRAMA [162] row conflict side channel can be used to reverse-engineer the mappings from physical addresses to the channels, ranks, banks, columns, and rows that make up the organization of memory at the silicon level.

To use Rowhammer for a practical attack such as privilege escalation within an operating system or fault injection against a cryptographic algorithm, a seemingly random flip generated by the bug must occur in a very particular location in memory. The first technique to achieve this reliably, published by Seaborn et al. [184], is called page-table spraying or simply page spraying. There are four main steps to this method:

1. The attacker allocates a large amount of memory and searches it for a target page with flip(s) occurring at a specific *page offset* to flip the

intended bit(s). The address(es) used to hammer this page are also noted.

2. The attacker deallocates the page that had flips at the right offset.
3. The attacker causes the allocation of many pages containing the targeted memory. With the target page recently deallocated, the probability that it will be refilled with the targeted memory values is high.
4. The attacker uses the previously found address(es) to hammer the target page.

The major limitations of this method are that the attacker must know the offset of the target bit, and must be able to cause the victim to allocate pages. In the case of Seaborn’s attack, which targets privilege control bits in page table entries, the layout of page table entries is publicly known, and simply allocating pages of memory allocates page table entries as well. The advantage of page spraying is its flexibility across architectures, operating systems, and runtime environments. Other techniques rely on more particular memory features, like deduplication [168], which can be disabled for security, or specific deterministic memory allocation algorithms [199], which are not present on all systems.

2.7 Meltdown and MDS

In 2018, the Meltdown [123] attack showed that speculative execution could access data across security boundaries and encode it into a cache side-channel. This soon led to a whole class of similar attacks, known as microarchitectural data sampling (MDS), including Fallout [37], Rogue In-flight Data Load (RIDL) [201], TSX Asynchronous Abort (TAA) [201], and Zombieload [182]. These attacks all follow the same general pattern to exploit speculative execution:

1. The victim handles secret data that passes through a cache or CPU buffer.
2. The attacker executes a specifically chosen instruction that causes the CPU to speculatively forward the secret to the attacker’s instruction.
3. The attacker preserves the transiently learned secret by sending it through a covert channel.

The original Meltdown vulnerability targeted cache forwarding and allowed data extraction in this manner from *any* memory address that was present in the cache. Newer MDS attacks target specific buffers in the on-core microarchitecture and work under more specific timing and co-location conditions.

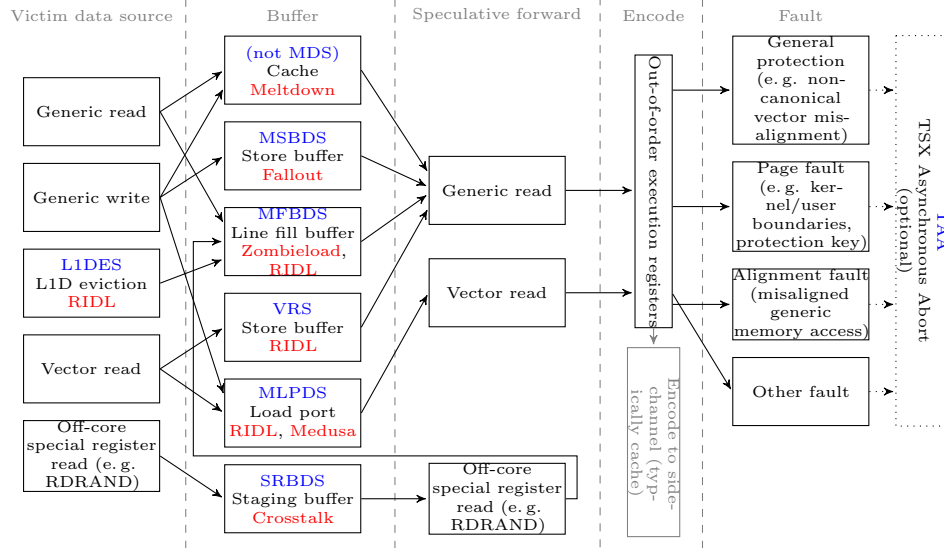


Figure 2.1: Major MDS attack pathways and variant names on Intel CPUs. The blue names at the top are names of vulnerabilities given by Intel; the red names at the bottom are names given by researchers or the names of the papers in which the vulnerabilities were reported. Not all fault types work with all vulnerabilities on all systems—cataloging every known combination would be beyond the scope of this paper.

2.7.1 Basic MDS Variants

Figure 2.1 charts the major known MDS attack pathways on Intel CPUs and the names given to different variants by Intel and by the researchers who reported them. Intel tends to categorize MDS vulnerabilities in their CPUs by the specific buffer from which data is speculatively forwarded, since these buffers tend to be used for a number of different operations. Intel names in this style include Microarchitectural Load Port Data Sampling (MLPDS), Microarchitectural Fill Buffer Data Sampling (MFBDS), Microarchitectural Store Buffer Data Sampling (MSBDS), Special Register Buffer Data Sampling

(SRBDS), and Vector Register Sampling (VRS). L1 Data Eviction Sampling (L1DES) and TSX Asynchronous Abort (TAA) describe closely related vulnerabilities. L1DES is a method by which the attacker forces the victim's data from the cache into the buffer from which it is eventually leaked. TAA, described in detail in section 2.7.3, is an alternate method for triggering speculative execution.

2.7.2 Medusa

Medusa [138] is a category of MDS attacks classified by Intel as MLPDS variants [88]. The Medusa vulnerabilities exploit the pattern-matching algorithms used to *speculatively* combine stores in the write-combine (WC) buffer (part of the load port) of Intel processors. There are three known Medusa variants, each exploiting a different feature of the WC buffer to cause a speculative leakage:

Cache Indexing: a faulting load is speculatively combined with an earlier load with a matching cache line offset.

Unaligned Store-to-Load Forwarding: a valid store followed by a dependent load that triggers an misaligned memory fault causes random data from the WC buffer to be forwarded.

Shadow REP MOV: a faulting REP MOV instruction followed by a dependent load leaks the data of a different REP MOV.

2.7.3 TSX Asynchronous Abort

The hardware vulnerability TSX Asynchronous Abort (TAA) [87] is a speculation mechanism for carrying out an MDS attack. While standard MDS attacks access restricted data with a standard speculated execution, TAA uses an atomic memory transaction as implemented by TSX. When an atomic memory transaction encounters an asynchronous abort due to a fault, the architectural state is "rolled back" to its state before the transaction started. During this rollback, instructions that have started speculatively executing can continue to do so, as in steps (2) and (3) of other MDS attacks. TAA impacts all Intel processors that support TSX, and the case of newer processors that are not affected by other MDS attacks, MDS mitigations or TAA-specific mitigations (such as disabling TSX) are required for protection against TAA [87].

2.7.4 MDS Mitigations

Though Meltdown and MDS-class vulnerabilities exploit low level microarchitectural operations, they can be mitigated with microcode firmware patches on most vulnerable CPUs.

Page table isolation Historically, kernel page tables have been included in user-level process page tables so that a user-level process can make a system call to the kernel with minimal overhead. Page table isolation (first proposed by Gruss et al. as KAISER [62]) maps only the bare minimum necessary

kernel memory into the user page table and introduces a second page table only accessible by the kernel. With the user process unable to access the kernel page table, accesses to the rest of kernel memory are stopped before they reach the lower level caches where a Meltdown attack begins. Page table isolation works best when layered on top of kernel Address Space Layout Randomization (ASLR), which randomizes the layout of kernel memory on each boot.

Buffer overwrite MDS attacks that target on-core CPU buffers require a lower-level and more targeted defense. Intel introduced a microcode update that overwrites vulnerable buffers when the first-level data (L1d) cache (a common target of cache timing side-channel attacks) is flushed or the `VERW` instruction is run [88]. The kernel can then protect against MDS attacks by triggering a buffer overwrite when switching to an untrusted process.

The buffer overwrite mitigation targets MDS attacks at their source, but is imperfect. Processes remain vulnerable to attacks from concurrently running threads on the same core when SMT is enabled (since both threads share vulnerable buffers without the active process actually changing on either thread). On some Skylake CPUs, buffers are overwritten with stale data [201], and remain vulnerable even with mitigations enabled and SMT disabled. Still other processors are vulnerable to TAA but not non-TAA MDS attacks, and did not receive a buffer overwrite microcode update. On these CPUs, TSX must be disabled to prevent MDS attacks [71, 87].

2.8 Spectre

In 2018, Jan Horn and Paul Kocher [113] independently reported the first Spectre variants. Since then, many different Spectre variants [77, 113, 116, 128] and sub-variants [36, 111, 208] have been discovered. Spectre attacks make the CPU speculatively access memory that would not be accessed architecturally and leak the data into the architectural state. Therefore, all Spectre variants consist of three components [102]:

The first component is the Spectre gadget that the CPU executes speculatively. Spectre variants are commonly distinguished by the source of the misprediction they exploit. Spectre-PHT—which allows for speculative bounds check bypass—results from the prediction outcome by the pattern history table (PHT) for conditional branches [36, 111, 113]. The branch target buffer (BTB) predictions of branch targets for indirect jumps allows for speculative return-oriented programming attacks through Spectre-BTB [36, 113], Spectre-RSB exploits the return address prediction by the return stack buffer (RSB) [36, 116, 128]. Spectre-STL [77] exploits store-to-load (STL) dependency prediction to read stale data or provoke transient buffer overflows.

The second component is the attacker’s control over the gadgets. Control may be possible through user input, file contents, or other architectural mechanisms. Additionally, transient mechanisms like LVI [34] or floating point value injection [166] may allow an attacker the necessary control over accessed data or executed instructions.

The third component is the covert channel that transfers the transient microarchitectural state into an architectural state to exfiltrate data. Cache covert channels [156, 164, 218] are the most prominent candidates but Spectre attacks using MDS [37, 182, 201] or port contention [52, 174, 175] are also known in the literature.

2.8.1 Spectre Mitigations

Many countermeasures are discussed in the literature. Early countermeasures target the availability or accuracy of covert channels [107, 110, 113, 215], though, such countermeasures tend to be incomplete due to the numerous covert channels that may be used. Countermeasures that focus on removing the attacker’s control over the prediction outcome are more promising and used today. Spectre-BTB is mitigated by Retpoline [198] or microcode updates like IBRS, STIBP, and IBPB [86]. Spectre-RSB can be mitigated through RSB filling or the IBRS microcode update and the SSBD [86] microcode update protects against Spectre-STL. Disabling SMT partitions branch prediction hardware between concurrent tenants but implies a significant performance penalty and still allows sequential tenants to share the branch prediction unit.

2.9 RSA-CRT Signing

RSA signatures are computed by raising a plaintext m to a secret power d modulo $N = pq$, where p and q are prime and secret, and N is public [27].

These numbers must all be large for RSA to be secure, which makes the exponentiation rather slow. However, there is an algebraic shortcut for modular exponentiation: the Chinese Remainder Theorem (CRT), used in many RSA implementations, including in the WolfSSL we attack in section 4.4 and in OpenSSL [39]. The basic form of the RSA-CRT signature algorithm is shown in algorithm 1. The CRT algorithm is much faster than simply computing $m^d \bmod N$ because d_p and d_q are of order p and q respectively while d is of order N , which, being the product of p and q , is significantly greater than p or q ; it is around four times faster to compute the two exponentiations m^{d_p} and m^{d_q} than it is to compute m^d outright [20].

```

1 Function sign( $m, d, p, q$ )
   input :  $m$  – message to be signed;  $d$  – private exponent;  $p$  –
           private factor;  $q$  – private factor
   output :  $S$  – signature
2    $S_p \leftarrow m^{d_p} \bmod p$  // equivalent to  $m^d \bmod p$ 
3    $S_q \leftarrow m^{d_q} \bmod q$  // equivalent to  $m^d \bmod q$ 
4    $I_q \leftarrow q^{-1} \bmod p$  // inverse of  $q$ 
5   return  $S \leftarrow S_q + q((S_p - S_q)I_q \bmod p)$ 

```

Algorithm 1: Chinese remainder theorem RSA signature.

2.10 ECDSA Signing

The US 1994 NIST standard Digital Signature Algorithm (DSA) is based on the ElGamal Signature Scheme [47]. The Elliptic Curve Digital Signature Algorithm (ECDSA) is an algorithmic conversion of a step from the multiplicative group of a finite field to the group of points on an elliptic curve.

While DSA has been officially phased out by NIST [152], ECDSA is a common digital signature algorithm [17, 103, 105] employed in blockchain applications, the TLS and SSH protocols [170], and document signing, among other things. The best known algorithms for solving the discrete logarithm problem in the finite field are currently sub-exponential, while those for solving the elliptic curve discrete logarithm problem (ECDLP) are exponentially complex, so using the elliptic curve group as opposed to the multiplicative group of a finite field allows for the use of smaller parameters while still maintaining the same security level [112, 133]. More details can be found in [1, 22].

Despite being one of the most widely used signature schemes now in use, ECDSA has a number of implementation issues, particularly because the random value nonce created as part of the signing method is extremely sensitive. The scalar multiplication of a point on the elliptic curve by a secret nonce that is created pseudo-randomly is one of the most fundamental operations of the ECDSA algorithm. The nonce’s confidentiality is crucial to the algorithm’s security. Previous studies show that attacks on the secret key can be effectively used to take advantage of nonce bits’ partial exposure [33, 151]. As was the situation with the PlayStation 3 gaming system, which used a fixed value for signing its binaries, every reuse of the nonce for a different message trivially results in key recovery. However, there are other types of bias that has been proven to make an ECDSA key vulnerable than repeated nonce values. In fact, if there are enough signatures, a variety of ECDSA signature nonuniformities can actually reveal the secret key. However,

it is now possible to recover the secret key even if short bit substrings of the nonces are compromised by expanding this straightforward observation, according to cryptanalysts.

2.11 Lattice Attacks on the Hidden Number Problem (HNP)

Boneh and Venkatesan [28] introduced the HNP in order to study the bit security of the Diffie-Hellman scheme. For a secret d and public modulus n we are given samples $k_i = t_i d \pmod{n}$ for $0 \leq k_i < n$ for uniformly and randomly chosen integers $t_i \in \mathbb{Z}_n^*$. Boneh and Venkatesan showed how to recover the secret integer d in polynomial time using lattice-based algorithms, if the attacker learns sufficiently many samples from the most significant ℓ bits of t_i . This problem can be formulated as a variant of the Closest Vector Problem (CVP) called Bounded Distance Decoding (BDD). BDD works by finding the closest vector in a lattice according to some target point t . This close vector can be found through lattice reduction, and using this close vector the secret parameter is recovered. The constraints of solving the secret lies in the uniqueness of the vector.

Formulating Biased ECDSA Samples as HNP If information on nonces is leaked, e.g. through a side-channel, one may formulate the ECDSA signature key recovery problem as a HNP. Here we closely follow the notation given

in [4]. Assume we are given a signature sample $s = k^{-1}(H(M) + dr) \bmod n$ where (r, s) is the signature, k is the biased nonce, $H(M)$ denotes the message hash, d is the secret key, and $r = (kP)_x$, i.e. the x coordinate of the random point kP . Reformulating the signature s we obtain

$$k - s^{-1}rd - s^{-1}H(m) = 0 \bmod n$$

Assume we are given m such signature samples. Relabelling $a = -s^{-1}r$, and $t = s^{-1}H(m)$, we end up with a system of m equations with $m + 1$ unknowns k_i and d . We can eliminate the unknown d by simply taking a sample, e.g. $a_0 + k_0 = t_0d$ and by scaling with an appropriate multiple, i.e. $t_0^{-1}t_i$ and subtracting it from each sample: $(a_i + k_i) - t_0^{-1}t_i(a_0 + k_0) = t_id - t_0^{-1}t_it_0d \pmod n$. Hence, our updated parameters become $a'_i = a_i - t_0^{-1}t_ia_0 \pmod n$, and $t'_i = t_it_0^{-1}$

Assume the nonces are bounded: $k_i < K < n$. We can now define a lattice

by reformulating m signature samples: $k_i + t_i = a_i d \bmod n$ as follows

$$\Lambda = \begin{bmatrix} n & & & & & \\ & n & & & & \\ & & n & & & \\ & & & \ddots & & \\ & & & & n & \\ t'_1 & t'_2 & t'_3 & \dots & t'_{m-1} & 1 \\ a'_1 & a'_2 & a'_3 & \dots & a'_{m-1} & K \end{bmatrix}$$

The rows of Λ form a lattice in which by construction $\mathbf{k} = (k_1, k_2, \dots, k_m, K)$ is a short vector. Finding \mathbf{k} , we can recover the secret signing key $d = -t_i^{-1}(k_i + b_i) \bmod n$.

The Lattice Barrier The basic form of the attack is effective as long as a BDD solver can recover the target vector from Λ . The BDD solver is expected to succeed as long as $\|\mathbf{k}\|_2 = \sqrt{m+1}K$ is less than the Gaussian Heuristic $gh(\Lambda) \approx \sqrt{\dim \Lambda / (2\pi e)} \text{Vol}(\Lambda)^{1/\dim \Lambda}$. Here $\text{Vol}(\Lambda) = n^{m-1}K$. Hence,

$$gh(\Lambda) \approx \sqrt{(m+1)/(2\pi e)} \text{Vol}(\Lambda)^{1/(m+1)}$$

$$= \sqrt{(m+1)/(2\pi e)} (n^{m-1}K)^{1/(m+1)}$$

When the leakage (or nonce bias) is high the condition will hold and given sufficient samples the BDD solver will recover the nonce vector. However, when the leakage is limited to a single bit then the condition becomes hard to satisfy and lattice based techniques are expected to fail with high probability, given that the secret vector is no longer significantly shorter than the other lattice vectors [19]. This view motivated a hard limit, the so-called “lattice barrier” that seems impossible to overcome for single bit leakage [18]. This belief extends to 2-bit biases, and even 3-bit biased HNPs are considered hard to tackle regardless of the number of samples.

BDD with Predicate Albrecht and Heninger [4] introduced several optimizations to bridge the lattice barrier. First they note that the upper bound norm estimate on the secret vector is too conservative and instead they use the expected norm of a uniformly distributed vector. The second observation of they make is that the lattice barrier can be overcome. Even if $\|\mathbf{k}\| \geq gh(\Lambda)$, then it is possible to recover \mathbf{k} by spending additional computation time. To this end, the authors introduce the unique-SVP with predicate problem we are seeking for a short vector v , that also satisfies a predicate function $f(v) = 1$. The authors proposed two algorithms to solve the unique-SVP with predicate problem: one based on enumeration and one based on sieving. The algorithms were implemented by modifying the `fpLLL` and `G6k` libraries. Running extensive experiments they were able to show that indeed one can

use efficient lattice based techniques to target cases with fewer than 4-bit nonce bias, and most notably, the two bit nonce bias for 256-bits is within reach.

2.12 Bleichenbacher’s Fourier Analysis Based Technique

Aranha et al. [18] employed the FFT approach to attack 160-bit ECDSA where k is 1-bit biased. They can succeed in retrieving the secret key with 2^{33} signatures in about 2^{37} time and with 2^{33} memory complexity. To recover a 256-bit ECDSA key d , they estimate that HNP can be solved with 2^{52} signatures with 1-bit leakage. A more recent work [19] takes advantage of leakage in the Montgomery ladder implementation of ECDSA for FFT based key recovery with fewer signatures. They estimate that 2^{20} and 2^{45} signatures, for 160-bit and 256-bit curves, respectively, will be required in the best scenario with 1 bit nonce leakage. The authors also estimate that around 2^{10} signatures with 3 MSB bits leakage are needed for full secret key recovery.

2.13 Serverless Cloud Computing and Micro-VMs

An increasingly popular model for cloud computing is serverless computing, in which the CSP manages scalability and availability of the servers that run the cloud user’s code. The CSP can manage its users’ workloads however it pleases, optimize for minimal operating cost, and implement flexible pricing where users pay for the execution time and memory that they use. The user does not need to worry about server infrastructure design or management, and so reduces the costs of development and maintenance work.

Serverless providers use a variety of systems to manage running functions and containers. Container systems like Docker, Podman, and LXD provide a convenient and lightweight way to package and run sandboxed applications in any environment. However, compared to the virtual machines used for many more traditional forms of cloud computing, containers offer less isolation and therefore less security. In recent years, major CSPs have introduced microVMs that back traditional containers with lightweight virtualization for extra security [2, 220]. The efficiency of hardware virtualization with kernel-based virtual machine (KVM) and lightweight design of microVMs means that code in virtualized, containerized or container-like systems can run nearly as fast as unvirtualized code and with comparable overhead to a traditional container.

2.14 AWS Firecracker

Firecracker [2] is a microVM developed by AWS to isolate workloads on its serverless platforms. As a microVM, Firecracker sacrifices hardware, OS, and I/O flexibility to be very light-weight in the size of its code base and in-memory overhead, as well as very quick to boot or shut down. In their paper, AWS itemizes the design requirements for the isolation system that eventually became Firecracker as follows [2]:

Isolation: It must be safe for multiple functions to run on the same hardware, protected against privilege escalation, information disclosure, covert channels, and other risks.

Overhead and Density: It must be possible to run thousands of functions on a single machine, with minimal waste.

Performance: Functions must perform similarly to running natively. Performance must also be consistent, and isolated from the behavior of neighbors on the same hardware.

Compatibility: AWS Lambda [10] allows functions to contain arbitrary Linux binaries and libraries. These must be supported without code changes or recompilation.

Fast Switching: It must be possible to start new functions and clean up old functions quickly.

Soft Allocation: It must be possible to over commit CPU, memory and other resources, with each function consuming only the

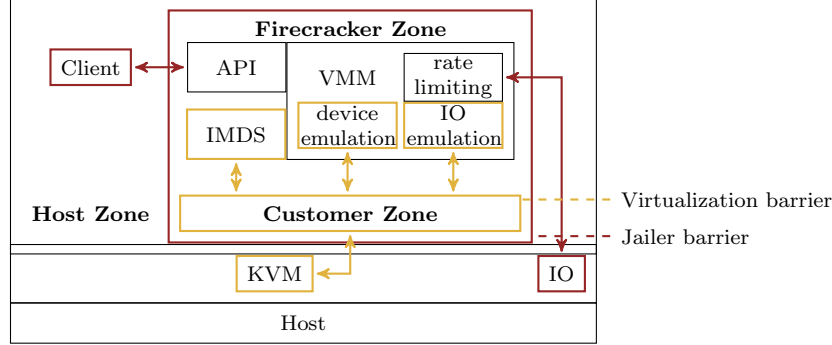


Figure 2.2: Firecracker threat containment diagram—adapted from [11]. The jailer provides container-like protections for components running in the host user space. The customer workload runs inside a virtual machine.

resources it needs, not the resources it is entitled to.

We are particularly interested in the *isolation* requirement and stress that microarchitectural attacks are declared *in-scope* for the Firecracker threat model. The “design” page in AWS’s Firecracker Git repository elaborates on the isolation model and provides a useful diagram which we reproduce in fig. 2.2. The outermost layer of protection is the jailer, which uses container isolation techniques to limit interactions between the Firecracker process and the host kernel. Within the Firecracker process, there are threads for the VMM, management components, and guest workload inside the VM. Since the VM is isolated via hardware virtualization techniques, the user’s code, the guest kernel, and the VMM operate in separate address spaces and cannot architecturally or transiently access each other’s memory. However, many microarchitectural attacks including MDS and Spectre variants ignore address space boundaries and leak data or manipulate execution through internal CPU buffers.

2.14.1 Firecracker Security Recommendations

Prior to our disclosure to AWS, the Firecracker documentation recommended the following precautions for protecting against microarchitectural side-channels [13]: (1) disable SMT, (2) enable kernel page-table isolation, (3) disable kernel same-page merging, (4) use a kernel compiled with Spectre-BTB mitigation (e.g., IBRS and IBPB on x86), (5) verify Spectre-PHT mitigation, (6) enable L1TF mitigation, (7) enable Spectre-STL mitigation, (8) use memory with Rowhammer mitigation, and (9) disable swap or use secure swap. As a result of this work, the current version of this documentation [14] includes many of the same recommendations but emphasizes the incompleteness of the list and strongly recommends the use of security documentation from the Linux kernel and CPU vendors for the most up-to-date firmware patches and configuration recommendations.

Chapter 3

Related Works

3.1 Works on FPGA Security

Classical power analysis techniques like Kocher et al.'s differential power analysis [114] have been applied in new attacks on inter-chip FPGAs [177, 178, 224]. Such integrated and inter-chip FPGAs are available in various cloud environments and system-on-chips (SoCs) products. In particular, Zhao et al. [224] demonstrated how to build an on-chip power monitor using ring oscillators (ROs) which can be used to attack the host CPU or other FPGA tenants. In multi-tenant FPGA scenarios where partial reconfiguration by two separate security domains is possible, more powerful attacks become possible. For instance, the long wires on the FPGA can spy on adjacent wires using ROs [56, 163, 167]. Ramesh et al. [167] exploited the speed of ROs to infer the carried bit in the adjacent wire and demonstrated a key recovery attack

on AES. ROs can also be used as power wasters to create voltage drop and timing faults [57, 117]. Note that such attacks rely on FPGA multi-tenancy which is not widely used yet. In contrast, in this work, we only assume that the FPGA-CPU memory subsystem is shared among tenants.

3.2 Works on Heterogeneous Microarchitectural Attacks

A number of researchers have constructed timing side-channels from CPUs to GPUs, enabling key recoveries, neural network model extraction, and other exploits [146]. Our initial publication of Jackhammer [207] in 2020 was the first demonstration of FPGA-based Rowhammer and FPGA-CPU timing side-channels. This was followed by Purnal et al.’s 2022 paper “Double Trouble,” which presented a new and powerful approach for finding eviction sets in non-inclusive shared caches that requires both a CPU and an FPGA working together [165].

3.3 Attacks on IOMMUs

In the past, several attacks have been shown that circumvent the IOMMU to gain direct memory access or use the misconfiguration of the IOMMU to exploit device drivers through code injection or control-flow hijacking. However, the root cause always was a misconfigured IOMMU or a software

vulnerability. We are not aware of any attacks that were made possible solely by the IOMMU hardware.

For example, a malicious peripheral can bypass the IOMMU by adding appropriate entries to the page table on startup before the IOMMU is activated by the BIOS [140, 141], or by exploiting PCIe address translation services (ATS), which allows a peripheral to mark any memory request as “translated” and bypass IOMMU translation and isolation [129]. Malicious devices may also exploit vulnerabilities in the kernel or device drivers. IOMMU address translation only works on a page-granular level, so memory that was never intended to be shared might be allocated to a shared page, leaking secret data or enabling code injection attacks that can compromise the whole system [129].

3.4 Works on MicroVM Security

To our knowledge, this work is the first that evaluates a class of attacks in and out of a particular virtual machine platform. Other works have investigated software vulnerabilities in microVMs. Xiao et al. showed that even with the minimal attack surface between a microVM and the host kernel, an attacker from within the VM can trigger host kernel functions and system calls to perform a wide range of attacks, including privilege escalation, performance degradation, and crashing the host [212].

A number of works have focused on efficiently integrating trusted execution environments into MicroVMs or serverless platforms, with implementation

methods including Intel SGX [30], Trusted Platform Modules (TPMs) [158], and pure software enclaves [225]. While these environments can harden and verify high-priority code, they can still be vulnerable to microarchitectural attacks. In some cases, the additional microcode and hardware in SGX and TPMs intended to provide isolation even introduce new exploits that strengthen existing attacks [35, 139].

Chapter 4

JackHammer

**Efficient Rowhammer on Heterogeneous FPGA-CPU
Platforms**

4.1 Introduction

4.1.1 Contributions

We demonstrate novel attacks between the memory interface of Intel Arria 10 GX platforms and their host CPUs. Furthermore, we demonstrate a Rowhammer mounted from the FPGA against the CPU to cause faults in the WolfSSL RSA signature implementation, and to leak a private RSA modulus factor. In summary:

- We thoroughly reverse-engineer and analyze the cache behavior and

investigate the viability of cache attacks on realistic FPGA-CPU hybrid systems.

- Based on our study of the cache subsystem, we build JackHammer, a Rowhammer from the FPGA that bypasses caching to hammer the main memory. We compare JackHammer with the CPU Rowhammer and show that JackHammer is twice as fast as a CPU attack, causing faults that the CPU Rowhammer is unable to replicate. JackHammer remains stealthy to CPU monitors since it bypasses the CPU microarchitecture.
- Using both JackHammer and conventional CPU Rowhammer, we demonstrate a fault attack on recent versions of RSA implementation in the WolfSSL library and recover private keys. We show that the base blinding used in this RSA implementation leaves the algorithm vulnerable to the Bellcore fault injection attack.
- We systematically analyze cache attack techniques on different scenarios: FPGA to CPU, CPU to FPGA, and FPGA to FPGA, and demonstrate a cache covert channel that can transmit up to 1.5 – 1.8 MBit/s from the FPGA to the CPU.

Vulnerability Disclosure We informed the WolfSSL team about the vulnerability to Bellcore-style RSA fault injection attacks on November 25, 2019. WolfSSL acknowledged the vulnerability on the same day, and released WolfSSL 4.3.0 with a fix for the vulnerability on December 20, 2019. The

vulnerability can be tracked via CVE-2019-19962 [134].

4.1.2 Experimental Setup

We experiment with two distinct FPGA-CPU platforms with Intel Arria 10 FPGAs: **1)** integrated into the CPU package and **2)** Programmable Acceleration Card (PAC). The integrated Intel Arria 10 is based on a prototype E5-2600v4 CPU with 12 physical cores. The CPU has a Broadwell architecture in which the last level cache (LLC) is inclusive of the L1/L2 caches. The CPU package has an integrated Arria 10 GX 1150 FPGA running at 400 MHz. All measurements done on this platform are strictly done from userspace only, as access is kindly provided by Intel through their Intel Lab (IL) Academic Compute Environment.¹ The IL environment also gives us access to platforms with PACs with Arria 10 GX 1150 FPGA installed and running at 200 MHz. These systems have Intel Xeon Platinum 8180 CPUs that come with non-inclusive LLCs. We carried out the Rowhammer experiments on our local Dell Optiplex 7010 system with an Intel i7-3770 CPU, and a single DIMM of Samsung M378B5773DH0-CH9 1333 MHz 2 GB DDR3 DRAM equipped with the same Intel PAC running with a primary clock speed of 200 MHz.²

The operating system (OS) running in the IL is a 64-bit Red Hat Enterprise Linux 7 with Kernel version 3.10. The OPAE version was compiled and installed on July 15th, 2019 for both the FPGA PAC and the integrated

¹<https://wiki.intel-research.net/>

²The PAC is intended to support 400 MHz clock speed, but the current version of the Intel Acceleration Stack has a bug that halves the clock speed.

FPGA platform. We used Quartus 17.1.1 and Quartus 16.0.0 to synthesize AFUs for the PACs and integrated FPGAs, respectively. The bitstream version of the non-user-configurable Board Management Controller (BMC) firmware is 1.1.3 on the FPGA PAC and 5.0.3 on the integrated FPGA. The OS on our Optiplex 7010 workstation is Ubuntu 16.04.4 LTS with Linux kernel 4.13.0-36. On this system, we installed the latest stable release of OPAE, 1.3.0, and on its FPGA PAC, we installed the compatible 1.1.3 BMC firmware bitstream.

4.2 Analysis of Intel FPGA-CPU Systems

This section explains the hardware and software interfaces that the Intel Arria 10 GX FPGA platforms use to communicate with their host CPUs and the firmware, drivers, and architectures that underlay them. Figure 4.1 gives an overview of such architecture.

Introduction to Intel Terminology Intel refers to a single logical unit implemented in FPGA logic and having a single interface to the CPU as an Accelerator Functional Unit (AFU). So far, available FPGA platforms only support one AFU per Partial Reconfiguration Unit (PRU, also called the *Green Region*). The AFU is an abstraction similar to a program that captures the logic implemented on an FPGA. The FPGA Interface Manager (FIM) is part of the non-user-configurable portion (*Blue Region*) of the FPGA and

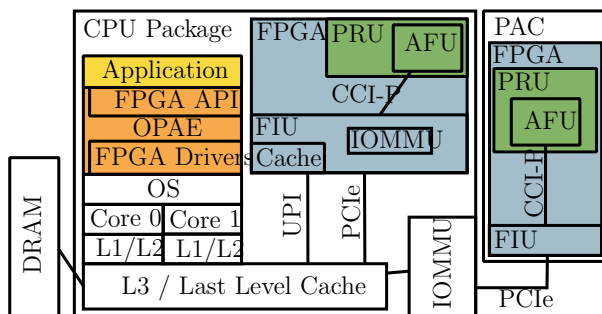


Figure 4.1: Overview of the architecture of Intel FPGAs. The software part of the Intel Acceleration Stack called OPAE is highlighted in orange. Its API is used by applications (yellow) to communicate with the AFU. The Green Region marks the part of the FPGA that is re-configurable from userspace at runtime. The Blue Region describes the static soft core of the FPGA. It exposes the CCI-P interface to the AFU.

contains external interfaces like memory and network controllers as well as the FPGA Interface Unit (FIU), which bridges those external interfaces with internal interfaces to the AFU.

4.2.1 Intel FPGA Platforms

Intel's Arria 10 GX Programmable Acceleration Card (PAC) is a PCIe expansion card for FPGA acceleration [85]. The Arria 10 GX FPGA on the card communicates with its host processor over a single PCIe Gen3x8 bus. Memory reads and writes from the FPGA to the CPU's main memory use physical addresses; in virtual environments, the PCIe controller on the CPU side implements an I/O memory management unit (IOMMU) to translate physical addresses in the virtual machine (what Intel calls I/O Virtual Addresses or IOVA) to physical addresses in the host. Alongside the FPGA, the PAC

carries 8 GB of DDR4, 128 MB of flash memory, and USB for debugging.

An alternative accelerator platform is the **Xeon server processor with an integrated Arria 10 FPGA** in the same package [78]. The FPGA and CPU are closely connected through two PCIe Gen3x8 links and an UltraPath Interconnect (UPI) link. UPI is Intel’s high-speed CPU interconnect (replacing its predecessor QPI) in Skylake and later Intel CPU architectures [142]. The FPGA has a 128 KiB directly mapped cache that is coherent with the CPU caches over the UPI bus. Like the PCIe link on the PAC, both the PCIe links and the UPI link use I/O virtual addressing, appearing as physical addresses to virtualized environments. As the UPI link bypasses the PCIe controller’s IOMMU, the FIU implements its own IOMMU and Device TLB to translate physical addresses for reads and writes using UPI [84].

4.2.2 Intel’s FPGA-CPU Compatibility Layers

Open Programmable Acceleration Engine (OPAE) Intel’s latest generations of FPGA products are designed for use with the OPAE [83] which is part of the Intel Acceleration Stack. The principle behind OPAE is that it is an open-source, hardware-flexible software stack designed for interfacing with FPGAs that use Intel’s Core Cache Interface (CCI-P), a hardware host interface for AFUs that specifies transaction requests, header formats, timing, and memory models [84]. OPAE provides a software interface for software developers to interact with a hosted FPGA, while CCI-P provides a hardware interface for hardware developers to interact with a host CPU.

Excluding a few platform-specific hardware features, any CCI-P compatible AFU should be synthesizable (and the result should be logically identical) for any CCI-P compatible FPGA platform; OPAE is built on top of hardware- and OS-specific drivers and as such is compatible with any system with the appropriate drivers available. As described below, the OPAE/CCI-P system provides two main methods for passing data between the host CPU and the FPGA.

Memory-mapped I/O (MMIO) OPAE can send 32- or 64-bit MMIO requests to the AFU directly or it can map an AFU's MMIO space to OS virtual memory [83]. CCI-P provides an interface for incoming MMIO requests and outgoing MMIO read responses. The AFU may respond to read and write requests in any way that the developer desires, though an MMIO read request will time out after 65,536 cycles of the primary FPGA clock. In software, MMIO offsets are counted as the number of bytes and expected to be multiples of 4 (or 8, for 64-bit reads and writes), but in CCI-P, the last two bits of the address are truncated, because at least 4 bytes are always being read or written. There are 16 available address bits in CCI-P, meaning that the total available MMIO space is 2^{16} 32-bit words, or 256 KiB [84].

Direct memory access (DMA) OPAE can request the OS to allocate a block of memory that can be read by the FPGA. There are a few important details in the way this memory is allocated: most critically, it is allocated in a contiguous physical address space. The FPGA will use physical addresses

to index the shared memory, so physical and virtual offsets within the shared memory must match. On systems using Intel Virtualization Technology for Directed I/O (VT-d), which employs the IOMMU to provide an IOVA to PCIe devices, the memory will be allocated in continuous IOVA space. Either way, this ensures that the FPGA will see an accessible and continuous buffer of the requested size. For buffer sizes up to and including one 4 kB memory page, a normal memory page will be allocated to the calling process by the OS and configured to be accessible by the FPGA with its IOVA or physical address. For buffer sizes greater than 4 kB, OPAE will call the OS to allocate a 2 MB or 1 GB huge page. Keeping the buffer in a single page ensures that it will be contiguously allocated in physical memory.

4.2.3 Cache and Memory Architecture on the Intel FPGAs

Arria 10 PAC The Arria 10 PAC has access to the CPU's memory system as well as its own local DRAM with a separate address space from that of the CPU and its memory. The PAC's local DRAM is always directly accessed, without a separate caching system. When the PAC reads from the CPU's memory, the CPU's memory system will serve the request from its LLC if possible. If the memory that is read or written is not present in the LLC, the request will be served by the CPU's main DRAM. The PAC is unable to place cache lines into the LLC with reads, but writes from the PAC update

the LLC.

Integrated Arria 10 The integrated Arria 10 FPGA has access to the host memory. Additionally, it has its own 128 kB cache that is kept coherent with the CPU’s caches over UPI. Memory requests over PCIe take the same path as requests issued by an FPGA PAC. If the request is routed over UPI, the local coherent FPGA cache is checked first, on a cache miss, forwarding the request to the CPU’s LLC or main memory.

Table 4.1: Overview of the caching hints configurable over CCI-P on an integrated FPGA. *_I hints invalidate a cache line in the local cache. Reading with `RdLine_S` stores the cache line in the shared state. Writing with `WrLine_M` caches the line modified state.

Cache Hint	<code>RdLine_I</code>	<code>RdLine_S</code>	<code>WrLine_I</code>	<code>WrLine_M</code>	<code>WrPush_I</code>
Desc.	No FPGA caching	Leave FPGA cache in S state	No FPGA caching	Leave FPGA cache in M state	Intent to cache in LLC
Available	UPI, PCIe	UPI	UPI, PCIe	UPI	UPI, PCIe

Reverse-engineering Caching Hint Behavior

An AFU on the Arria 10 GX can exercise some control over caching behavior by adding caching hints to memory requests. The available hints are summarized in table 4.1. For memory reads, `RdLine_I` is used to not cache data locally and `RdLine_S` to cache data locally in the shared state. For memory writes, `WrLine_I` is used to prevent local caching on the FPGA, `WrLine_M` leaves

written data in the local cache in the modified state. `WrPush_I` does not cache data locally but hints the cache controller to cache data in the CPU’s LLC. The CCI-P documentation lists all caching hints as available for memory requests over UPI [84]. When sending requests over PCI, only `RdLine_I`, `WrLine_I`, and `WrPush_I` can be used while other hints are ignored. However, based on our experiments, not all cache hints are implemented exactly to specification.

To confirm the behavior of caching hints available for DMA writes, we designed an AFU that writes a constant string to a configurable memory address using a configurable caching hint and bus. We used the AFU to write a cache line and afterward timed a read access to the same cache line on the CPU. These experiments confirm that nearly 100% of the cache lines written to by the AFU are placed in the LLC, as access times stay below 100 CPU clock cycles while main memory accesses take 175 cycles on average. This behavior is independent of the caching hint, the bus, or the platform (PAC, integrated **Arria 10**). The result is surprising as the caching hint meant to cache the data in the cache of the integrated **Arria 10** and the caching hint meant for writing directly to the main memory are either ignored by the Blue Region and the CPU or not implemented yet. Intel later verified that the Blue Region in fact ignores all caching hints that apply to DMA writes. Instead, the CPU is configured to handle all DMA writes as if the `WrPush_I` caching hint is set. The observed LLC caching behavior is likely caused by Intel’s Data Direct I/O (DDIO), which is enabled by default in

recent Intel CPUs. DDIO is meant to give peripherals direct access to the LLC and thus causes the CPU to cache all memory lines written by the AFU. DDIO restricts cache access to a subset of ways per cache set, which reduces the attack surface for Prime+Probe attacks. Nonetheless, attacks against other DDIO-enabled peripherals are possible [118, 190].

4.3 JackHammer Attack

Contribution In this section, we present and evaluate a simple AFU for the Arria 10 GX FPGA that is capable of performing Rowhammer against its host CPU’s DRAM as much as two times faster and four times more effectively than its host CPU can. In a Rowhammer, a significant factor in the speed and efficacy of an attack is the rate at which memory can be repeatedly accessed. On many systems, the CPU is sufficiently fast to cause some bit flips, but the FPGA can repeatedly access its host machine’s memory system substantially faster than the host machine’s CPU can. Both the CPU and FPGA share access to the same memory controller, but the CPU must flush the memory after each access to ensure that the next access reaches DRAM; memory reads from the FPGA do not affect the CPU cache system so no time is wasted flushing memory with the FPGA implementation. We further measure the performance of CPU and FPGA Rowhammer implementations with caching both enabled and disabled, and find that disabling caching brings CPU Rowhammer speed near that of our FPGA Rowhammer implementation.

Crucially, the architectural difference also means that it is much more difficult for a program on the CPU to detect the presence of an FPGA Rowhammer than that of a CPU Rowhammer — the FPGA’s memory accesses leave far fewer traces on the CPU.

4.3.1 JackHammer: Our FPGA Implementation of Rowhammer

JackHammer supports configuration through the MMIO interface. When the JackHammer AFU is loaded, the CPU first sets the target physical addresses that the AFU will repeatedly access. It is recommended to set both addresses for a double-sided attack, but if the second address is set to 0, JackHammer will perform a single-sided attack using just the first address. The CPU must also set the number of times to access the targeted addresses.

When the configuration is set, the CPU signals the AFU to repeat memory accesses and issue them as fast as it can, alternating between addresses in a double-sided attack. Note that unlike a software implementation of Rowhammer, the accessed addresses do not need to be flushed from cache — DMA read requests from the FPGA do not cache the cache line in the CPU cache, though if the requested memory is in the last-level cache, the request will be served to the FPGA by the cache instead of by memory (see section 4.2.3 for more details on caching behavior). In this attack, the attacker needs to ensure that the cache lines used for inducing bit flips are

never accessed by the CPU during the attack. The number of times to access the target addresses can be read again to get the number of remaining accesses; this is the simplest way to check in software whether or not the AFU has finished sending these accesses. When the last read request has been sent by the AFU, the total amount of time taken to send all of the requests is recorded.³

4.3.2 JackHammer on the FPGA PAC vs. CPU Rowhammer

Figure 4.2 shows a box plot of the 0th, 25th, 50th, 75th, and 100th percentile of measured “hammering rates” on the Arria 10 FPGA PAC and its host i7-3770 CPU. Each measurement in these distributions is the average hammering rate over a run of 2 billion memory requests. Our JackHammer implementation is substantially faster than the standard CPU Rowhammer, and its speed is far more consistent than the CPU’s. The FPGA can manage an average throughput of one memory request, or “hammer,” every ten 200 MHz FPGA clock cycles (finishing 2 billion hammers in an average of 103.25 seconds); the CPU averages one hammer every 311 3.4 GHz CPU clock cycles (finishing 2 billion hammers in an average of 183.41 seconds). Here we can see that even if the FPGA were clocked higher, it would still spend most of its time

³The time to send all the requests is not precisely the time to complete all the requests, but it is very close for sufficiently high numbers of requests. The FPGA has a transaction buffer that holds up to 64 transactions after they have been sent by the AFU. The buffer does take some time to clear, but the additional time is negligible for our performance measurements of millions of requests.

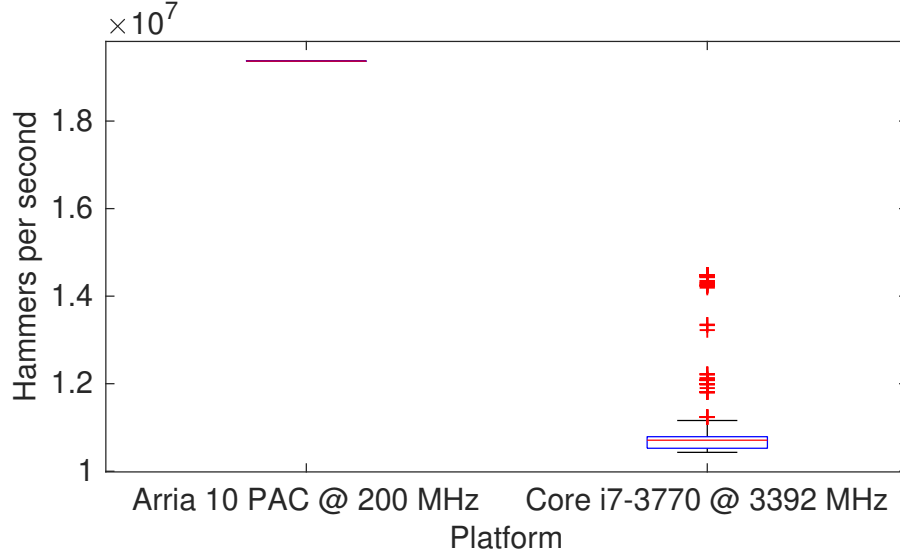


Figure 4.2: Distributions of hammering rates (memory requests per second) on FPGA PAC and i7-3770.

waiting for entries in the PCIe transaction buffer in the non-reconfigurable region to become available.

Figure 4.3 shows measured bit flip rates in the victim row for the same experiment. Runs where zero flips occurred during hardware or software hammering were excluded from the flip rate distributions, as they are assumed to correspond with sets of rows that are in the same logical bank, but not directly adjacent to each other. The increased hammering speed of JackHammer produces a more than proportional increase in flip rate, which is unsurprising due to the highly nature of Rowhammer. As the Rowhammer is underway, electrical charge is drained from capacitors in the victim row. However, the memory controller also periodically refreshes the charge in the capacitors. When there are more memory accesses to adjacent rows within

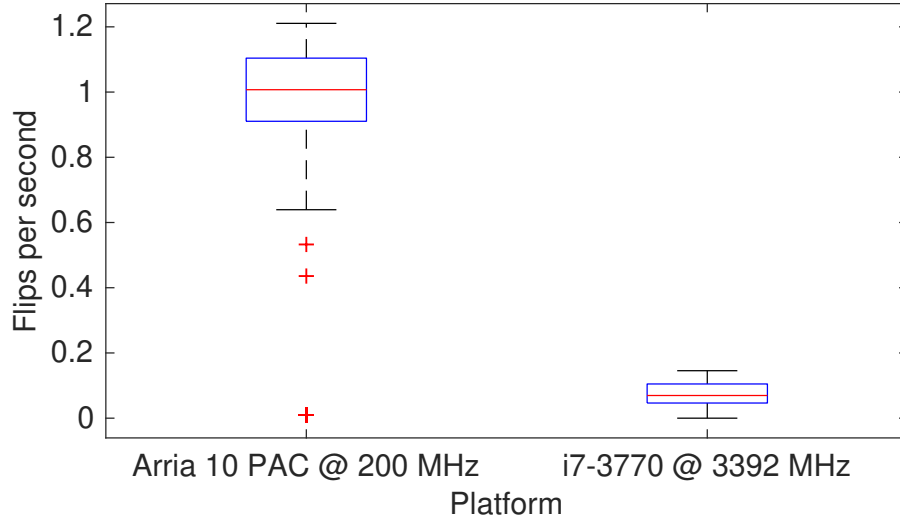


Figure 4.3: Distributions of flip rates on FPGA PAC and i7-3770.

each refresh window, it is more likely that a bit flip occurs before the next refresh. This is why the FPGA’s increased memory throughput is more effective for conducting Rowhammer against the same DRAM chip.

Another way to look at hammering performance is by counting the total number of flips produced by a given number of hammers. Figure 4.4 and fig. 4.5 show minimum, maximum, and every 10th percentile of the number of flips produced by the AFU and CPU respectively for a range of total number of hammers from 200 million to 2 billion. These graphs demonstrate how much more effectively the FPGA PAC can generate bit flips in the DRAM even after the same number of memory accesses. For hammering attempts that resulted in a non-zero number of bit flips, the AFU exhibits a wide distribution of flip count in the range of 200 million to 800 million hammers which then rapidly narrows in the range of 800 million to 1.2 billion and

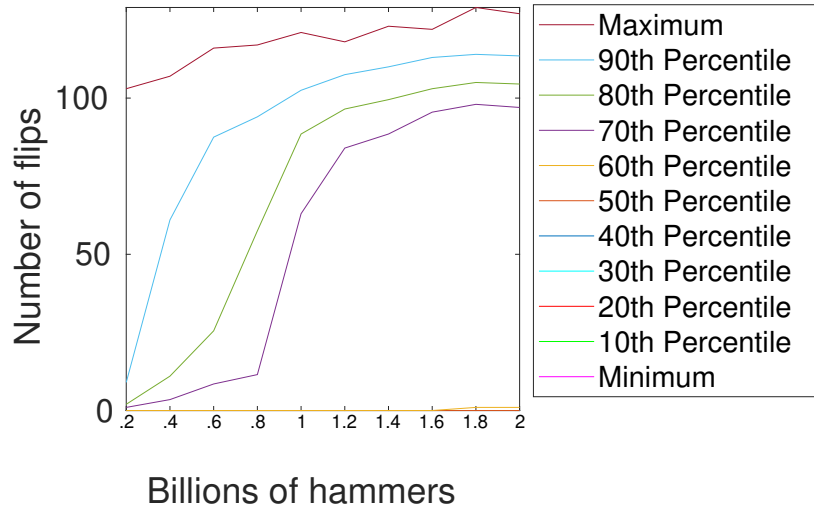


Figure 4.4: Distributions of total flips after 200 million to 2 billion hammers on PAC.

finally levels out by 1.8 billion hammers. This set of distributions seems to indicate that “flippable” rows will ultimately reach about 80-120 total flips after enough hammering, but it can take anywhere from 200 million hammers (about 10 seconds) to 2 billion hammers (about 100 seconds) to reach that limit.

There are also a few rows that only incur a few flips. These samples appear in a consistent pattern demonstrated in fig. 4.6, which plots a portion of the data used to create fig. 4.4 in detail. Each impulse in this plot represents the number of flips after a single run of 2 billion hammers on a particular target row. In fig. 4.6, at indices 23 and 36, two of these outliers are visible, each appearing two indices after several samples in the standard 80-120 flip range. These outliers could indicate rows that are affected vary slightly by hammering on rows that are nearby but not adjacent.

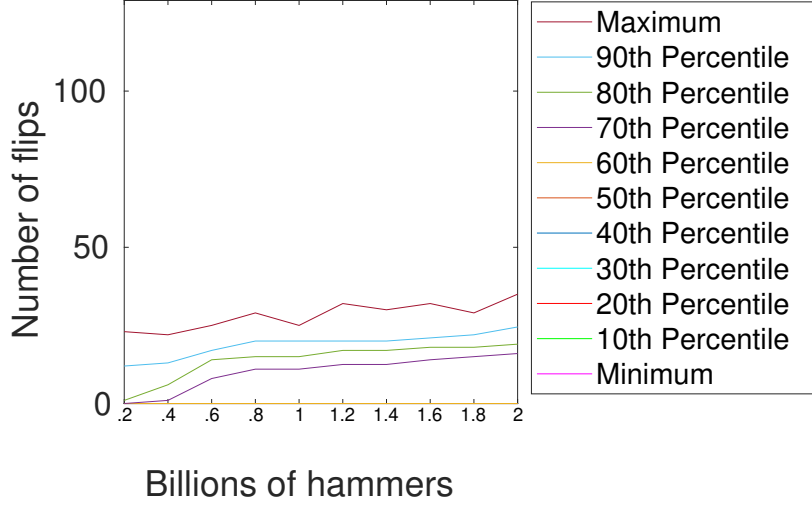


Figure 4.5: Distributions of total flips after 200 million to 2 billion hammers on i7-3770.

4.3.3 JackHammer on the Integrated Arria 10 vs. CPU Rowhammer

The JackHammer AFU we designed for the integrated platform is the same as the AFU for the PAC, except that the integrated platform has access to more physical channels for the memory reads. The PAC only has a single PCIe channel; the integrated platform has one UPI channel and two PCIe channels, as well as an “automatic” setting which lets the interface manager select a physical channel automatically. Therefore we present the hammering rates on this platform with two different settings — alternating PCIe lanes on each access and using the automatic setting.

However, this platform is only available to us on Intel’s servers, so we have only been able to test on one DRAM setup and have been unable to

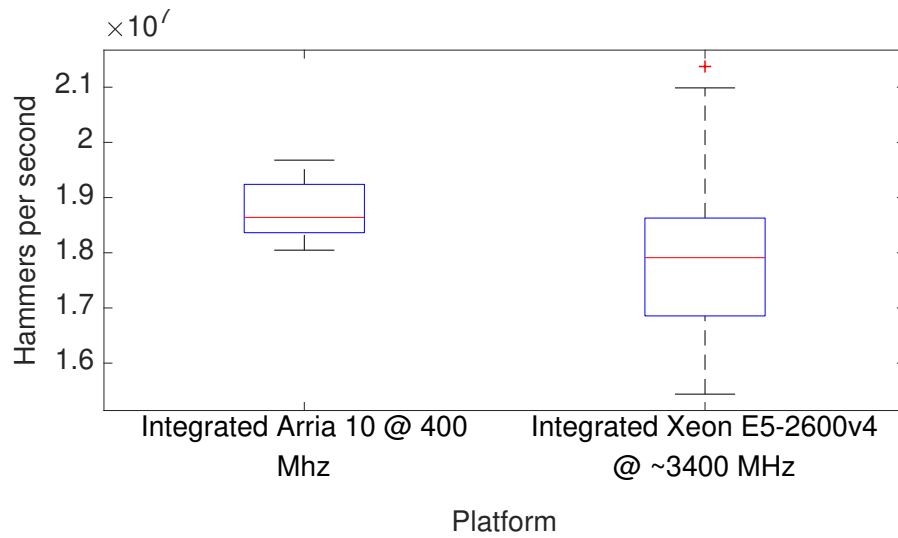


Figure 4.7: Distributions of hammering rates on integrated Arria 10 and Xeon E5-2600 v4.

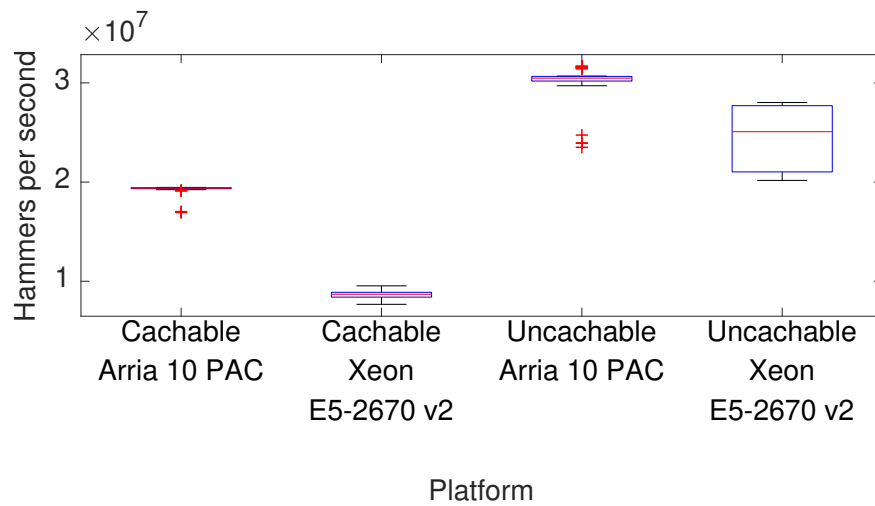


Figure 4.8: Distributions of hammering rates with cachable and uncachable memory.

4.3.4 The Effect of Caching on Rowhammer Performance

We hypothesized that a primary reason for the difference in Rowhammer performance between JackHammer on the FPGAs and a typical Rowhammer implementation on the CPUs is that when one of the FPGAs reads a line of memory from DRAM, it is not cached, so the next read will miss the cache and be directed to the DRAM as well. On the other hand, when the CPUs access a line of memory, it is cached, and the memory line must be flushed from cache before the next read is issued, or the next read will hit the cache instead of DRAM, and the physical row in the DRAM will not be “hammered.”

To evaluate our hypothesis that caching is an important factor in the performance disparity we observed between FPGA- and CPU-based Rowhammer, we used the PTEditor [181] kernel module to set allocated pages as uncachable before testing hammering performance. We edited the setup of the Rowhammer performance tests to allocate many 4 kB pages and set all of those as uncachable instead of one 2 MB huge page, as the kernel module we used to set the pages as uncachable was not correctly configuring the huge pages as uncachable. However, it is still easy to find a large continuous range of physical addresses — when these pages are allocated by OPAE, the physical address is directly available to the software. So the software simply allocates thousands of 4 kB pages, sorts them, and then finds the biggest continuous

range within them and attempts to find colliding row addresses within that range. The JackHammer AFU required no modifications from the initial performance tests; the assembly code to hammer from the CPU was edited to not flush the memory after reading it, since the memory will not be cached in the first place.

We performed this experiment by placing the FPGA PAC on a Dell Poweredge R720 system with a Xeon E5-2670 v2 CPU fixed to a clock speed of 2500 MHz and two 4 GB DIMMs of DDR3 DRAM clocked at 1600 MHz. Figure 4.8 shows the performance of the FPGA PAC and this system's CPU with caching enabled and disabled. Disabling caching produces a significant speedup in hammering for both the PAC and the CPU, but especially for the CPU, which saw a 188% performance increase. With caching enabled, the median hammering rate of the PAC was more than twice that of the CPU, but with caching disabled, the median hammering rate of the PAC was only 22% faster than that of the CPU. Of course, memory accesses on modern systems are extremely complex (even with caching disabled), so there are likely some factors affecting the changes in hammering rate that we cannot describe, but our experimental evidence supports our hypothesis that time spent flushing the cache is a major factor slowing down CPU Rowhammer implementations compared to FPGA implementations.

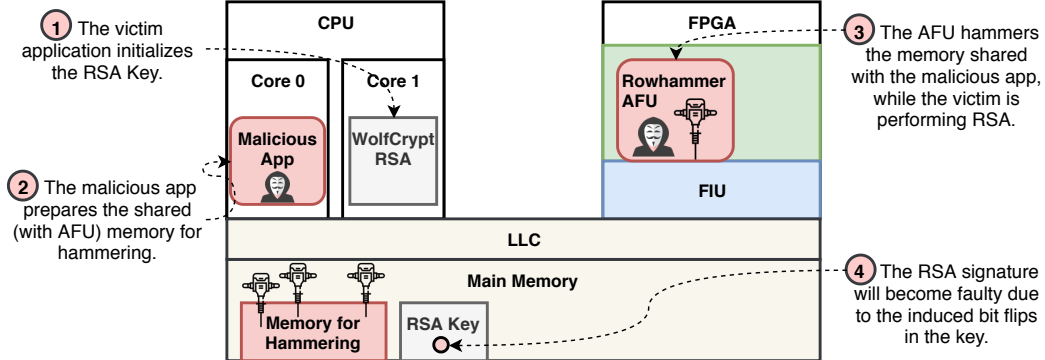


Figure 4.9: WolfSSL RSA Fault Injection Attack.

4.4 Fault Attack on RSA using JackHammer

Rowhammer has been used for fault injections on cryptographic schemes [25, 26] or for privilege escalation [63, 184, 200]. Using JackHammer, we demonstrate a practical fault injection attack from the Arria 10 FPGA to the WolfSSL RSA implementation running on its host CPU. In the RSA fault injection attack proposed by Boneh et al. [27], an intermediate value in the Chinese remainder theorem modular exponentiation algorithm is faulted, causing an invalid signature to be produced. Similarly, we attack the WolfSSL RSA implementation using JackHammer from the FPGA PAC and Rowhammer from the host CPU, and compare the efficiency of the two attacks. The increased hammering speed and flip rate of the Arria 10 FPGA makes the attack more practical in the time frame of about 9 RSA signatures.

Figure 4.9 shows the high-level overview of our attack: the WolfSSL RSA application runs on one core, while a malicious application runs adjacent to it, assisting the JackHammer AFU on the FPGA in setting up the attack.

JackHammer causes a hardware fault in the main memory, and when the WolfSSL application reads the faulty memory, it produces a faulty signature and leaks the private factors used in the RSA scheme.

4.4.1 RSA Fault Injection Attacks

We implement a fault injection attack against the Chinese remainder theorem implementation of the RSA algorithm, commonly known as the Bellcore attack [27]. Algorithm 1 shows the Chinese remainder theorem (CRT) RSA signing scheme where the signature S is computed by raising a message m to the private exponent d th power, modulo N . d_p and d_q , are precomputed as $d \bmod p - 1$ and $d \bmod q - 1$, where p and q are the prime factors of N [20]. When one of the intermediates S_q or S_p is computed incorrectly, an interesting case arises. Consider the difference between a correctly computed signature S of a message m and an incorrectly computed signature S' of the same message, computed with an invalid intermediate S'_p . The difference $S - S'$ leaves a factor of q times the difference $S_p - S'_p$, so the GCD of $S - S'$ and N is the other factor p [20]. This reduces the problem of factoring N to a simple subtraction and GCD operation, so the private factors (p, q) are revealed if the attacker has just one valid signature and one faulty signature, each signed on the same message m . These factors can also be recovered with just one faulty signature if the message m and public key e are known; it is also equal to the GCD of $S'^e - m$ and N .

Fault Injection Attack with RSA Base Blinding A common modification to any RSA scheme is the addition of base blinding, effective against simple and differential power analysis side-channel attacks, but vulnerable to a correlational power analysis attack demonstrated by [210]. Base blinding is used by default in our target WolfSSL RSA-CRT signature scheme. In this blinding process, the message m is blinded by a randomly generated number r by computing $m_b = m \cdot r^e \bmod n$. The resulting signature $S_b = (m \cdot r^e)^d \bmod n = m^d \cdot r \bmod n$ must then be multiplied by the inverse of the random number r to generate a valid signature $S = S_b \cdot r^{-1} \bmod n$.

This blinding scheme does not prevent against the Bellcore fault injection attack. Consider a valid signature blinded with random factor r_1 and an invalid signature blinded with r_2 . When the faulty signature is subtracted by the valid signature, the valid and blinded intermediates S_{pb} are each unblinded and cancel as before, as shown in eq. (4.1).

$$\begin{aligned}
S - S' &= [S_{qb} + q \cdot ((S_{pb} - S_{qb}) \cdot q^{-1} \bmod p)] \cdot r_1^{-1} \bmod N \\
&\quad - [S_{qb} + q \cdot ((S'_{pb} - S_{qb}) \cdot q^{-1} \bmod p)] \cdot r_2^{-1} \bmod N \\
&= q \cdot [(S_{pb} \cdot q^{-1} \bmod p) \cdot r_1^{-1} - (S'_{pb} \cdot q^{-1} \bmod p) \cdot r_2^{-1}] \bmod N
\end{aligned} \tag{4.1}$$

Ultimately, there is still a factor of q in the the difference $S - S'$ which can be extracted with a GCD as before.

4.4.2 Our Attack

Approach and Justification We developed a simplified attack model to test the effectiveness of the Arria 10 Rowhammer in a fault injection scenario. Our model simplifies the setup of the attack so that we can efficiently measure the performance of both CPU Rowhammer and JackHammer. We sign the same message with the same key repeatedly while the Rowhammer exploit runs, and count the number of correct signatures until a faulty signature is generated, which is used to leak the private RSA key.

Attack Setup In summary, our simplified attack model works as follows: The attacker first allocates a large block of memory and checks it for conflicting row addresses. It then quickly tests which of those rows can be faulted with hammering using JackHammer. A list of rows that incur flips is saved so that it can be iterated over. The program then begins the “attack,” iterating through each row that incurred flips during the test, and through the sixty-four 1024-bit offsets that make up the row. During the attack, the JackHammer AFU is instructed to repeatedly access the rows adjacent to the target row. Meanwhile, in the “victim” program, the targeted data (the precomputed intermediate value $d \bmod q - 1$) is copied to the target address, which is computed as an offset of the targeted row. The victim then enters a loop where it reads back the data from the target row and uses it as part of an RSA key to create a signature from a sample message. Additionally, the “attacker” opens a new thread on the CPU which repeatedly flushes the target

row on a given interval. It is necessary for the attacker to flush the target row because the victim is repeatedly reading the targeted data and placing it in cache, but the fault will only ever occur in main memory. For the victim program to read the faulty data from DRAM, there cannot be an unaffected copy of the same data in cache or the CPU will simply read that copy. As we show below, the performance of the attack depends significantly on the time interval between flushes.

One of the typical complications of a Rowhammer fault injection attack is ensuring that the victim’s data is located in a row that can be hammered. In our simplified model, we choose the location of the victim data manually within a row that we have already determined to be one that incurs flips under a Rowhammer attack so that we may easily test the effectiveness of the attack at various rows and various offsets within the rows. In a real attack, the location of the victim program’s memory can be controlled by the attacker with a technique known as page spraying [184], which is simply allocating a large number of pages and then deallocating a select few, filling the memory in an attempt to cause the victim program to allocate the right pages. Improvements in this process can be made; for example, [25] demonstrated how cache attacks can be used to gather information about the physical addresses of data being used by the victim process.

The other simplification in our model is that we force the CPU to read from DRAM using the `clflush` instruction to flush the targeted memory from cache. In an end-to-end attack, the attacker would use an eviction set

to evict the targeted memory since it is not directly accessible in the attack process’s address space. However, the effect is ultimately the same — the targeted data is forcibly removed from the cache by the attacker.

4.4.3 Performance of the Attack

In this section, we show that our JackHammer implementation with optimal settings can cause a faulty signature an average of 17% faster than a typical CPU-based, software-driven Rowhammer implementation with optimal settings. In some scenarios, the performance is as much as 4.8 times that of the software implementation. However, under some conditions, the software implementation can be more likely to cause a fault over a longer period of time. Our results indicate that increasing the DRAM row refresh rate provides significant but not complete defense against both implementations.

The performance of this fault injection attack is highly dependent on the time interval between evictions, and as such we present all of our results in this section as functions of the eviction interval. Each eviction triggers a subsequent reload from memory when the key is read for the next signature, which refreshes the capacitors in the DRAM. Whenever DRAM capacitors are refreshed, any accumulated voltage error in each capacitor (due to Rowhammer or any other physical effect) is either solidified as a new faulty bit value or reset to a safe and correct value. Too short of an interval between evictions will cause the DRAM capacitors to be refreshed too quickly to be flipped with a high probability. On the other hand, however, longer intervals can mean

the attack is waiting to evict the memory for a longer time while a bit flip has already occurred. It is crucial to note, also, that DRAM capacitors are automatically refreshed by the memory controller on a 64 ms interval⁵ [65]. On some systems, this interval is configurable: faster refresh rates reduce the rate of memory errors, including those induced by Rowhammer, but they can impede maximum performance because the memory spends more time doing maintenance refreshes rather than serving read and write request. For more discussion on modifying row refresh rates as a defense against Rowhammer, see section 4.6.

In table 4.2 we present two metrics with which we compare JackHammer and a standard CPU Rowhammer implementation. This table shows the mean number of signatures until a faulty signature is produced and the ultimate probability of success of an attack within 1000 signatures against a random key in a randomly selected chunk of memory within a row known to be vulnerable to Rowhammer. With an eviction interval of 96 ms, the JackHammer attack achieves the lowest average number of signatures before a fault, at only 58, 25% faster than the best performance of the CPU Rowhammer. The CPU attack is impeded significantly by shorter eviction latency, while the JackHammer implementation is not, indicating that on systems where the DRAM row refresh rate has been increased to protect against memory faults and Rowhammers, JackHammer likely offers substantially improved attack

⁵More specifically, DDR3 and DDR4 specifications indicate 64 ms as the maximum allowable time between DRAM row refreshes.

Table 4.2: Performance of our JackHammer exploit compared to a standard software CPU Rowhammer with various eviction intervals. JackHammer is able to achieve better performance in many cases because it bypasses caching architecture, sending more memory requests during the eviction interval and causing bit flips at a higher rate.

Eviction Interval	Mean signatures to fault			Successful fault rate		
	CPU	JackHammer	% Inc. Speed	CPU	JackHammer	% Inc. Rate
16	280	186	51%	0.4%	0.2%	-46%
32	627	219	185%	0.2%	0.8%	264%
48	273	124	120%	14%	19%	39%
64	81	76	7%	17%	26%	56%
96	74	58	27%	46%	49%	8%
128	73	70	4%	52%	50%	-1.2%
256	106	115	-7%	57%	55%	-3%
Best performance	73	58	25%	57%	55%	-3%

performance. Figure 4.10 highlights the mean number of signatures until a faulty signature for the 16 ms to 96 ms range of eviction latency.

4.5 Cache Attacks on Intel FPGA-CPU Platforms

In section 4.2.3, we reverse-engineered the behavior of the memory subsystem on current Arria 10 based FPGA-CPU platforms. In this section, we systematically analyze cache attacks exploitable by an AFU- or CPU-based attacker attacking the CPU or FPGA, respectively, and demonstrate a cache covert channel from FPGA to CPU. At last, we discuss the viability of intra-FPGA cache attacks. table 4.3 summarizes our findings.

To measure memory access latency on the FPGA, we designed a timer clocked at 200 MHz/400 MHz. The advantage of this hardware timer is that

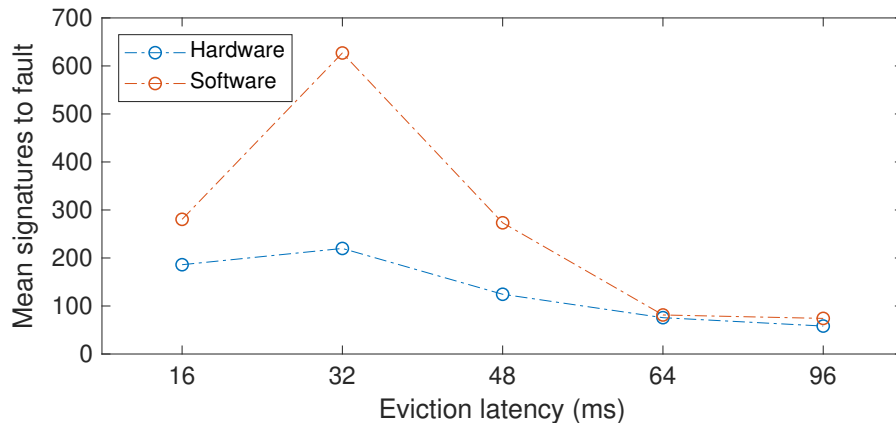


Figure 4.10: Mean number of signatures to fault at various eviction intervals.

it runs uninterruptible in parallel to all other CPU or FPGA operations. Therefore, the timer precisely counts FPGA clock cycles, while timers on the CPU, such as `rdtsc`, may yield noisier measurements due to interruptions by the OS and the CPU’s out-of-order pipeline.

Table 4.3: Summary of our cache attacks analysis: **OPAE** accelerates eviction set construction by making huge pages and physical addresses available to userspace.

Attacker	Target	Channel	Attack
FPGA PAC AFU	CPU LLC	PCIe	E+T, E+R, P+P
Integrated FPGA AFU	CPU LLC	UPI	E+T, E+R, P+P
Integrated FPGA AFU	CPU LLC	PCIe	E+T, E+R, P+P
CPU	FPGA Cache	UPI	F+R, F+F
Integrated FPGA AFU	FPGA Cache	CCI-P	E+T, E+R, P+P

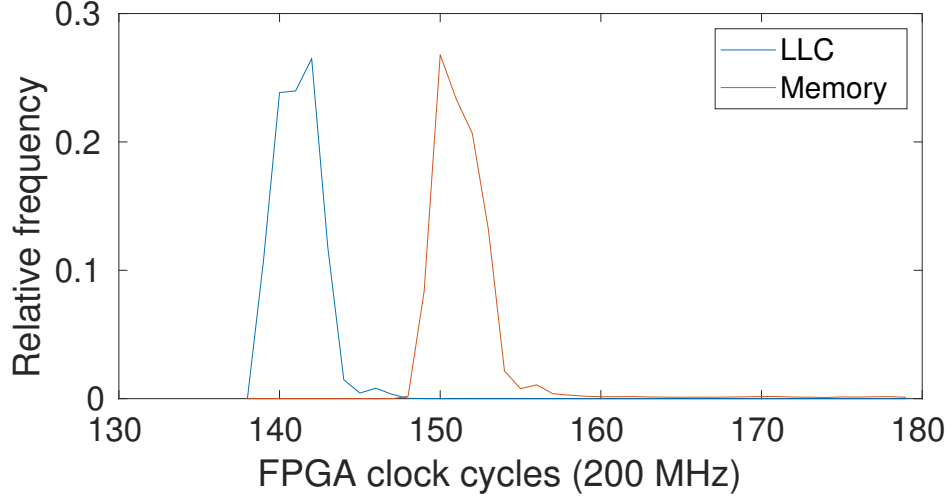


Figure 4.11: Latency for PCIe read requests on an FPGA PAC served by the CPU’s LLC or main memory.

4.5.1 Cache Attacks from FPGA PAC to CPU

The Intel PAC has access to one PCIe lane that connects it to the main memory of the system through the CPU’s LLC. The CCI-P documentation [84] mentions a timing difference for memory requests served by the CPU’s LLC and those served by the main memory. Using our timer we verified the suggested differences as shown in fig. 4.11. Accesses to the LLC take between 139 and 145 cycles; accesses to main memory take 148 to 158 cycles. These distinct distributions of access latency form the basis of cache attacks, as they enable an attacker to tell which part of the memory subsystem served a particular memory request. Our results indicate that FPGA-based attackers can precisely distinguish memory responses served by the LLC from those served by main memory.

In addition to probing, some way of influencing the state of the cache is

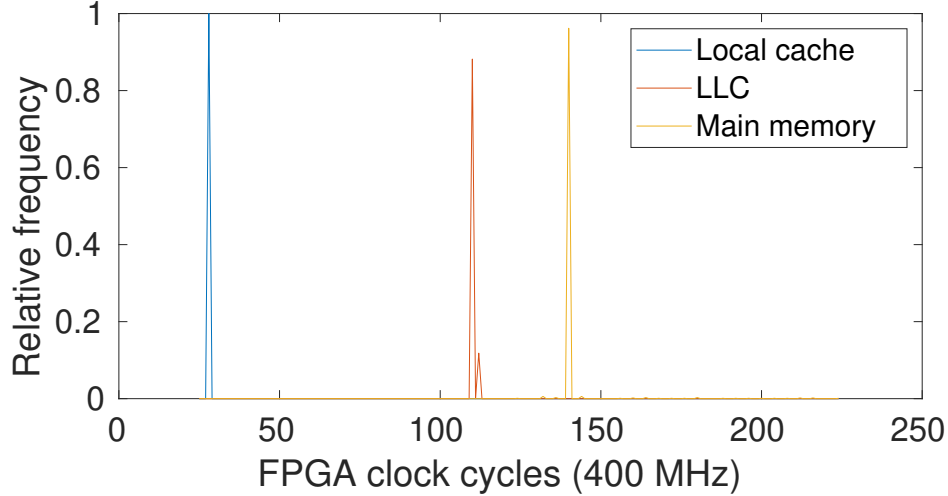


Figure 4.12: Latency for UPI read requests on an integrated **Arria 10** served by the FPGA’s local cache, CPU’s LLC, or main memory.

needed to perform cache attacks. We investigated all possibilities of cache interaction offered by the CCI-P interface on an FPGA PAC and found that cache lines read by the AFU from the main memory will not get cached. While this behavior is not usable for cache attacks, it boosts Rowhammer performance as we saw in section 4.3. On the other hand, cache lines written by an AFU on the PAC end up in the LLC with nearly 100% probability. The reason for this behavior was already discussed together with the analysis of the caching hints. This can be used to evict other cache lines from the cache and perform eviction based attacks like Evict+Time, Evict+Reload, and Prime+Probe. For E+T, DMA writes can be used to evict a cache line and our hardware timer measures the victim’s execution time. Even though an AFU cannot load data into the LLC, E+R can be performed as the purpose of reloading a cache line is to learn the latency and not literally reloading

the cache line. So the primitives for E+R on the FPGA are DMA writes and timing DMA reads with a hardware timer. P+P can be performed using DMA writes and timing reads. In the case where DDIO limits the number of accessible ways per cache set, other DDIO-enabled peripherals are attackable. Flush-based attacks like Flush+Reload or Flush+Flush cannot be performed by an AFU as CCI-P does not offer a flush instruction.

4.5.2 Cache Attacks from Integrated Arria 10 FPGA to CPU

The integrated Arria 10 has access to two PCIe lanes (each functioning much like the PCIe lane on the FPGA PAC) and one UPI lane connecting it to the CPU's memory subsystem. It also has its own additional cache on the FPGA accessible over UPI (cf. section 4.2.3).

By timing memory requests from the AFU using our hardware timer, we show that distinct delays for the different levels of the memory subsystem exist. Both PCIe lanes have delays similar to those measured on a PAC (cf. fig. 4.11). Our memory access latency measurements for the UPI lane, depicted in fig. 4.12, show an additional peak for requests being answered by the FPGA's local cache. The two peaks for LLC and main memory accesses are likely narrower and further apart than in the PCIe case because UPI, Intel's proprietary high-speed processor interconnect, is an on-chip and inter-CPU bus only connecting CPUs and FPGAs. On all interfaces, read

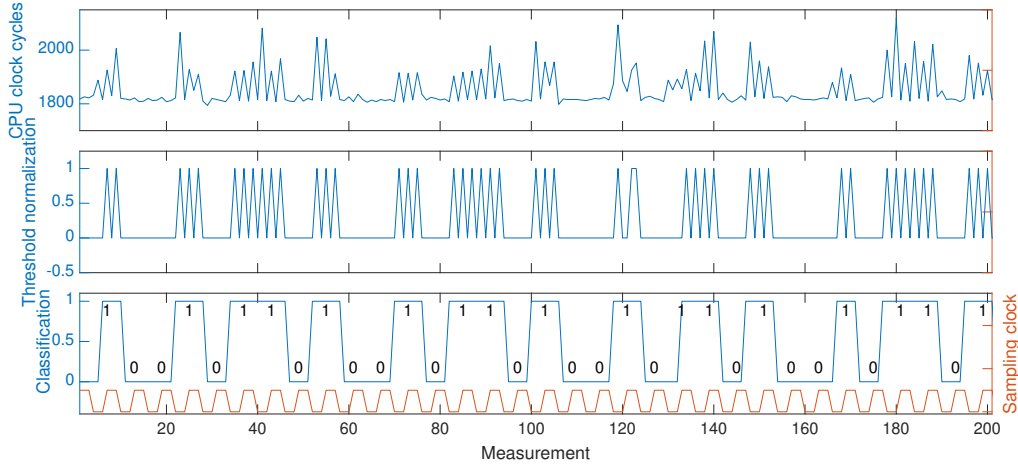


Figure 4.13: Covert channel measurements and decoding. The AFU sends each bit three times, which results in three peaks at the receiver if a ‘1’ is transmitted (middle plot).

requests, again, are not usable for evicting cache lines from the LLC. DMA writes, however, can be used to alter the LLC on the CPU. Because the UPI and PCIe lanes behave much like the PCIe lane on a PAC, we state the same attack scenarios (E+T, E+R, P+P) to be viable on the integrated Arria 10.

Constructing a Covert Channel from AFU to CPU

The fact that an AFU can place data in at least one way per LLC slice allows us to construct a covert channel from the AFU to a co-operating process on the CPU using side effects of the LLC. To do so, we designed an AFU that writes a fixed string to a pre-configured cache line whenever a ‘1’ is transmitted and stays quiet whenever a ‘0’ is sent. Using this technique, the AFU sends messages which can be read by the CPU. For the rest of this

section, we will refer to the address the AFU writes to as the *target address*.

The receiver process⁶ first constructs an eviction set for the set/slice-pair the target address is in. To find an eviction set, we run a slightly modified version of Algorithm 1 using Test 1 in [204]. Using the OPAE API to allocate hugepages and get physical addresses (cf. section 4.2.2) allows us to construct the eviction set from a rather small set of candidate addresses all belonging to the same set.

We construct the covert channel on the integrated platform as the LLC of the CPU is inclusive. Additionally, the receiver has access to the target address via shared memory to have the receiver test its eviction set against the target address directly. This way, we do not need to explicitly identify the target address’s LLC slice. In a real-world scenario, either the slice selection function has to be known [79, 81, 94] or eviction sets for all slices have to be constructed by seeking conflicting addresses [126, 154]. The time penalty introduced by monitoring all cache sets can be prevented by multi-threading.

Next, the receiver primes the LLC with the eviction set found and probes the set in an endless loop. Whenever the execution time of a probe is above a certain threshold, the receiver assumes that the eviction of one of its eviction set addresses was the result of the AFU writing to the target address and therefore interprets this as receiving a ‘1’. If the probe execution time stays below the threshold, a ‘0’ is detected as no eviction of the eviction set addresses

⁶This process is not the software process directly communicating with the AFU over OPAE/CCI-P.

occurred. An example measurement of the receiver and its decoding steps are depicted in fig. 4.13.

To ease decoding and visualization of results, the AFU sends every bit thrice and the CPU uses six probes to detect all three repetitions. This high level of redundancy comes at the expense of speed, as we achieve a bandwidth of about 94.98 kBit/s, which is low when compared to other work [126,132,211]. The throughput can be increased by reducing the three redundant writes per bit from the AFU as well as by increasing the transmission frequency further to reduce the redundant CPU probes per AFU write. Also, multiple cache sets can be used in parallel to encode several bits at once. The synchronization problem can be solved by using one cache set as the clock, where the AFU writes an alternating bit pattern [190]. An average probe on the CPU takes 1855 clock cycles. With the CPU operating in the range of 2.8 – 3.4 GHz, this results in a theoretic throughput of 1.5 – 1.8 MBit/s. On the other side, the AFU can on average send one write request every 10 clock cycles without filling the CCI-P PCIe buffer and thereby losing the write pattern. In theory, this makes the AFU capable of sending 40 MBit/s over the covert channel when clocked at 400 MHz.⁷

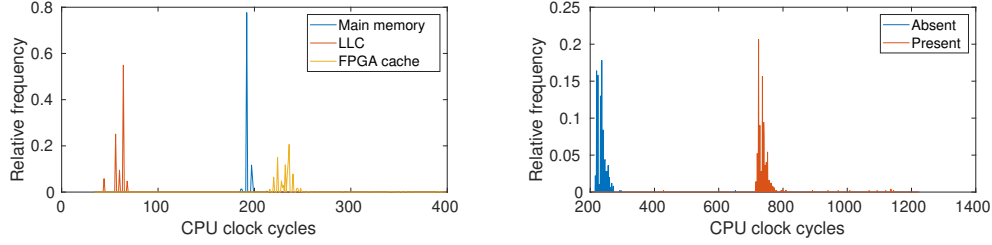
Even though caching hints for memory writes are being ignored by the Blue Region, an AFU can place data in the LLC because the CPU is configured to handle write requests as if `WrPush_I` is set, allowing for producing evictions

⁷This is a worst-case scenario where every transmitted bit is a ‘1’-bit. For a random message, this estimation goes up again as ‘0’-bits do not fill the buffer, allowing for faster transmission.

in the LLC. We corroborated our findings by establishing a covert channel between the AFU and the CPU with a bandwidth of 94.98 kBit/s. By exposing physical addresses to the user and by enabling hugepages, OPAE further eases eviction set finding from userspace.

4.5.3 Cache Attacks from CPU to Integrated Arria 10 FPGA

We also investigated the CPU’s capabilities to run cache attacks against the coherent cache on the integrated Arria 10 FPGA. First, we measured the memory access latency depending on the location of the address accessed using the `rdtsc` instruction. The results in fig. 4.14a show that the CPU can clearly distinguish where an accessed address is located. Therefore, the CPU is capable of probing a memory address that may or may not be present in the local FPGA cache. It is interesting to note that requests to main memory return faster than those going to the FPGA cache. This can be explained by the much slower clock speed of the FPGA running at 400 MHz while the CPU operates at 1.2–3.4 GHz. Another explanation is that our test platform is one of the prototypes and the coherency protocol implementation of the Blue Region is still buggy. As nearly all known cache attack techniques rely on some form of probing phase, the capability to distinguish location of data is a good step in the direction of having a fully working cache attack from the CPU against the FPGA cache.



(a) Memory access latency from the CPU with data being present in FPGA local cache, CPU LLC, or main memory. (b) The `flush` execution time on the CPU with the flushed address being absent or present in the FPGA cache.

Figure 4.14: Memory access and `flush` execution latency measured from a Broadwell CPU with integrated Arria 10.

Besides the capability of probing the FPGA cache, we also need a way of flushing, priming, or evicting cache lines to put the FPGA cache into a known state. While the AFU can control which data is cached locally by using caching hints, there is no such option documented for the CPU. Therefore, priming the FPGA cache to evict cache lines is not possible. This disables all eviction-based cache attacks. However, as the CPU has a `clflush` instruction, we can use it to flush cache lines from the FPGA cache, because it is coherent with the LLC. Hence, we can flush and probe cache lines located in the FPGA cache. This enables us to run a Flush+Reload attack against the victim AFU where the addresses used by the AFU get flushed before the execution of the AFU. After the execution, the attacker then probes all previously flushed addresses to learn which addresses were used during the AFU execution. Another possible cache attack is the more efficient Flush+Flush attack. Additionally, we expect the attack to be more precise as flushing a cache line that is present in the FPGA cache takes about 500

CPU clock cycles longer than flushing a cache line that is not (cf. fig. 4.14b), while the latency difference between memory and FPGA cache accesses adds up to only about 50-70 CPU clock cycles.

In general, the applicability of F+R and F+F is limited to shared memory scenarios. For example, two users on the same CPU might share an instantiation of a library that uses an AFU for acceleration of a process that should remain private, like training a machine learning model with confidential data or performing cryptographic operations.

4.5.4 Intra-FPGA Cache Side-Channels

As soon as FPGAs support simultaneous multi-tenancy, that is, the capability to place two AFUs from different users on the same FPGA at the same time, the possibility of intra-FPGA cache attacks arises. As the cache on the integrated Arria 10 is directly mapped and only 128 kB in size, finding eviction sets becomes trivial when giving the attacker AFU access to huge pages. As this is the default behavior of the OPAE driver when allocating more than one memory page at once, we assume that it is straightforward to run eviction based attacks like Evict+Time or Prime+Probe against a neighboring AFU to e.g. extract information about a machine learning model. Flush-based attacks would still be impossible due to the lack of a flush instruction in CCI-P.

4.6 Countermeasures

Hardware Monitors Microarchitectural attacks against CPUs leave traces in hardware performance counters (HPCs) like cache hit and miss counters. Previous works have paired these HPCs with machine learning techniques to build real-time detectors for these attacks [31, 40, 68, 221]. In some cases, CPU HPCs may be able to trace incoming attacks from FPGAs. While HPCs do not exist in the same form on the Arria 10 GX platforms, they could be implemented by the FIM. A system combining FPGA and CPU HPCs could provide thorough monitoring of the FPGA-CPU interface.

Increasing DRAM Row Refresh Rate An approach to reduce the impact of Rowhammer is increasing the DRAM refresh rate. DDR3 and DDR4 specifications require that each row is refreshed at least every 64 ms, but many systems can be configured to refresh each row every 32 or 16 ms for better memory stability. When we measured the performance of our fault injection attack in section 4.4, we measured the performance with varying intervals between evictions of the targeted data, simulating equivalent intervals in row refresh rate, since each eviction causes a subsequent row refresh when the memory is read by the victim program. Table 4.2 shows that under 1% of attempted Rowhammers from both CPU and FPGA were successful with an eviction interval of 32 ms, compared to 14% of CPU attacks and 26% of FPGA attacks with an interval of 64 ms, suggesting that increasing the row refresh rate would significantly impede even the more powerful FPGA

Rowhammer.

Cache Partitioning and Pinning Several cache partitioning mechanisms have been proposed to protect CPUs against cache attacks. While some are implementable in software [108, 110, 219, 226] others require hardware support [60, 61, 125]. When trying to protect FPGA caches against cache attacks, hardware-based approaches should be taken into special consideration. For example, the FIM could partition the FPGA’s cache into several security domains, such that each AFU can only use a subset of the cache lines in the local cache. Another approach would introduce an additional flag to the CCI-P interface telling the local caching agent which cache lines to pin to the cache.

Disabling Hugepages and Virtualizing AFU Address Space Intel is aware of the fact that making physical addresses available to userspace through OPAE has negative security consequences [83]. Additionally to exposing physical addresses, OPAE makes heavy use of hugepages to ensure physical address continuity of buffers shared with the AFU. However, it is well known that disabling hugepages increases the barrier of finding eviction sets [93, 126] which in turn makes cache attacks and Rowhammer more difficult. We suggest disabling OPAE’s usage of hugepages. To do so, the AFU address space has to be virtualized independent of the presence of virtual environments.

Protection Against Bellcore Attack Defenses against fault injection attacks proposed in the original Bellcore whitepaper [27] include verifying the signature before releasing it, and random padding of the message before signing, which ensures that no unique message is ever signed twice and that the exact plaintext cannot be easily determined. OpenSSL protects against the Bellcore attack by verifying the signature with its plaintext and public key and recomputing the exponentiation by a slower but safer single exponentiation instead of by the CRT if verification does not match [39]. After we reported the vulnerability to WolfSSL, they issued a patch in version 4.3.0 including a signature verification to protect against Bellcore-style attacks.

Chapter 5

IOTLB-SC

An Accelerator-Independent Leakage Source in Modern Cloud Systems

5.1 Introduction

Our Contribution This work exposes a vulnerability in an overlooked attack surface present in multi-tenant, peripheral-heavy cloud systems: the microarchitecture of the I/O Memory Management Unit (IOMMU). Knowing that the IOMMUs in modern CPUs have translation look-aside buffers (IOTLBs) to speed up repeated translations [15, 92, 148], we present a hardware design for an FPGA acceleration card that uses memory access timing to reliably identify whether or not a translation is present in an IOTLB. With that design, we propose and evaluate an algorithm for IOTLB eviction set

finding. With those eviction sets, we demonstrate the first two IOTLB-based covert channels. We use the FPGA to collect side-channel IOTLB traces from two other peripheral devices and analyze the viability and threat models of a full side-channel attack.

We show that the IOTLB is the source of side-channel vulnerability that CSPs are currently not aware of and thus do not protect against. We show that the IOTLB is an excellent source for constructing covert channels between co-located peripherals and can also be abused to extract information from neighboring peripherals such as GPU-accelerated databases. We provide comprehensive threat analysis of this vulnerability, in both the present and the near future, and present viable defenses and countermeasures. In summary, our main contributions are:

- We demonstrate a *previously ignored IOTLB timing side-channel* against PCIe peripherals *before* technologies such as CXL and PCIe 5.0 gain widespread adoption, and fine-grained attacks become viable on a large installation base.
- We develop a new algorithm that finds eviction sets without any prior assumptions of organization and demonstrate its advantages in finding IOTLB eviction sets over a similar eviction set finding algorithm.
- We use a custom FPGA hardware function to exploit the IOTLB timing side-channel and study traces collected from an SQL database acceleration library for a GPU.

- We leak IOTLB timing side-channel traces from a GPU-accelerated SQL database library and analyze the vulnerability of the library to a practical attack.
- We demonstrate the *first* two IOTLB covert channels, including a peripheral-to-peripheral channel with a generic application as the sender and our custom FPGA function as the receiver.
- We propose countermeasures for applications, cloud systems, and IOMMU implementations to counter the side-channel we identified.

5.2 Identifying IOTLB Side-Channels

In this section, we demonstrate two fundamental techniques for implementing IOTLB side-channel attacks on these or similar systems. We measure the latency difference between DMA accesses to addresses with cached and uncached translations in the IOMMU. We also demonstrate a new algorithm for reliably finding IOTLB eviction sets with no prior assumptions about size or organization. We have access to three different system setups that we will investigate throughout this work. Table 5.1 summarizes the key features of each. A detailed description of the setups is given next.

Table 5.1: Overview of the system setups used in this work.

Name	<i>a10l</i>	<i>a10v</i>	<i>s10v</i>
CPU	2 Xeon Silver 4114	2 Xeon Platinum 8180	2 Xeon Platinum 8280
#PCIe RP	4 per CPU	4 per CPU	4 per CPU
#IOMMUs	4 per CPU	4 per CPU	4 per CPU
FPGA PAC	Arria 10	Arria 10	Stratix 10
OPAE ver.	1.1.2-1	2020-01-01	2020-01-01
Bitstream ver.	1.2.3	1.1.3	2.0.3
Root/phys. access	yes	no	no

5.2.1 System Setup

For our experiments, we rely on three systems that are representative of modern cloud services featuring FPGA resources. The systems feature recent server-grade CPUs as well as FPGA extension cards based on Intel FPGAs. The FPGAs are managed by the Intel Acceleration Stack (IAS) which is designed to ease management of cloud deployments. The first system, *a10l*, is a system we have physical and administrative access to. The other two systems *a10v* and *s10v* are cloud-like systems that are accessible through the Intel Labs (IL) Academic Compute Environment (ACE)¹. We operate the two IL ACE systems with user privileges only. This is why we evaluate our eviction set finding algorithm on all three systems but rely solely on *a10l* for the side- and covert channel experiments. More detailed information about the different systems is given in table 5.1 and in the following paragraphs.

a10l: As our local setup, we use a Dell PowerEdge R740 server with two

¹<https://wiki.intel-research.net/>

Intel Xeon Silver 4114 CPUs. Each CPU reports 4 PCIe root bridges with one IOMMU per root port. The system contains a Realtek PCIe ethernet network interface card (NIC). It is assigned to a dedicated IOMMU group. The NIC is passed-through to a virtual machine (VM) on the server. An ethernet cable connects the NIC with one of the on-board NICs. An NVIDIA Tesla T4 GPU is assigned to another dedicate IOMMU group that is managed by a different IOMMU than the NIC. Therefore, the NIC and the GPU do not share an IOTLB. An Intel Programmable Acceleration Card (PAC) with Intel Arria 10 GX FPGA shares the IOTLB with the NIC or the T4, depending on the experiment, by connecting it to PCIe slots that are managed by the IOMMU also managing the NIC or the GPU respectively. All other PCIe devices like the on-board NICs, memory controllers, etc. are connected to different IOMMUs and therefore cannot interfere with our measurements. The system has IAS 1.2 installed which contains OPAE version 1.1.2-1. Running `fpgainfo` reports bitstream id `0x123000200000185` and bitstream version `1.2.3`. We execute the GPU-accelerated database OmniSciDB² in version 5.10., which is the latest version at the time of writing. Additionally, CUDA version 11.4 and GPU driver version 470.57.02 are installed. The database consists of one table filled with the Meta Kaggle data set³. We have root access to this machine.

a10v: The IL ACE contains servers with two Intel Xeon Platinum 8180

²<https://docs.omnisci.com/overview/overview#omniscidb>

³<https://www.kaggle.com/kaggle/meta-kaggle>

CPUs. Each CPU reports 4 PCIe root bridges with IOMMU per root port. Two PCIe PACs with Arria 10 GX FPGAs are managed by two separate IOMMUs. All other PCIe devices are managed by other IOMMUs. The servers use IAS 1.1 and OPAE was installed on 01/01/2020 from the Git repository. Running `fpgainfo` reports bitstream id `0x113000200000177` and bitstream version `1.1.3`. We operate these machines with user privileges only.

s10v: The IL ACE features servers with two Intel Xeon Platinum 8280 CPUs. Each CPU reports 4 PCIe root bridges with one IOMMU per root port. An Intel FPGA PAC D5005 is connected via PCIe. All other PCIe devices are managed by other IOMMUs than the one managing the PAC. The servers use IAS 2.0 and OPAE was installed on 01/01/2020 from the Git repository. Running `fpgainfo` reports bitstream version `2.0.3` and bitstream id `0x203000200000339`. We operate these machines with user privileges only.

5.2.2 IOTLBs Cause Timing Behavior

During their PCIe performance benchmarking, Neugebauer et al. [148] found that an IOTLB miss results in a latency increase of 330 ns. Since the FPGAs in our systems are clocked at 200 MHz, the expected difference between fast and slow accesses is 66 clock cycles. Peglow’s [161] work matches our expectation. With disabled IOMMU, the memory read latency for any address in main memory is distributed around 160 and 185 cycles. When the system is configured to use the IOMMU, this distribution shifts to 225 and 270 cycles

for addresses that are accessed for the first time. Access times for subsequent accesses are distributed similarly to access times measured without IOMMU. Thus the measurable latency difference between accesses to addresses where the translation is present in or absent from the IOTLB lies between 65 and 85 clock cycles. We reproduced all values for the *a10l* system. On the IL ACE systems *a10v* and *s10v*, the latency difference between first accesses and subsequent accesses lies in the expected range. However, we cannot disable the IOMMU on the IL ACE systems to check whether the latency difference disappears.

5.2.3 Tools for Testing IOMMU Behavior

The IOMMU translates addresses for peripherals. Therefore, the CPU alone can only interact with the IOMMU in limited ways; we have to rely on a peripheral device to perform the experiments. For this purpose we used the PCIe PACs with DMA capabilities. We implement a hardware function for the FPGA that is programmable from software to capture the required measurements.

IOTLB Control from the CPU

To assist with these experiments, we also develop a kernel module that enables a program on the CPU to flush all entries from the IOTLB of a given IOMMU. When loaded, the kernel module uses a variety of functions and structures from the Linux kernel source, including those found in `<linux/pci.h>`,

`<linux/iommu.h>`, and `<linux/dmar.h>` to find a PCIe device structure based on its vendor and device IDs, and from there find the device structure corresponding to the IOMMU that manages that PCIe device. That IOMMU device structure already contains a pointer to a function for flushing the IOMMU, so that function merely needs to be called. The kernel module uses a character file and `ioctl` as an interface by which user programs can call for the kernel module to flush the IOMMU. However, it takes root access to load a kernel module, since the module must read and write kernel memory. Therefore, we only tested algorithm 2 with the optional flush on our local system *a10l*.

Hardware Design

Our `iotlb.pnp` hardware module is designed against the Intel Acceleration Stack as would be the case in a cloud environment. The module is capable of performing memory accesses and timing the access latency. `iotlb.pnp` can be programmed with up to 7 instructions. Currently, the design supports 5 instructions: `evset_prime`, `evset_probe`, `target_prime`, `target_probe`, and `wait`. Configuration and programming of the hardware module is performed via MMIO through OPAAE. The prime instructions make the hardware module access a configured address (target) or set of addresses (eviction set). Probe instructions behave in the same way as the prime instructions but additionally count clock cycles. When probing an eviction set, the module can be configured to either measure the overall execution time of the instruction or time each

memory access individually. The eviction sets used during priming and probing can be configured independent from each other, as is the case for the target instructions. The wait instruction simply makes the hardware module do nothing for a configured number of clock cycles.

Software

The software counterpart to the hardware module uses the OPAE C library to interact with the hardware design on the FPGA. This library allows us to control and observe the operation of the hardware module with memory-mapped I/O (MMIO) as well as — crucially for the work that this module must do — allocate shared pages of the system’s main memory that the FPGA as well as the CPU can read and write.

5.2.4 Threat Models

We consider two general threat models with two variants each, as illustrated in fig. 5.1. All four threat models include a malicious actor that can program and control a fast and programmable PCIe device (referred to in this section as the monitoring device) with direct memory access (such as an FPGA or GPU) and an IOMMU providing address translation services for that device. Each model also includes a second peripheral (referred to in this section as the sending device) which also uses the same IOMMU for DMA address translation but does not need to be fast or directly programmable as part of the threat model. The monitoring device must be capable of timing memory

accesses and reliably differentiate IOTLB hits from misses. The attacker must further be able to program the monitoring device directly to find eviction sets and execute Prime+Probes. The sending device only needs to have memory access patterns that can be triggered by a user, either by direct control, or triggerable through an application or system interface.

Models $1k$ and $1u$ are adversarial threat models for a *side-channel attack*, where a malicious user in control of the monitoring device exploits IOTLB contention to gain secret information from another user’s application that triggers memory accesses in the sending device. Models $2k$ and $2u$ outline the requirements for a *covert channel* with cooperative sending and monitoring devices, where colluding malicious users in control of applications in separate security domains uses the IOTLB to transmit data covertly across the two devices. Models ending in k include kernel access alongside the monitoring device, and models ending in u do not. Kernel access is necessary to implement an IOTLB flush through a custom kernel module as outlined in section 5.2.3. In section 5.3 we show how fine-grained flushing control allows for more reliable eviction set construction. However, eviction set construction and Prime+Probe-based IOTLB side-channel attacks are still possible without flushing capabilities.

Whereas some side-channel attacks can be carried out with JavaScript from a web browser against a personal computer, we consider cloud environments as the primary site of IOTLB attacks, since the attacker must already have control of a peripheral. Renting a single GPU or FPGA in a cloud environment

is easy; the primary logistical challenge of setting up a practical IOTLB side-channel or covert channel is IOMMU co-location – that is, ensuring that the monitoring device shares an IOMMU (and IOTLB) with the sending device. However, research into similar problems, like co-locating cloud instances for cache attacks, has yielded strategies for co-location that can be adapted to the IOTLB channel. İnci et al. [80] demonstrated two reliable co-location techniques for last-level caches that rely only on basic cache contention and so could be adapted to the IOTLB relatively easily. In a cooperative (covert channel) scenario, the sender instance sends a predetermined signal and the receiving instance searches the channel for a signal and attempts to match it with the agreed-upon signal. In an adversarial (attack) scenario, the attacker first chooses a target program and profiles it locally to learn to identify the traces it leaves. Then the attacker searches for such traces. For cache profiling, co-location is not necessary; the target program can be profiled within a single instance. In the case of IOTLB profiling, covert channel co-location may be used to first co-locate the cloud instance controlling the monitoring peripheral with another cloud instance that runs the target program which relies on a sending peripheral.

5.3 Constructing Eviction Sets

5.3.1 Initial IOTLB Organization Hypothesis

Initially, we hypothesized that the IOTLB would be organized like the CPU TLBs reverse-engineered in [58], with 2^s sets where s is an integer, some small number of ways per set, and a set mapping algorithm wherein the lowest s bits of the page address select the set number or some other combination of various bits of the page address forms the set number that the page is associated with. Initial experiments on all three systems showed that 128-address eviction sets of any randomly allocated pages reliably evicted any other single page, so we hypothesized that the IOTLB was organized with 128 sets and 1 way. We tested this hypothesized eviction set architecture in a scenario on *a10l* where the FPGA used Prime+Probe to monitor an IOTLB that it shared with a network card.

Figure 5.2 shows the hardware and software setup for this test, an example of threat model *1u*. A virtual machine is configured with the IOMMU in a pass-through mode (Virtual Function I/O or VFIO) to allow a Realtek 8168 NIC direct access to the virtual environment, where it uses the standard r8169 drivers. The test application runs directly on the host, and uses the Broadcom BCM57416 NIC to exchange packets with the Realtek NIC over ethernet. The test application also manages our Prime+Probe hardware on the Arria 10 GX FPGA and uses it to collect IOTLB side-channel traces while the network is active. The eviction sets used in the Prime+Probe tests

are constructed under the assumption that the IOTLB contains 128 sets of one way each.

Prime+Probe data from this experiment are visualized in fig. 5.3. There was substantial variation of IOTLB activity after a reboot of the virtual machine operating the Realtek NIC, so results are plotted as means across many reboots. More evictions were detected in the probes of the Prime+Probe while the network was active, indicating a side-channel leakage in the IOTLB that originated from the Realtek NIC. There are two other phenomena of note that are observable in the data from this experiment. First, the excess evictions caused by the network activity (shown in blue in the figure) varied substantially in the number of sets they occupied. Whenever the virtual machine was rebooted, the number of sets that were evicted during network activity changed, but there were always evictions in one set (set 11). After examining the network driver source code, we found that it allocates the transaction buffers used by the network card by calling a kernel function `dma_map_single` on startup, and we verified that by unloading and reloading the network driver, we could reproduce the randomizing effect of rebooting the virtual machine. Second, sets 1-10 and 126-128 were always evicted in the probe, even absent any network activity or with the network drivers unloaded. This showed that the 128-page eviction sets, while effective in evicting IOTLB entries, were actually bigger than necessary, since they were evicting their own members.

Table 5.2: Notation used in algorithms.

Symbol	Meaning
$A \leftarrow B$	A gets the value of B
$A \leftarrow_{\in} B$	A chosen randomly from B
$A \leftarrow_{+} B$	B added to the set A
$A \leftarrow_{-} B$	B removed from the set A
$A \leftarrow_{/} B$	Elements in B removed from A

5.3.2 A New Approach to Eviction Set Construction

We developed a novel and platform-independent algorithm for finding eviction sets for any TLB or cache where the timing difference between a present entry and an evicted entry is known and measurable. Our approach is inspired by the baseline reduction algorithm in [203], which only reduces an already existing eviction set to its minimum necessary size, and the grow-split eviction set construction approach of Algorithm 1 in [126].

Like [126], our algorithm constructs eviction sets from a large pool of addresses by gathering candidates for an eviction set and then systematically discarding unnecessary ones; addresses not present in candidate eviction sets are used as test targets. The grow-split algorithm in [126] is specifically designed for a partitioned cache: it first constructs an eviction set for the entire cache, and then splits it into separate sets for each of the partitions. Our grow-reduce algorithm makes no assumption about cache organization, and uses a more generalized approach of building one eviction set at a time by adding addresses until evictions are reliable and then testing which addresses


```

1 Function evicts(target, evset)
   input   : target – address to be evicted
             evset – eviction set used for eviction attempt
   output  : True, if 100 eviction attempts are successful
             False, otherwise
2   count ← 0 // # of contentions
3   for  $0 \leq i < 100$  do
4       flush IOTLB // optional
5       target_prime()
6       evset_prime()
7       time ← target_probe()
8       if time  $\geq$  threshold then
9           count ← count + 1
10  return count == 100

```

Algorithm 2: The algorithm tests whether a given eviction set evicts a given target address from the IOTLB. The `target_prime` and `evset_prime` function calls have the FPGA access the respective set of addresses. The function call `target_probe` has the FPGA time the access time to the target address.

can be discarded without losing reliability. It aims to create an exhaustive set of eviction sets by searching the entire address pool; redundant sets are avoided by ensuring that potential test targets are not already reliably evicted by another set.

Grow-Reduce Algorithm

The most basic function in our algorithm tests whether or not a hypothetical eviction set evicts a given target address (see algorithm 2). The software uses the hardware module described previously to perform a prime and probe test. First, the FPGA accesses the target followed by an access to each address in the eviction set. Then the target is accessed again and the

```

1 Function constructEvset(target, pool)
   input   : target – target address to be evicted
             pool – address pool
   output  : evset – an eviction set for target
2   evset  $\leftarrow \emptyset$ 
3   count  $\leftarrow 0$  // # of contentions
   // Grow
4   while count  $\leq 50$  and  $|pool| \neq 0$  do
5       page  $\leftarrow_{\in}$  pool; evset  $\leftarrow_{+}$  page; pool  $\leftarrow_{-}$  page
6       if evicts(target, evset) then
7           count  $\leftarrow$  count + 1
   // Reduce
8   foreach page in evset do
9       evset  $\leftarrow_{-}$  page
10      if not evicts(target, evset) then
11          evset  $\leftarrow_{+}$  page
12  return evset

```

Algorithm 3: The algorithm constructs an IOTLB eviction set for a given target address. The addresses for the eviction set are chosen from the given address pool.

access latency is measured. We define that an eviction set evicts a target if the latency of the second access to the target is above a certain threshold. We choose the threshold in the middle of the observed latency gap between fast and slow accesses observed on the different systems. Before each prime and probe test, we optionally cleared the IOTLB.

The construction of an eviction set for a fixed target address is given in algorithm 3. It takes a target address and a pool of addresses as inputs. The eviction set is initialized as an empty set. During the "grow" step random addresses are chosen from the address pool and added to the eviction set until the eviction set contains enough addresses to evict the target. Obviously, the eviction set may contain unnecessary addresses at this point. This is why

```

1 Function evsetFinding(poolSize)
   input    : poolSize – number of addresses to be allocated
   output   : evsets – Eviction sets for the IOTLB
2   pool ← alloc(poolSize)
3   targets ← ∅
4   evsets ← ∅
5   while poolSize ≠ 0 do
6       target ←∈ pool // Random page as target
7       pool ←- target
8       if evsets do not evict target then
9           targets ←+ target
10          evsets ←+ constructEvset(target, pool)
11          pool ←/ evsets
12      poolSize ← size(pool);
13 return evsets

```

Algorithm 4: This algorithm constructs as many eviction sets as needed to evict any target address from the IOTLB. The algorithm takes an integer as input that indicated the size of the address pool that is used to construct the eviction sets. A pool size of 4096 was used for the tests in this paper.

a reduction step follows where each address is tested for its necessity. If an address is not needed, it is removed from the eviction set and put back in the address pool.

At the highest level, our algorithm shown in algorithm 4 automatically constructs as many eviction sets as it can find. The program first allocates a pool of memory pages. For our experiments we used a pool size of 4096 addresses. The algorithm manages two sets: The *targets* set is used to store the different target addresses used during eviction set construction. The *evsets* set stores all eviction sets constructed by the algorithm. After this initialization step, the algorithm picks a random target address from the pool and removes it from the pool. If *evsets* does not contain an eviction set for

the target address yet, a new eviction set is constructed. The target address and the new eviction set are added to their corresponding sets. All addresses in the newly constructed eviction set are then removed from the pool. This procedure is repeated until the pool does not contain any addresses anymore.

Evaluation of New Eviction Set Algorithm

We found that the optional flushing of the IOTLB has an impact on the size and reliability of IOTLB eviction sets.⁴ The major differences are laid out in table 5.3, which enumerates general performance metrics of eviction sets constructed with our grow-reduce algorithm and [126]’s grow-split algorithm both with and without flushing. Enabling IOTLB flushes before the Prime+Probe step will make both algorithms return a single eviction set containing 118 addresses. The success rate of such eviction sets is 100% in every case we observed.

Without IOTLB flushes, neither algorithm produces such consistently sized or reliable eviction sets. This is likely due to a replacement policy that we were unable to deduce. In this scenario we can better see the advantage of our grow-reduce algorithm. It produces eviction sets that are both smaller and twice as reliable than those produced by the grow-split algorithm.

Figure 5.4 visualizes in detail the results of further experimentation with small implementation tweaks in our algorithm. In these experiments we found that the size and number of eviction sets constructed were very similar on

⁴Flushing the IOTLB requires kernel access; see threat models *1k* and *2k* in section 5.2.4. For this reason, table 5.3 contains data only from experiments on the *a10l* system.

Table 5.3: Comparison of eviction set finding algorithms on the IOTLB of the *a10l* test system. All tests were conducted on the *a10l* system using pools of 4096 addresses, and repeated 40 times. Eviction set orders were randomized between prime and probe steps during testing.

Flush	Algorithm	# of sets	Set size	Useful sets per target	Average best eviction rate
enabled	Grow-Reduce (this work)	1.00	118.00	1.00	100.00 %
	Grow-Split ([126])	1.00	118.00	1.00	100.00 %
	Grow-Reduce (this work)	32.08	110.05	0.98	82.23 %
disabled	Grow-Split ([126])	10.70	50.69	0.98	28.00 %

all tested systems, *a10l*, *a10v*, and *s10v*. We thus conclude that the IOTLB architecture on all tested systems is very similar in terms of IOTLB size, organization and replacement policy.

5.4 Analysis of Side-Channel Leakages

We now use the constructed eviction sets to further investigate the amount of leakage from PCIe devices observable in the IOMMU. Though we use the FPGA for channel monitoring outside of a virtualized environment for simplicity’s sake, this channel still poses a threat from one virtual environment to another or from a virtual environment to hypervisor. Major cloud platforms like AWS and Alibaba Cloud now allow users to rent direct access to FPGAs with DMA capabilities, meaning that malicious tenants could easily run hardware designs that monitor the IOMMU side-channel without root privileges. Any other PCIe devices that are co-located on the IOMMU with a malicious FPGA and using translated DMA (most modern devices use DMA, and virtualized DMA always requires translation if the IOMMU is shared) are sources of leakage and therefore potential attack targets. We focus our analysis on an in-memory SQL database accelerated by a graphics card.

5.4.1 Web Access Leakage

In section 5.3.1 we showed that the operation of a network card leaves traces in the IOTLB. With that, we set out to explore a common target for side-channel attacks: web fingerprinting. In a web fingerprinting attack, an attacker collects side-channel data while accessing various websites in a controlled environment and uses those data to build a model. Attacks have been built using a wide variety of data sources on a variety of platforms, including network traffic [73, 157], cache traces [70, 183, 187], and hardware performance events [69]. Then the attacker collects side-channel data from the victim and uses the model to predict the sites the victim was accessing.

We collected IOTLB Prime+Probe fingerprints while accessing a variety of websites from virtual machine using a PCIe network card co-located with the FPGA. As a browser, we used Firefox 88.0, the stable version available from the standard Ubuntu repositories at the time of data collection. We also collected side-channel data while the browser was inactive. Periodically, data collection was paused so the network card drivers could be reloaded. This triggers a reallocation of the relevant transmission buffers as described in section 5.3.1; an attack on this network card to be practical it would have to work after any driver reset. IOTLB probing generates an extremely large amount of data, even when artificially slowed. We collected traces at 4,000 probes per second, or one probe every 250 microseconds. As a result, the data requires some pre-processing before a machine learning model can effectively learn the fingerprints of the sites.

First, each timing measurement is converted to a logical 1 or 0, 1 representing a probable eviction (a slow memory access), and 0 representing a faster access to an address that was not evicted. This binary signal is still quite information-dense, but a spectrogram reveals patterns in its frequency content that shift over time and visibly vary between websites. The lowest frequency in the spectrogram was found to be extremely noisy compared to all others and was removed completely. The spectral data is then converted from amplitude to power and represented in decibels to tighten and normalize the distribution of data. Changes in the trace from one network driver reset to another must be accounted for in pre-processing as well. The mean of a random 70% of the idle signal (so that some variation from the remaining 30% could be observed) after a given reset of the network driver is subtracted from each other sample taken after that same reset. Because of the nature of the decibel scale, data points with zero power correspond to negative infinity decibels, which is inconvenient for batch comparisons of data, so all points below -100 decibels were raised to -100 decibels. Finally, all the samples were averaged; the interested reader may inspect the mean spectral data for each class after pre-processing in chapter [A](#).

Some distinguishing features of various sites are immediately clear in these spectrograms, like the relative absence of IOTLB activity when there is no network activity, a vertical band (broad spectrum signal across a short period of time) of activity early in the signal for all the sites followed by horizontal bands (consistent activity in certain frequency ranges across

time). With a sophisticated machine learning model like those used in other web fingerprinting attacks [70, 171], these traces could likely be classified automatically.

5.4.2 GPU-Accelerated SQL Database Leakage

We now inspect the amount of IOTLB leakage observable from the FPGA when it is co-located with a GPU that runs an SQL database. For our tests, we co-locate the FPGA with an NVIDIA Tesla T4 GPU that runs the OmniSci SQL server on it. We wish to understand the data leakage patterns of the GPU-accelerated database application, so for these experiments we consider threat model *1k*, where the attacker has the most precise control over the channel. Figure 5.6 shows a stack diagram of the setup on our *a10l* platform. The test application interacts with our hardware module on the FPGA to construct, prime and probe an eviction set for the IOTLB. Additionally, the application can issue SQL queries to the database which computes the result on the GPU.

After constructing an eviction set for the IOTLB, the test app primes the IOTLB. During the waiting phase, the app runs an SQL query on the GPU. The tested queries differ (significantly) in the size of the returned results. After the SQL result is returned to the test application, the FPGA probes the IOTLB and reports the access latency back to the application.

Figure 5.7 (b) - (d) show probe measurements for queries returning no, one

and 409,600 rows⁵ of data from the database. During the measurement shown in fig. 5.7 (a), no query was executed on the GPU. The separate access times for each eviction set address are plotted along the x-axis. The y-axis shows the measured latency for this address. Clearly, the GPU leaves a footprint in the IOTLB when it computes an SQL query. But, there is no measurable difference between the queries even if their results significantly differ in size.

Changing the test app to probe the eviction set while the SQL query executes on the GPU shows that the observable activity in the IOTLB is similar for all queries over time, besides the fact that queries with larger results produce longer traces as it takes longer to compute the result. Interestingly, the activity in the IOTLB happens towards the beginning of the query's computation. At the time where the computed result is sent back to the CPU, there is no activity in the IOTLB. This is easily explained by the way CUDA realizes the data transfer of the result from the GPU to the CPU: it uses MMIO⁶ instead of DMA⁷. We verified the explanation by inspecting the PCIe performance counters with the PCM tools⁸. The performance counters showed an increased amount of MMIO read requests that in total match the size of the returned result.

⁵One row in our case contains 36 bytes of data.

⁶The CPU initializes the data transfer.

⁷The peripheral initializes the transfer.

⁸pcm-pcie – <https://github.com/opcm/pcm>

5.4.3 Side-Channel Impact

So far, the observed leakage introduced by the IOTLB is mostly limited to a single bit describing whether a neighboring accelerator is in use or not. This is caused by two facts:

(a) Controlling an accelerator via MMIO rather than through DMA is a common usage model and limits the attack surface for IOTLB-based side-channel attacks because the CPU performs the address translation in the CPU’s MMU instead of the GPU translating addresses via the IOMMU.

(b) Current PCIe devices usually perform DMA as bulk transfers, thereby limiting the overall PCIe protocol overhead. Loading data in a bulk transfer into device memory, computing on the data locally and eventually transferring the result back to the main memory in a bulk transfer means that no data-dependent access patterns – which would leak information – are observable in general.

The two facts mentioned will likely change in the near future as PCIe 5.0 is rolled-out and Compute eXpress Link (CXL) is introduced.⁹ PCIe 5.0 reaches transfer speeds that are comparable with CPU main memory accesses. This may lead device developers to include smaller memory on their devices and in turn access the main memory more often. Furthermore, CXL features a coherency protocol that streamlines caching between main memory and PCIe device memory. Again, this will lead device and driver developers

⁹AMD CPUs and Intel FPGAs supporting CXL are already available. Intel plans rolling out compatible CPUs in the beginning of 2023 [16, 186].

to change from bulk transfers to more fine-grained data-dependent DMA accesses.

In addition, FPGA vendors keep pushing for FPGA devices being the first-class compute device in a system while the CPU is merely used to manage the system and provide the FPGA with (increasingly sensitive) data. Therefore, while the described side-channel is not yet very dangerous at the time of writing, it will become important in the near future. We highlight the side-channels existence and relevance *before* widespread deployment of CXL and PCIe 5.

5.5 Covert Channels

After identifying the IOTLB leakage and different ways to trigger and observe it, we now use our knowledge to construct two covert channels to prove the practicality of the channel with threat models $\mathcal{2u}$ and $\mathcal{2k}$. The first channel is constructed between two peripherals and requires user privileges and DMA access to pages in main memory (model $\mathcal{2u}$). This channel could be implemented between two virtual machines, each with control of a DMA-enabled peripheral, such as Amazon’s F1 FPGA instances or various GPU-enabled EC2 instances, as long as the two instances’ peripherals share an IOMMU. The performance of the covert channel can be improved if the receiver has root access on the host. The second channel is unidirectional from CPU to peripheral and requires the sender to have root access to the

host machine (model $2k$), thereby mostly serving as a proof of concept. For both channels, the receiver must be able to measure time, e. g. through precise internal timers or high-speed network connection with external timers. This is the case for, e. g. GPUs [49], NICs and FPGAs. All experiments in this section were run on the a10l system.

Table 5.4: Throughput and error rate for the covert channels tested on the *a10l* system. For the peripheral-peripheral channel, sender and receiver are perfectly synchronous. The channel itself is very reliable which leads to nearly no errors. The throughput depends on the number of 1-bits in the message as each 1-bit is encoded into running a SQL-query on the sender peripheral which takes a rather long time of 0.3 seconds. For the CPU-peripheral channel, sender and receiver are not perfectly synchronous which leads to the rather high error rate. The throughput is limited by the speed of the CPU flushing the IOTLB. For both channels, plain bits were sent without encoding.

	Sender	Receiver	Method	Environment	Throughput	Error rate	Content of message
Section 5.5.1	Peripheral	Peripheral	Prime+Probe	Bare metal	3.4 bps	0%	All 1s
					6.65 bps	0%	Even mix of 1s and 0s
					246.15 bps	0.1%	All 0s
					7.58 bps	0%	ASCII
Section 5.5.2	CPU	Peripheral	Flush+Reload	Bare metal	15023 bps	30.09%	<i>Performance not dependent on content</i>

5.5.1 Covert Channel between Peripherals

Another research question is whether two peripherals can use the IOTLB to construct a covert channel between each other. To answer this question, we co-locate the Arria 10 with the Tesla T4. Our goal is to use the footprint that an SQL query computed on the GPU leaves in the IOTLB to send information to the FPGA. Such a covert channel exists in a scenario where the sender

uses a website that, depending on the actions performed on the website, runs SQL queries on a GPU-accelerated database. The sender can then exploit the website to send information to the co-located FPGA.

We prepare a10l as shown in fig. 5.6. The sender encodes a one into running an SQL query and running no query encodes a zero. The receiver uses the `iotlb_pnp` hardware function on the FPGA to monitor the IOTLB using the Prime+Probe technique. Each SQL query evicts 18-20 entries of the receiver’s eviction set (cf. fig. 5.7 (b) - (d)). A plot of the number of IOTLB misses measured during message transmission is given in fig. 5.8a. We found that basically no errors occur if sender and receiver are synchronized. This means that the channel is nearly free of bit-flip errors. If perfect synchronization is not achievable, the channel suffers from insertion and deletion errors. In this case techniques from [131] can be applied to overcome these errors. The channel’s throughput highly depends on the number of one bits in the message. This is because the execution time of a single SQL query takes about 0.3 seconds. Table 5.4 shows more detailed measurements for different 0-1-ratios in the message that is transferred over the covert channel. Of course, a GPU application optimized for acting as a sender in this scenario would allow us to increase the bandwidth of the channel.

For the previous test, the eviction set used by the FPGA was constructed with IOTLB flushes to work with eviction sets of optimal reliability. As mentioned earlier, IOTLB flushes require kernel privileges on commodity host Linux systems. User-level receivers or receivers located in virtual machines

have to use the less reliable eviction sets constructed without IOTLB flushes. As can be seen in fig. 5.8b, this results in more noise in the measurements. The depicted transmission is still free of errors but some bits are at the edge of being falsely classified. To overcome potential bitflip errors, error detection mechanisms like CRC codes or error correction codes like Hadamard codes can be applied [131]. The presented covert channel works between any two peripherals that use DMA to access the main memory. For the receiver, the accessible memory region needs to be sufficiently large to allow for eviction set construction. Additionally, the receiver needs a mechanism to measure the memory access latency. Programmable or configurable peripherals like FPGAs or GPUs will meet both receiver requirements even in the most stringent cloud environments if bare metal instances are available for rent. An FPGA or GPU sender has fine-grained control of the channel, but a more opaque sender like a smart NIC or PCIe-enabled storage device could work as a sender, albeit more likely to be noisy or unreliable.

Peripherals that manage secrets and perform DMAs depending on the value of the secret must be aware that neighboring devices connected to the same IOMMU may be able to observe their access patterns. This is especially true for peripherals where the programming model assumes unified memory that abstracts separate physical memory locations like device and system memory away from the developer as in this case the leaking DMA may occur without the knowledge of the developer. As of today, data-dependent DMA is used seldomly due to the overhead that renders it inefficient. But we

expect this behavior to change with the introduction of PCIe 5.0 and CXL as mentioned in earlier sections.

5.5.2 Covert Channel from CPU to Peripheral

The CPU is very limited in interacting with the IOTLB directly. Because the IOMMU translates addresses for peripherals only, memory accesses from the CPU do not interfere with the IOTLB. The only way for the CPU to interfere with the IOTLB is by changing page table entries or instructing the IOMMU to flush certain (or all) entries in the IOTLB. Usually, only the OS, hypervisor or VMM issues page table changes or IOTLB flushes, which is why the Linux kernel does not provide an interface for flushing the IOTLB to userland. To overcome this problem, we load a self-developed kernel module that exposes a IOTLB flush API to our test application. An overview of our system setup for this covert channel is given in fig. 5.6.

Since a peripheral can distinguish IOTLB hits from misses, flushing the IOTLB allows the CPU to send information covertly to peripherals. A global IOTLB flush takes $17\mu\text{s}$ on average. Flushing all entries from the IOTLB encodes a 1 and sleeping for $17\mu\text{s}$ encodes a 0. As the receiver we use the `iotlb_pnp` hardware module described in section 5.2.3. The hardware function is programmed to continuously probe a fixed target address. Whenever a probe reports a slow access, a 1 is received. Otherwise, the hardware receives a 0. We implement the covert channel in a trivial way without applying any encoding for error correction or synchronization. Because a memory access from the

FPGA running at 200 MHz only takes around $1\ \mu\text{s}$ we roughly synchronize the FPGA with the CPU by making the FPGA `wait` for a certain amount of cycles. We determined the number of cycles to wait by repeatedly flushing the IOTLB and increasing the number of wait cycles until all FPGA memory accesses are slow. After this very rough synchronization step, a message of $2^{16} - 1$ bits generated by a linear feedback-shift register is transmitted to measure throughput and error rates. The result is given in table 5.4. As can be seen, this basic covert channel without further optimizations already achieves a throughput of around 15 kBit/s. The error rate is 30% which can be improved significantly by applying error-correction and error-handling techniques as e. g. described in [131].

Because so far the covert channel only offers communication in one direction, we tried to improve the channel to offer bi-directional message transfer. To do so we checked the timing behavior of flushing the IOTLB. The `clflush` instruction on x86 CPUs has a data-dependent execution time [66]. In our case, a data-dependency of the flush time on IOTLB entries would allow us to construct the reverse covert channel. However, our experiments show no measurable timing behavior of the flush that can be related to the usage of the IOTLB; an IOTLB flush takes around $17\ \mu\text{s}$ independent of FPGA memory accesses before or even during the flush. The latency is also independent from whether only addresses of a certain peripheral or all entries of the IOTLB are flushed. However, peripheral-to-CPU covert channels based on the CPU cache do exist [207].

The demonstrated covert channel is reliable without applying special synchronization, error-correction, or error-detection techniques. However, only peripherals can act as the receiver while the CPU is limited to the role of the sender. Also, with the standard IOMMU drivers in Linux, the sending process is required to run kernel-level code to perform IOTLB flushes. A privileged device driver that flushes the IOTLB under certain circumstances may expose this flushing capability to an unprivileged user. Device drivers that make extensive use of IOTLB flushes may also be vulnerable to a side-channel attack from an untrusted peripheral device that monitors the IOTLB for flushes. For example, a driver developer may chose to include IOTLB flushes to remove traces of a trusted peripheral’s activity for security; however, the timing between flushes could leak information about the operation of an application using that peripheral.

5.6 Countermeasures

Like many microarchitectural attacks, there are a variety of defenses against IOTLB side-channels that can be implemented at nearly any level of a system. We first present immediately available actions that can be taken by system administrators and cloud application developers, and then discuss defenses that can be built into future IOMMU architectures.

5.6.1 Securing Existing Systems

In cases where multiple users who do not trust each other may use the same machine, ensuring that no two users (or no one user and the hypervisor) have access to peripherals on the same IOMMU hardware is sufficient to protect against IOTLB side-channel attacks. On a Linux host, `/sys/class/iommu/` provides information on a system's IOMMU devices and the PCIe devices that use them [169]. Typically, systems have several IOMMU devices, each of which is linked to a few PCIe endpoints, which may be internal PCIe devices or external devices plugged into PCIe slots on the motherboard. Endpoints cannot be reassigned to new IOMMUs, so ensuring full isolation may limit scaling capacity. For example, a CSP could not use a motherboard with eight full-size, full-speed PCIe slots managed in pairs by four IOMMUs to provide eight fully isolated single-GPU cloud instances, even though eight GPUs fit in the PCIe slots of the system.

On the *application level*, code and hardware involved in data dependent computation can rely on constant time algorithms with constant memory access patterns, so no information about the operations is leaked through the IOTLB. For cryptographic implementations this is a common technique but for database systems constant memory access patterns and timings are not easily achieved. Private Information Retrieval (PIR) protocols [41, 119] can be a solution, but modern implementations¹⁰ usually only support index queries. Recent attempts [72] to also support range queries may still leak

¹⁰e. g. <https://github.com/ReverseControl/MuchPIR>

information about the response size.

A *hypervisor* can enable Address Translation Services (ATS) [160] for a peripheral to remove all of its traces from its IOTLB. Address Translation Services (ATS) allows a device to maintain and use a local on-device TLB for address translation and selectively bypass IOMMU translation. Since locally-translated requests are not translated by the IOMMU, they do not leave any trace in the IOTLB. However, devices must specifically support ATS to use it, and furthermore, allowing ATS for untrusted devices is not advisable.

ATS allows a device to provide *any* physical address as part of a DMA request and mark it as “translated”. Malicious devices may exploit ATS for unrestricted physical memory access [129]. Therefore, ATS must only be allowed for trusted devices.

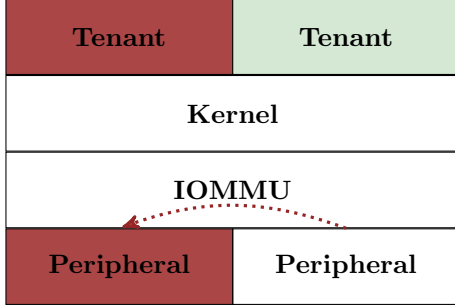
Hypervisors can also achieve a separation of the IOTLB between mutually untrusted tenants by IOTLB partitioning. For set-associative IOTLBs, set partitioning can be done by the hypervisor in software by only allocating I/O virtual addresses of sets to each tenant [219]. However, set-based partitioning may not work with peripherals that rely on the address space being contiguous.

5.6.2 Securing Future IOMMUs

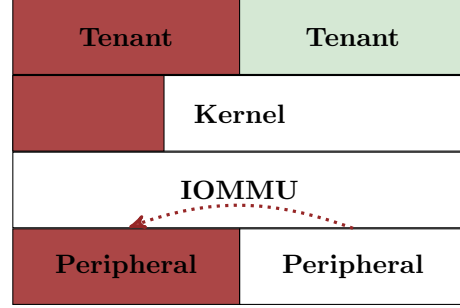
If *hardware modifications* are a viable option to implement countermeasures, then way-based partitioning is another option. It needs to be supported by the IOMMU hardware so that the hypervisor can map each address of a

thread to a fixed number of ways like is possible with Intel CAT [125, 150] for CPU-internal caches.

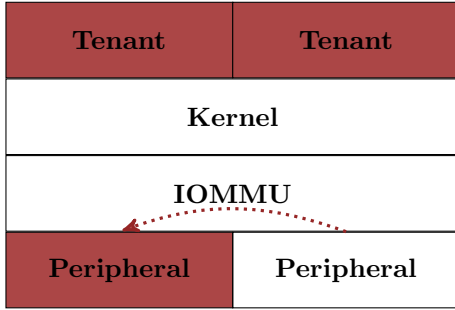
Future IOMMUs could include support for flagging a page translation as uncacheable. This would ensure that it is never stored in the IOTLB and that the use of that page would never affect the IOTLB state, so it would be invisible to any side-channel attack. However, all accesses to that page would be as slow as IOTLB misses, increasing latency and likely reducing maximum throughput.



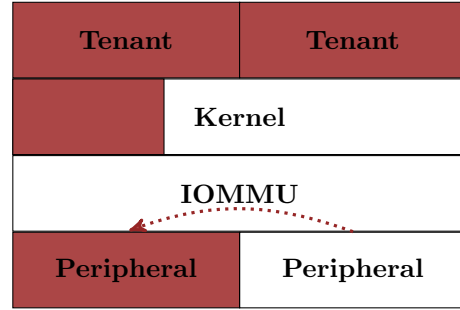
(a) Model *1u*. Side-channel attacker with *user* privilege.



(b) Model *1k*. Side-channel attacker with *kernel* module.



(c) Model *2u*. Covert channel with *user* privilege.



(d) Model *2k*. Covert channel with *kernel* module.

Figure 5.1: Comparison of threat models. Dark red fills indicate functional units controlled by a malicious actor, and light green fills indicate functional units controlled by a victim. Diagonal lines indicate functional units that are only under coarse or indirect control, e.g., a simple network interface card or an accelerator that assists with certain applications but is not directly programmable. The dashed arrows indicate the flow of data through the channel.

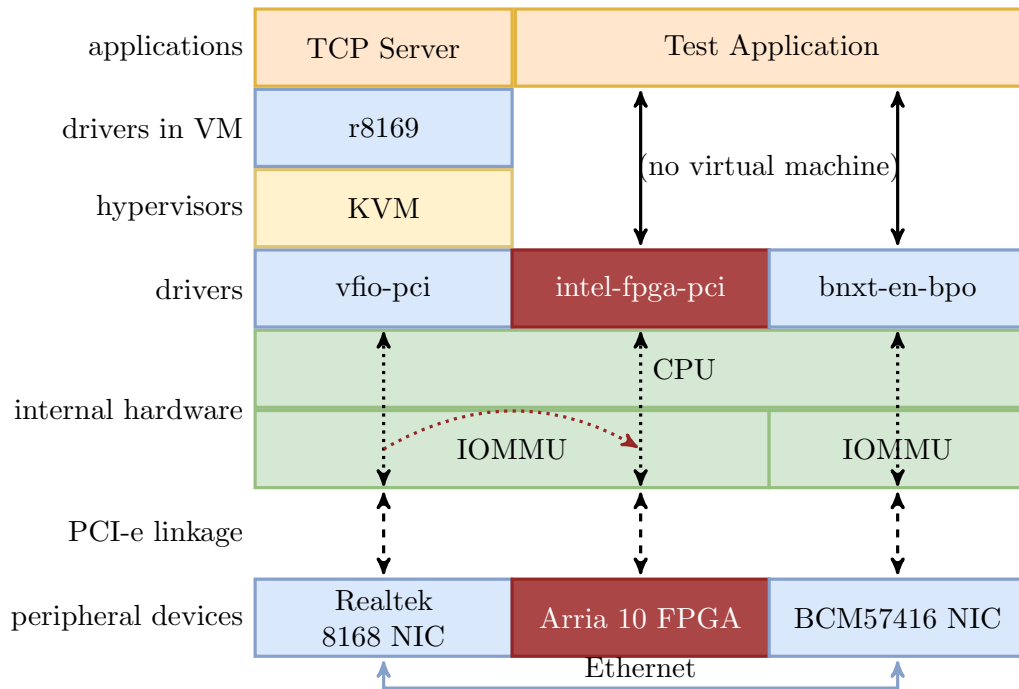


Figure 5.2: Stack diagram of the network card side-channel test. The Realtek network interface card (NIC) is "passed through" to a virtual machine with the VFIO driver. The test application exchanges packets with the TCP server in the virtual machine over the ethernet connection between the two network cards; meanwhile, the FPGA (connected to the same IOMMU as the VM's network card) probes the IOTLB for traces of network activity.

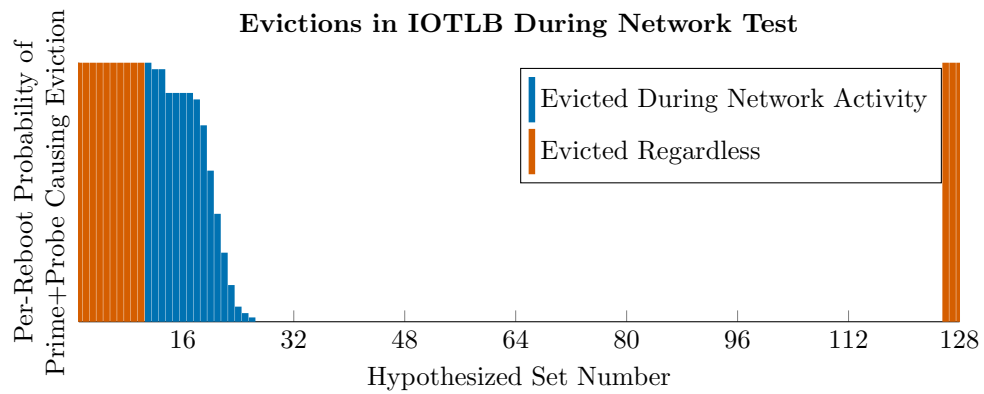


Figure 5.3: Behavior is consistent after a reboot of the virtual machine shown in fig. 5.2, but inconsistent between reboots; this graph shows the likelihood that an IOTLB entry will be consistently evicted by a Prime+Probe after a reboot of the system. Entries marked in red are evicted whether or not there is network activity and do not vary between reboots; entries in blue are those that are evicted when there is network activity but not when there is no activity and vary significantly.

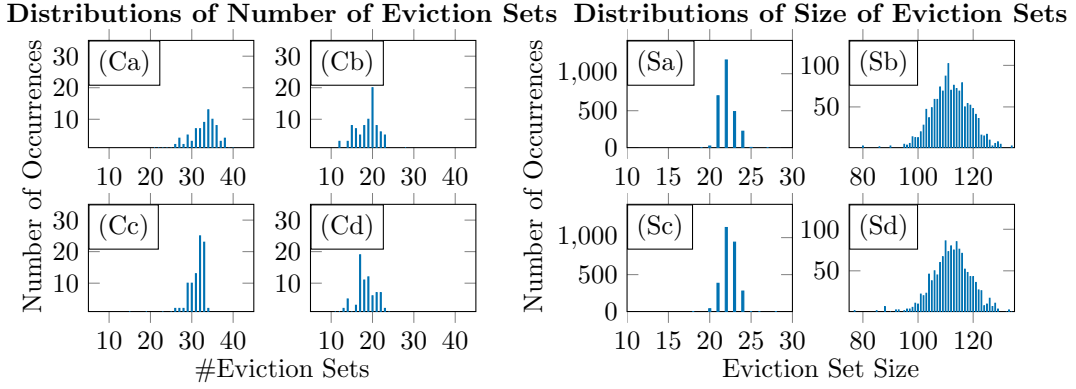


Figure 5.4: Number of eviction sets and the size of each constructed set needed to evict any target IOVA after running algorithm 4 for 100 times each. During eviction set construction, randomization of the eviction set was turned off for measurements (a) and (c) and turned on for (b) and (d). For measurements (c) and (d), the algorithm waited 100 ns between each eviction test. For measurements (a) and (b) this was not the case. If the order of accesses during the `evset_prime()` is static throughout one run of algorithm 4, the resulting eviction sets contain 20 to 25 addresses each. The average success rate is slightly below the average success rate of eviction sets constructed with randomized access order during `evset_prime()`. In turn, randomizing the access order yields on average slightly less but bigger sets. The success rate of these sets, with or without randomized access order, evict a target with probabilities above 90%.

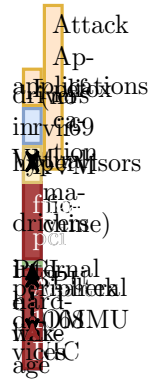


Figure 5.5: Stack diagram of the web access fingerprinting attack. The Firefox web browser runs on the VM and uses the network card with VFIO pass-through to access various websites. Meanwhile, the attacker periodically probes an IOTLB eviction set to collect fingerprints of network activity.



Figure 5.6: Stack diagram of the CPU to peripheral and peripheral to peripheral covert channel and side-channel tests.

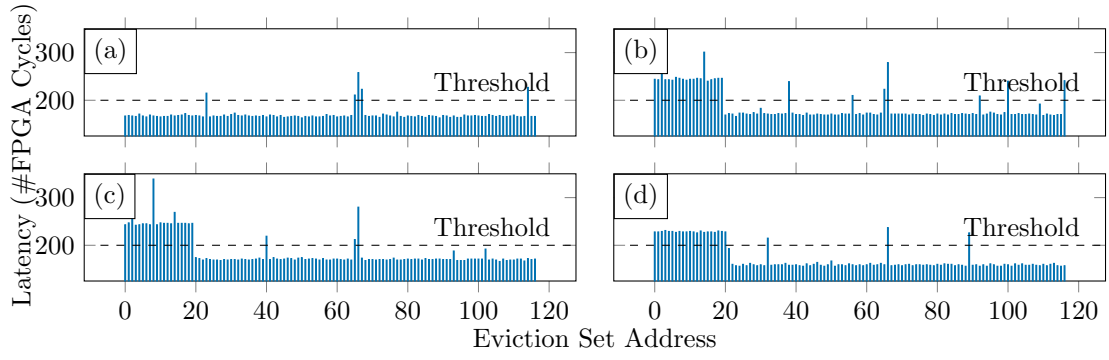
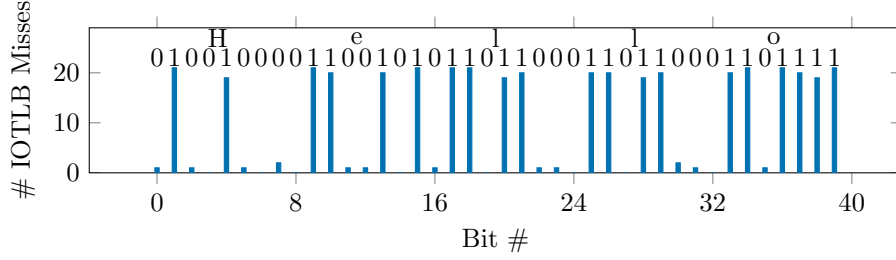
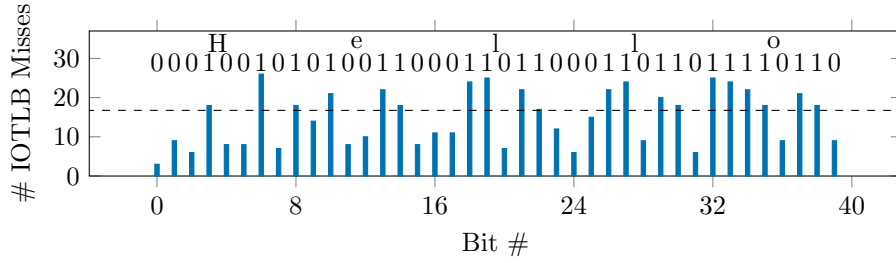


Figure 5.7: Measurements for the conducted experiments with the SQL database. During measurement (a), the test app did not run any query. The queries run in measurements (b) - (d) returned no, one and 409600 rows of data from the database. It is clearly visible that the SQL queries leave a footprint in the IOTLB.



(a) Scenario as in fig. 5.6; big endian transmission. The FPGA uses an eviction set that was constructed using IOTLB flushes. This results in very reliable eviction sets and in turn a reliable transmission.



(b) Scenario as in fig. 5.9; little endian transmission. The FPGA uses eviction sets constructed *without* IOTLB flushes. Even though the transmission is free of errors, it turns out to be more noisy.

Figure 5.8: Peripheral to peripheral covert channel transmissions. The message “Hello” was sent in big endian format.

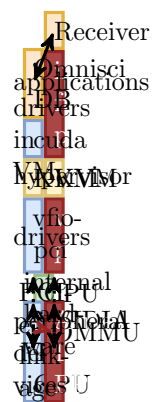


Figure 5.9: Stack diagram of the GPU accelerated SQL database covert channel across virtual machines.

Chapter 6

Microarchitectural Security of AWS Firecracker VMM for Serverless Cloud Platforms

6.1 Introduction

AWS claims that a Firecracker virtual machine running on a system with up-to-date microarchitectural defenses will provide sufficient hardening against microarchitectural attacks [2]. The Firecracker documentation also contains specific recommendations for microarchitectural security measures that should be enabled. *In this work, we examine Firecracker’s security claims and recommendations and reveal oversights in its guidance as well as wholly unmitigated threats.* After reviewing a pre-print of this paper, AWS has updated the Firecracker production host setup recommendations [14] to clearly acknowledge Firecracker’s inability to provide microarchitectural security for the host machine. The new recommendations also refer to operating system and hardware vendor guides which include corrections to specific oversights

we identified. In summary, our main contributions are:

- We provide a comprehensive security analysis of the cross-tenant and tenant-hypervisor isolation of serverless compute when based on Firecracker VM.
- We test Firecracker’s defense capabilities against microarchitectural attack proof-of-concepts (PoCs), employing available hardware and kernel protections. We show that the virtual machine itself provides *negligible protection* against major classes of microarchitectural attacks.
- We identify a variant of the Medusa MDS attack that becomes exploitable from within Firecracker VMs *even though it is not present on the host*. The mitigation that protects against this exploit is not mentioned by AWS’s Firecracker host setup recommendations. Additionally, we show that disabling SMT provides insufficient protection against the identified Medusa variant which urges the need of this mitigation.
- We identify Spectre-PHT and Spectre-BTB variants which leak data with recommended countermeasures in place. The Spectre-PHT variants even remain a problem when SMT is disabled if the attacker and victim share a CPU core in a time-sliced fashion.

6.1.1 Responsible Disclosure

We informed the AWS security team about our findings and discussed technical details. The AWS security team claims that the AWS services are not

affected by our findings due to additional security measures. AWS agreed that Firecracker does not provide micro-architectural security on its own but only in combination with microcode updates and secure host and guest operating systems. As such, the Firecracker developers have updated the microarchitectural attack section of the production host setup recommendations to make this very clear [14]. Updates to this section also include direct referral to OS and CPU vendor documentation on microarchitectural vulnerabilities and mitigations.

6.2 Threat Models

We propose two threat models applicable to Firecracker-based serverless cloud systems:

1. The *user-to-user* model: a malicious user runs arbitrary code sandboxed within a Firecracker VM and attempts to leak data, inject data, or otherwise gain information about or control over another user’s sandboxed application. In this model, we consider
 - (a) the time-sliced sharing of hardware, where the instances of the two users execute in turns on the CPU core, and
 - (b) physical co-location, where the two users’ code runs concurrently on hardware that is shared in one way or another (for example, two cores on the same CPU or two threads in the same core if SMT is enabled).

2. The *user-to-host* model: a malicious user targets some component of the host system, e. g., the Firecracker VMM, KVM, or another part of the host system kernel. For this scenario, we only consider time-sliced sharing of hardware resources because the host only executes code if the VM exits, e. g. due to a page fault that has to be handled by the host kernel or VMM.

For both models, we assume that a malicious user is able to control the runtime environment of its application. In our models, malicious users do *not* possess guest kernel privileges. Therefore, both models grant the attacker slightly fewer privileges than the model assumed by [2] where the guest kernel is chosen and configured by the VMM but assumed to be compromised at runtime. Rather, the attacker’s capabilities in our models match the capabilities granted to users in deployments of Firecracker in AWS platforms.

6.3 Analysis of Firecracker’s Containment Systems

fig. 2.2 shows the containment offered by Firecracker, as presented by AWS. In this section, we analyze each depicted component and their defenses against and vulnerabilities to microarchitectural attacks.

Kernel-based virtual machine (KVM) is the hypervisor implemented in modern Linux kernels that manages hardware virtualization of supervisor and user mode execution and context switches between VMs. Crucially, KVM’s

hardware virtualization includes address spaces for the guest kernel and guest user code that are separate from those of the host and of other guests—a significant barrier to both user-to-user and user-to-host attacks. Besides these architectural isolation mechanisms, KVM also implements mitigations against Spectre attacks on a VM-exit to protect the host OS or hypervisor from malicious guests. However, in contrast to most modern kernels, Firecracker guest kernels lack kernel ASLR support, making them especially vulnerable to microarchitectural attacks that ignore address space boundaries [76]. Since KVM is part of the Linux kernel, we define KVM to not be a part of Firecracker. Therefore, countermeasures against microarchitectural attacks that are implemented in KVM cannot be attributed to Firecracker’s containment system.

The *metadata*, *device*, and *I/O services* are the parts of the Firecracker VMM that interact directly with a VM. AWS touts the simplicity of these interfaces (for a reduced attack surface) and that they are written from scratch for Firecracker in Rust, a language known for its security features [21]. However, Rust most notably provides in-process protection against invalid and out-of-bounds memory accesses, but microarchitectural attacks can leak information between processes without directly hijacking a victim’s process.

Another notable difference between Firecracker and many other VMMs is that all of these services are run within the same host process as the VM itself, albeit in another thread. While the virtualization of memory addresses within the VM provides some obfuscation between the guest’s code and the

I/O services, some Spectre attacks work specifically within a single process. Intra-process attacks may pose less of a threat to real world systems, however, since two guests running on the same hardware each have their own copy of these services.

The *jailer* provides an additional barrier of defense around a Firecracker instance in the event that the API or VMM are compromised. It protects the host system’s files and resources with namespaces and control groups (cgroups), respectively [12]. Microarchitectural attacks do not threaten files, which are by definition outside the microarchitectural state. Cgroups allow a system administrator to assign processes to groups and then allocate and monitor system resource usage on a per-group basis [46]. It is plausible that limitations applied with cgroups could impede an attacker’s ability to carry out certain microarchitectural attacks which rely on the ability to allocate large amounts of memory or precisely measure the timing of CPU operations. In practice, Firecracker is not distributed with any particular cgroup rules [12]; in fact, it is specifically designed so that the default Linux resource allocation can run many VMs efficiently [11].

None of Firecracker’s isolation and containment systems seem to directly protect against user-to-user or user-to-host attacks. Thus, we proceeded to test various microarchitectural attack PoCs inside and outside of Firecracker VMs.

6.4 Analysis of microarchitectural attacks and defenses in Firecracker microVMs

In this section we present our analysis of a number of microarchitectural side-channel and speculative attack PoCs on Firecracker microVMs. We test these PoCs on the host and in Firecracker VMs, and test relevant microcode defenses in the various scenarios. We run our tests on a server with an Intel Skylake 4114 CPU which has virtualization hardware extensions and SMT enabled. The CPU runs on microcode version 0x2006b06¹. The host OS is Ubuntu 20.04 with a Linux 5.10 kernel. We used Firecracker v1.0.0, v1.4.0, and v1.5.0—the latest version as of Oct 2023—to run an Ubuntu 18.04 guest with Linux kernel 5.4 which is provided by Amazon when following the quick-start guide.²

In summary, the recommended production host setup provided with AWS Firecracker is insufficient when it comes to protecting tenants from malicious neighbors. Firecracker therefore fails in providing its claimed isolation guarantees. This is because

1. we identify a Medusa variant that only becomes exploitable when it is run across microVMs. In addition, the recommended countermeasures at the time of the research did not contain the necessary steps to

¹Updating the microcode to a newer version would disable TSX on our system which would make tests with TSX-based MDS variants impossible.

²<https://github.com/firecracker-microvm/firecracker/blob/dbd9a84b11a63b5e5bf201e244fe83f0bc76792a/docs/getting-started.md>

Table 6.1: Overview of discovered microarchitectural vulnerabilities not fully prevented by the recommended production host settings for AWS Firecracker prior to our disclosure.

Exploit Description	Firecracker only?	Cross-VM?	Mitigations
Medusa (CI ^a + block write secret)	✓	✓	<code>mds</code> (host)
Medusa (UStL ^b)	✗	✓	<code>mds</code> (host) + <code>nosmt</code>
RIDL/MFBDS (alignment fault)	✗	✓	<code>mds</code> (host)
RIDL/MFBDS (in-process)	✗	✗	<code>mds</code> (host or VM)

^a Cache Indexing variant

^b Unaligned Store-to-Load variant

mitigate the side-channel, or most other Medusa variants.

- we show that tenants are not properly protected from information leaks induced through Spectre-PHT or Spectre-BTB when applying the recommended countermeasures. The Spectre-PHT variants remain a problem even when disabling SMT.
- we observed no differences in PoC performance between the tested Firecracker versions.

We conclude that the virtualization layer provided by Firecracker has little effect on microarchitectural attacks, and Firecracker’s system security recommendations were incomplete prior to revisions prompted by this work.

6.4.1 Medusa

We evaluated Moghimi’s PoCs [136] for the Medusa [138] side-channels (classified by Intel as MLPDS variants of MDS [88]) on the host system and in Firecracker VMs. There is one leaking PoC for each of the three known

Table 6.2: Presence of Medusa side-channels with all microarchitectural defense kernel options disabled. Note that the combination of cache indexing leak and block write secret (highlighted) works in Firecracker VMs but not on the host.

Leak		Secret	Host	Firecracker
Cache Indexing	{	Block Write	\nRightarrow	\Rightarrow
		REP MOV	\Rightarrow	\Rightarrow
Unaligned Store-to-Load	{	Block Write	\Rightarrow	\Rightarrow
		REP MOV	\Rightarrow	\Rightarrow
Shadow REP MOV	{	Block Write	\nRightarrow	\nRightarrow
		REP MOV	\nRightarrow	\nRightarrow

\Rightarrow – Side-channel leakage is observable with all mitigations disabled.

\nRightarrow – Side-channel leakage is not observable.

variants described in section 2.7.2. We used two victim programs from the PoC library:

- The “Block Write” program writes a large amount of consecutive data in a loop (so that the processor will identify repeated stores and combine them).
- The “REP MOV” program performs a similar operation, but with the REP MOV instruction instead of many instructions moving smaller blocks of data in a loop.

Results

Table 6.2 shows the cases in which data is successfully leaked with all microarchitectural protections in the kernel disabled. The left two columns show

Table 6.3: Mitigations necessary to protect the host vs. Firecracker victims from Medusa attacks. Note that AWS’s recommended mitigation `nosmt` does not prevent the highlighted cache indexing/block write variant that is enabled by Firecracker (cf. table 6.2), or any other variants. All results were reproduced with Firecracker versions 1.0.0, 1.4.0, and 1.5.0.

Leak	Secret	Bare metal		Firecracker		
		mds	nosmt	mds(VM)	mds(H)	nosmt
Cache Indexing	Block Write	N/A	N/A	✗	✓	✗
	REP MOV	✓	✓	✓	✓	✓
Unaligned Store	Block Write	✓	✗	✗	✓	✗
	REP MOV	+	+	✗	+	+

✓ – mitigation prevents side-channel attack.

+ – mitigation prevents attack only in combination with other mitigation(s) marked +.

✗ – mitigation has no effect on this attack.

the possible combinations of the three Medusa PoCs and the two included victim programs. The right columns indicate which configurations work on the host and with the secret and leaking program running in parallel Firecracker instances. Most notably, with the Cache Indexing variant, the Block Write secret only works with Firecracker. This is likely because of the memory address virtualization that the virtual machine provides: the guest only sees virtual memory regions mapped by KVM, and KVM traps memory access instructions and handles the transactions on behalf of the guest. We found that the Shadow REP MOV variant did not work inside or outside of Firecracker.

We tested the effectiveness of `mds` and `nosmt` defenses against each com-

Table 6.4: Mitigations necessary to protect the host vs. Firecracker victims from RIDL and other MDS attacks. The recommended `nosmt` mitigation protects against most but not all of these variants. All proof of concepts were tested on Firecracker v1.0.0, v1.4.0, and v1.5.0 with identical results.

Exploit Details			Host		Firecracker		
Name	Target Buffer	Fault	<code>nosmt</code>	<code>mds</code>	<code>nosmt</code> (H)	<code>mds</code> (H)	<code>mds</code> (VM)
RIDL	Fill Buffer	Alignment	+	+	+	+	×
	Fill Buffer	Page	× ^b	✓	× ^b	✓	✓
	Fill Buffer	Page	✓	×	N/A ^c	N/A ^c	N/A ^c
L1DES	Fill Buffer	TSX abort	✓	×	✓	×	×
RIDL	Load Port	Page	✓	×	✓	×	×
RIDL/VRS	Store Buffer	Page	✓	×	✓	×	×
Crosstalk	Fill Buffer	Page	✓	×	N/A ^a	N/A ^a	N/A ^a

✓ – mitigation prevents side-channel attack.

⊕ – mitigation prevents attack only in combination with other mitigation(s) marked ⊕.

× – mitigation has no effect on this attack.

^a CPUID instruction used in this PoC is emulated by the VM and has no microarchitectural effect.

^b This attack leaks information about pages used in its own thread.

^c PoCs had to be modified to run in two processes before they could be tested in the virtual machine. These PoCs did not work on bare metal or in virtual machines when split into two processes.

bination of attacker and victim PoC on the host and in Firecracker VMs.

Table 6.3 lists the protections necessary to prevent Medusa attacks in the host and Firecracker scenarios. Across the four vulnerabilities in Firecracker, only one is mitigated by `nosmt` alone, and AWS does *not explicitly recommend* enabling the `mds` protection, though most Linux distributions ship with it enabled by default. That is to say, a multi-tenant cloud platform could be using Firecracker in a way that is compliant with AWS’s recommended security measures and still be vulnerable to the majority of Medusa variants,

including one where the Firecracker VMM itself leaks the user’s data that would not otherwise be leaked.

6.4.2 RIDL and More

In this section, we present an evaluation of the RIDL PoC programs [202] provided alongside van Schaik et al.’s 2019 paper [201]. RIDL is a class of MDS attacks that exploits speculative loads from buffers inside the CPU (not from cache or memory). The RIDL PoC repository also includes examples of attacks released in later addenda to the paper as well as one variant of the Fallout MDS attack.

Results

Table 6.4 shows some basic information about the RIDL PoCs that we tested and the efficacy of relevant countermeasures at preventing the attacks. Table B.1 in the appendix shows more details. We compared attacks on the host and in Firecracker to evaluate Amazon’s claims of the heightened hardware security of the Firecracker microVM system. For tests on the Firecracker system, we distinguish between countermeasure flags enabled on the host system (H) and the Firecracker guest kernel (VM). Besides the `nosmt` and `mds` kernel flags, we tested other relevant flags (cf. section 2.7.4, [75]), including `kaslr`, `pti`, and `l1tf`, but did not find that they had an affect on any of these programs. We excluded the `tsx_async_abort` mitigation since the CPU we tested on includes `mds` mitigation which makes the `tsx_async_abort` kernel

flag redundant [71].

In general, we found that the `mds` protection does not adequately protect against the majority of RIDL attacks. However, disabling SMT does mitigate the majority of these exploits. This is consistent with Intel’s [88] and the Linux developers’ [75] statements that SMT must be disabled to prevent MDS attacks across hyperthreads. The two outliers among these PoCs are `alignment.write`, which requires both `nosmt` and `mds` on the host, and `pgtable.leak.entsx`, which is mitigated only by `mds` countermeasures. The leak relying on `alignment.write` uses an alignment fault rather than a page table fault leak to trigger speculation [201]. The RIDL paper [201] and Intel’s documentation of the related VRS exploit [89] are unclear about what exactly differentiates this attack from the page-fault-based MFBDS attacks found in other PoCs, but our experimental findings indicate that the microarchitectural mechanism of the leakage is distinct. There is a simple and reasonable explanation for the behavior of `pgtable.leak.entsx`, which is unique among these PoCs for one key reason: it is the only exploit that crosses security boundaries (leaking page table values from the kernel) within a single thread rather than leaking from another thread. It is self-evident that disabling multi-threading would have little effect on this single-threaded exploit. However, the `mds` countermeasure flushes microarchitectural buffers before switching from kernel-privilege execution to user-privilege execution within the same thread, wiping the page table data accessed by kernel code from the line-fill buffer (LFB) before the attacking user code can leak it.

1	Vulnerability Spec store bypass:	Mitigation; Speculative Store Bypass disabled via prctl and seccomp
2	Vulnerability Spectre v1:	Mitigation; usercopy/swapgs barriers and __user pointer sanitization
3	Vulnerability Spectre v2:	Mitigation; Full generic retpoline, IBPB conditional, IBRS_FW, STIBP conditional, RSB filling

Figure 6.1: Spectre mitigations enabled in the host and guest kernel during the Spectre tests. This setup is recommended by AWS for host production systems [13].

In contrast to Medusa, most of these PoCs are mitigated by AWS’s recommendation of disabling `smt`. However, as with Medusa, the Firecracker VMM itself provides no microarchitectural protection against these attacks.

6.4.3 Spectre

While there have been many countermeasures developed since Spectre attacks were first discovered, many of them either come with a (significant) performance penalty or only partially mitigate the attack. Therefore, system operators often have to decide for a performance vs. security trade-off. To evaluate the wide range of Spectre attacks, we rely on the PoCs provided in [38]. For Spectre-PHT, Spectre-BTB, and Spectre-RSB, the repository contains four PoCs each. They differ in the way the attacker mistrains the branch prediction unit (BPU). The four possibilities are (1) *same-process*: the attacker has control over the victim process or its inputs to mistrain the BPU, (2) *cross-process*: the attacker runs its own code in a separate process to influence the branch predictions of the victim process, (a) *in-place*: the attacker mistrains the BPU with a branch instruction that resides at the same virtual address as the target branch, and (b) *out-of-place*: the attacker

mistrains the BPU with a branch instruction at an address that is congruent to the target branch in the victim process. The first two and latter two options are orthogonal, so each PoC combines two of them. For Spectre-STL, only same-process variants are known, which is why the repository only provides two PoCs in this case. For cross-VM experiments, we disabled address space layout randomization for the host and guest kernels³ as well as for the host and guest user level to ease finding congruent addresses that are used for mistraining.












































Results







With *AWS recommended countermeasures* [13]—the default for the Linux kernels in use—enabled on the host system and inside Firecracker VMs, we see that Spectre-RSB is successfully mitigated both on the host and inside and across VMs (cf. table 6.5). On the other hand, *Spectre-STL*, *Spectre-BTB*, and *Spectre-PHT* enabled information leakage in particular situations.


The PoCs for Spectre-STL show leakage. However, the leakage only occurs within the same process and the same privilege level. As no cross-process variants are known, we did not test the cross-VM scenario for Spectre-STL. In our user-to-user threat model, Spectre-STL is not a possible attack vector, as no cross-process variants are known. If two tenant workloads would be isolated by in-process isolation within the same VM, Spectre-STL could still be a viable attack vector. In the user-to-host model, Spectre-STL is mitigated


³In fact, Firecracker does not randomize the guest kernel address space anyway [76].

Table 6.5: Spectre PoCs run with AWS Firecracker recommended counter-measures (cf. fig. 6.1 and [13]). Experiments with Firecracker v1.0.0, v1.4.0, and v1.5.0 yielded identical results.

Variant	Platform	Configuration	Same process		Cross process	
			in-place	out-of-place	in-place	out-of-place
PHT	 ,  	any				
		any	N/A	N/A		
BTB		any				
						
						
			N/A	N/A		
			N/A	N/A		
		any				
RSB	 ,  	any	N/A	N/A		
STL	 , 	any		N/A	N/A	N/A

Experiments were run on the host system () , inside a single VM () , or across two VMs () . Victim and attacker share the same virtual and physical thread () , the same physical thread in separate virtual cores () , or two neighboring physical threads in separate virtual cores () .

 – mitigation prevents side-channel attack.

 – mitigation has no effect on this attack.

by countermeasures that are included in current Linux kernels and enabled by default.

For Spectre-PHT, the kernel countermeasures include the sanitization of user-pointers and the utilization of barriers (`lfence`) on privilege level switches. We therefore conclude that Spectre-PHT poses little to no threat to the host system. However, these mitigations do not protect two hyperthreads from each other if they execute on the same physical core in parallel. This is why all four Spectre-PHT mistraining variants are fully functional on the

host system as well as inside Firecracker VMs. As can be seen in table 6.5, VMs remain vulnerable *even if SMT is disabled*⁴. This makes Spectre-PHT a significant threat for user-to-user.

Spectre-BTB PoCs are partially functional when AWS recommended countermeasures are enabled. The original variant (1a) is fully functional while (1b) is successfully mitigated. Also, all attempts to leak information (2b) did not show any leakage. With (2a), however, we observed leakage. On the host system, the leakage occurred independent of SMT. Inside a VM, the leakage only occurred if all virtual CPU cores were assigned to a separate physical thread. Across VMs, disabling SMT removes the leakage.

Besides the countermeasures listed in fig. 6.1, the host kernel has Spectre countermeasures compiled into the VM entry and exit point⁵ to disable malicious guests from attacking the host kernel while the kernel handles a VM exit.

In summary, we can say that the Linux default countermeasures—which are recommended by the Firecracker developers—only partially mitigate Spectre. Precisely, we show:

- Spectre-PHT and Spectre-BTB can still leak information between tenants in the guest-to-guest scenario with the AWS recommended countermeasures—which includes disabling SMT—in place.
- The host kernel is likely sufficiently protected by the additional precau-

⁴Simulated by making attacker and victim share the same physical core (● and ■)

⁵<https://elixir.bootlin.com/linux/v5.10/source/arch/x86/kvm/vmx/vmenter.S#L191>

tions that are compiled into the Linux kernel to shield VM entries and exits. This, however, is orthogonal to security measures provided by Firecracker.

All leakage observed was independent of the Firecracker version in use.

Evaluation

We find that Firecracker does not add to the mitigations against Spectre but solely relies on general protection recommendations, which include mitigations provided by the host and guest kernels and optional microcode updates. Even worse, the recommended countermeasures insufficiently protect serverless applications from leaking information to other tenants. We therefore claim that Firecracker does *not* achieve its isolation goal on a microarchitectural level, even though microarchitectural attacks are considered in-scope of the Firecracker threat model.

The alert reader might wonder why Spectre-BTB remains an issue with the STIBP countermeasure in place (cf. fig. 6.1) as this microcode patch was designed to stop the branch prediction from using prediction information that originates from another thread. This also puzzled us for a while until recently Google published a security advisory⁶ that identifies a flaw in Linux 6.2 that kept disabling the STIBP mitigation when IBRS is enabled. We verified that the code section that was identified as being responsible for the issue is also present in the Linux 5.10 source code. Our assumption therefore is that the

⁶<https://github.com/google/security-research/security/advisories/GHSA-mj4w-6495-6crx>

same problem identified by Google also occurs on our system.

6.5 Impact

This work highlights the need for low-level defenses against microarchitectural attacks even when sophisticated virtualization and containerization isolation measures are in place. At the time of publishing, most of the vulnerabilities we evaluate in this paper are a few years old, and most have some countermeasures available or are even no longer present in the latest hardware; however, we consider wide selections of attack classes that have remained relevant for the better part of a decade since their initial discoveries. Meltdown and MDS-class attacks exploit a fundamental problem with the dependent speculative execution model that all modern CPU architectures employ for its powerful performance advantages. Buffer-clearing mitigations are only capable of plugging holes in this leaky ship one at a time, and there is no reason to believe that every hole has been found or that new designs won't introduce new holes. Similarly, Spectre attacks were introduced by sophisticated branch prediction techniques which are a source of significant performance gains in modern pipelined and out-of-order processors. In particular, Spectre-PHT remains an open attack vector that is not mitigated in newer CPUs by either changes in the prediction units or microcode updates; the OS, VMM, or other software must provide protections. Therefore, it is likely that Spectre will remain an issue for software developers in future CPU generations.

6.6 Related Works

To our knowledge, this work is the first that evaluates a class of attacks in and out of a particular virtual machine platform. Other works have investigated software vulnerabilities in microVMs. Xiao et al. showed that even with the minimal attack surface between a microVM and the host kernel, an attacker from within the VM can trigger host kernel functions and system calls to perform a wide range of attacks, including privilege escalation, performance degradation, and crashing the host [212].

A number of works have focused on efficiently integrating trusted execution environments into MicroVMs or serverless platforms, with implementation methods including Intel SGX [30], Trusted Platform Modules (TPMs) [158], and pure software enclaves [225]. While these environments can harden and verify high-priority code, they can still be vulnerable to microarchitectural attacks. In some cases, the additional microcode and hardware in SGX and TPMs intended to provide isolation even introduce new exploits that strengthen existing attacks [35, 139].

Chapter 7

DoubleHammer

Exploiting Adjacent Bit Faults

7.1 Introduction

In this work, we exploit fault injection and signature correction on ECDSA, targeting the nonce. We achieve this by overcoming two challenges:

Challenge 1: Adjacent Flips Rowhammer is a powerful software-only tool for fault injection however has many practical limitations. In particular, memory pages with flippable locations must be first identified and the victim baited into locating into such a page where the flip locates where a secret key or nonce would be allocated. The problem is flippable locations are far and few between and typically, we get isolated flips. By extensive Rowhammer experiments and bringing in the latest fault

acceleration technique we were able to, for the first time, locate pages with adjacent flippable locations.

Challenge 2: Synchronization The identified pages allow us to inject faults into adjacent bit locations making it possible to recover 2-bits of information from the nonces via the signature correction algorithm proposed by Mus et al. in Jolt. However, as in Jolt, we need many signatures generated by baiting the victim process to be allocated into the vulnerable pages. Synchronizing with the victim process over many signing sessions is a time consuming process. But unlike Jolt, we are targeting the nonce which is freshly generated for each signature. Hence, we can use the same vulnerable page over and over again. Indeed, we only need the victim to land once on the vulnerable page upon which we can request signing operations repeatedly (without losing more time for synchronization).

Challenge 3: Breaking the Lattice Barrier Once the faulty signatures are collected using Signature Correction we can recover 2 bits from the nonce of each signature. From here, using FFT to solve for the private key would be far too costly (requiring at least millions of signatures to be collected following by expensive post-processing). Instead, we employ the recently proposed lattice-based bounded distance decoding with predicate technique by Albrecht and Heninger [4] to recover the key much faster and with under 200 signatures.

7.1.1 Our Contributions

We introduce DoubleHammer a Signature Correction attack on signature scheme implementations. Specifically:

- We demonstrate the first real-world lattice attack by targeting on OpenSSL TLS key exchange recovering the full 196-bit and 256-bit ECDSA signing keys of a server. We only require co-location in a shared memory subsystem vulnerable to Rowhammer fault injection and a mechanism of carrying out a TLS handshake.
- The attack only requires co-location on system susceptible to Rowhammer fault injection as achieved in shared cloud instances or via Browser pages. Hence the proposed attack is a pure software-only attack, requiring no direct physical access.
- We demonstrate how an adversary can exploit Rowhammer vulnerabilities in shared memory systems to recover nonce leakages by post-processing faulty signatures.
- The earlier Jolt proposal [143], requires 1000's of signatures and many pages to achieve non-repeated coverage of secret key bits and expensive post-processing via Shank's algorithm. In contrast, our attack targets adjacent faulty locations in the nonce, allowing for reuse of the same vulnerable memory page. Using the same page we achieve 2-bit leakage recovery per faulty signature sample, enabling full key recovery with

only 200 faulty signatures and inexpensive post-processing.

- The nonce changing from one signing operation to the next is to our advantage since it allows us to use a single page over and over again. Compared to Jolt our attack only requires the victim process to land only once into the vulnerable page. We can collect all faulty signature required for the attack in one shot (by repeatedly asking for new signatures in the same session). Hence our attack is much faster than Jolt in the online signature collection phase.

7.2 Threat Model

We describe a brand-new fault injection attack against ECDSA in this section. The attack is effective in extracting secret key bits from faulty signatures, using only software-only Rowhammer attacks. The fault injection attack is composed of three stages. We start by locating susceptible memory regions where we can find adjacent two bit flips, known as pre-processing phase. Then in the online phase, we conduct a multi-sided Rowhammer attack on the victim and gather the faulty signatures. Finally, using the lattice attack algorithm, we post-process the faulty signatures and retrieve the secret key bits that were flipped. In the pre-processing phase, we assume the attacker knows what the victim program will be and can run that program on their own system, and that the attacker can run user-level code on the system where the attack will take place. In the online attack phase, we assume only

the one requirement of every Rowhammer attack: physical co-location in memory with the victim. We discuss known practical methods for achieving this in section 7.4.1.

7.3 Profiling

Before the online phase of the attack, we extensively profiled DRAM modules to identify locations of adjacent bit flips due to Rowhammer. We also profiled how OpenSSL allocates the nonce of an ECDSA signature. At a page level, we are concerned with how the dynamic variable of the nonce is allocated and reallocated throughout the lifetime of the OpenSSL server as it initializes itself and serves handshakes. Within the page, we are concerned with the offset of the nonce’s memory, as adjacent Rowhammer bit flips are relatively rare. To complete the attack, we will need a 4096-byte page where adjacent bit flips can occur *within* the 256-bit nonce. Furthermore, the attacker will need to be able to release such a page in such a way that it is likely that the victim OpenSSL server grabs that page while allocating memory and places the nonce within it.

7.3.1 OpenSSL Profiling

For this attack, we use the OpenSSL 1.0.1i SSL/TLS client and server provided in the OpenSSL executable for testing. While these features of the program are not intended to be used for production systems, they are the nearest

Virtual Address	Physical Address	Note
0x26394d0	0x4c8e24d0	Initialize server
0x2639c00	0x4c8e2c00	
0x2664d70	0x4c78bd70	
0x267b430	0x5933b430	
0x267ce30	0x4e1ffe30	2ND handshake
0x267ce30	0x4e1ffe30	3RD handshake
0x267ce30	0x4e1ffe30	4TH handshake
0x267ce30	0x4e1ffe30	5TH handshake

Table 7.1: Virtual addresses of the nonce throughout operation of the SSL/TLS server. Starting with the second handshake, the memory is reused for subsequent handshakes.

example of a reference implementation of SSL/TLS with the OpenSSL library that exists. We use link this executable with the OpenSSL FIPS 2.0.8 library. For profiling purposes, we made slight modifications to the OpenSSL library to log the virtual and physical address of the ECDSA nonce whenever it is allocated during the signature process. We then ran the SSL/TLS server using this library as a certificate server. We used an SSL/TLS client with the unmodified library as a client which initiates a standard SSL/TLS handshake with the server, thus triggering an ECDSA signature.

Virtual memory usage

By running the SSL/TLS server and client manually while monitoring the address log, a pattern was immediately apparent, illustrated with an example in table 7.1:

1. When the server is initialized, it performs three signatures, each allo-

cating the nonce at a different address in memory.

2. The first handshake triggers one signature, which allocates the nonce at another new address.
3. All subsequent handshakes trigger additional signatures with the nonce allocated at the *same address each time*.

The third property of memory allocation is extremely useful for this attack because once the victim has chosen the right memory location for Rowhammer fault injection, the attacker can reliably repeat the fault injection without relying on the randomness of memory allocation.

Page offsets

The nonce of a standard ECDSA signature system is 256 bits, but a standard memory page on x86 (32- and 64-bit versions) is 4096 bits. Therefore, aligning the offset of the nonce with the site of an adjacent pair of Rowhammer errors will be essential for producing an effective attack. We restarted the OpenSSL server 10,000 times, each time executing 5 handshakes after the server was started (as modeled in table 7.1). In 9,999 iterations, the second through fourth addresses allocated matched; in just one iteration, there was a communication error between the client and server and the handshakes were not completed.

Figure 7.1 plots a histogram of the memory offset within the page of the nonce; that is, the 12 least-significant bits of the virtual address which also

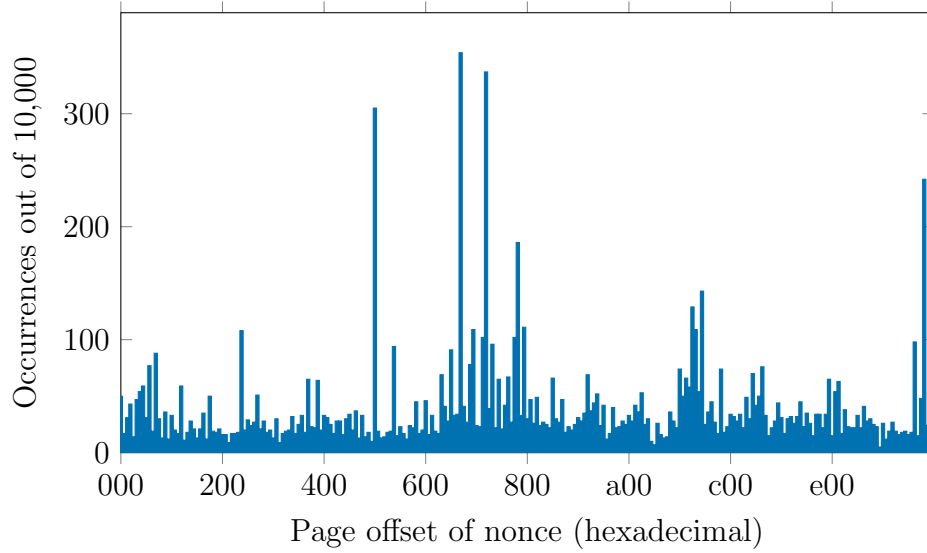


Figure 7.1: Distribution of 12-bit memory page offsets of the nonce k in OpenSSL on DDR4 system.

match the 12 least-significant bits of the physical address.

The nonce is always aligned to 16 bytes, so there are actually only $4096 \div 16 = 256$ possible offsets within a page. The most probable offsets are concentrated near 0xd00, with smaller concentrations at the start and end of a page. Notice, also, that the nonce itself is 256 bits long —graphically, the width of 16 bars of fig. 7.1, each of which represents 16 bytes of the page. Therefore, the chance of a random pair of faulty bits being in a page offset that corresponds to a possible page offset of the nonce is not as low as the visual sparsity of fig. 7.1 might suggest.

Physical page allocation

Last but not least, the attacker needs to understand the pattern of physical page allocation by the SSL/TLS server. After the attacker has found a physical page that is a candidate for side-by-side Rowhammer fault injection at an offset where the server might allocate the nonce, the attacker still needs to *release* that page in such a way that the server actually uses it for storing the nonce. Linux prefers to reuse recently deallocated pages, but the page usage of the victim program is not trivial to determine, nor is it deterministic. We devised an experiment as follows:

1. Allocate a pool of pages, and note the physical address of each in order.
2. Start the server and complete a single handshake.
3. Deallocate all pages in the same order they were recorded
4. Complete a second handshake and note the physical address that the server uses for the nonce during this handshake.
5. Compare the page that the nonce resides in to the list of pages that were deallocated to determine the order of reallocation.

In the attack, one of the deallocated pages will be the target page, which the attacker hopes that the victim will allocate and use for the nonce. Therefore, during the profiling stage, we are merely looking for the most likely page within a given pool to be allocated and used in this way. We found that

Pages in pool	Most frequently grabbed page	Frequency
1000	989	2.3%
100	100	2.3%
75	75	2.5
50	50	4%
45	45	2.8%
40	40	4.2%
37	37	2.4%
20	20	1.7%
10	10	2.9%
1	1	2.1%

Table 7.2: Results of the experiment described in section 7.3.1. Page numbers are given in the order of deallocation; that is, page 100 out of 100 pages is the page that is deallocated last. For small numbers of pages, the last page to be deallocated is most likely to be grabbed by the victim program.

differently sized pools perform in different ways, so we searched for a pool size with best odds of success for a single page, and decide. Table 7.2 shows a sampling of results from this test. We selected 40 pages as a practical and relatively reliable pool size.

Attack window timing

The window of time for a fault injection in the nonce of an ECDSA key during signing is relatively small. If the fault is injected too early or too late, the same nonce is used for both r and s , so the signature is valid and leaks no information about the secret key. We devised a simple experiment to find the timing of the window relative to the initiation of a handshake (the timing of which will be controlled by the attacker).

First we modified the OpenSSL library to include a variable indicating the state of the program in relation to the window (before, during or after). Then we added a signal handler to the OpenSSL server executable that logs the value of that variable when a particular signal is received. The actual test proceeds as follows: First, an SSL/TLS server is started. A Bash script initiates a handshake with the server in the background and then immediately sleeps for a predetermined time. When the sleep is finished, the script signals the server, which records its execution state.

We searched for the best sleep time by first trying sleep times of 0.0s, 0.1s, 0.2s, ..., 0.9s. We found that 0.0 seconds was too quick (the program reported that the window had not been reached) but 0.1 seconds was too long (the program reported being past the window already), so we proceeded to try 0.00s, 0.01s, ..., 0.9s. By following this process we eventually found that the window could be hit after sleeping for 0.00436 to 0.00438 seconds, but hitting the window was not reliable. We tested sleep times between 0.00430 and 0.00440 1000 times each, and no sleep time achieved a hit rate of more than 0.3%. We concluded that to make this attack practical, we would need to slow down the execution of the OpenSSL server, and decided to do this with a microarchitectural denial of service attack.

When running an self-modifying code (SMC)-based denial of service attack in a hyperthread adjacent to the server, we found that the window moved to around 0.0078, but more importantly, it widened significantly to the point where it became practical to target the window with just Bash's sleep

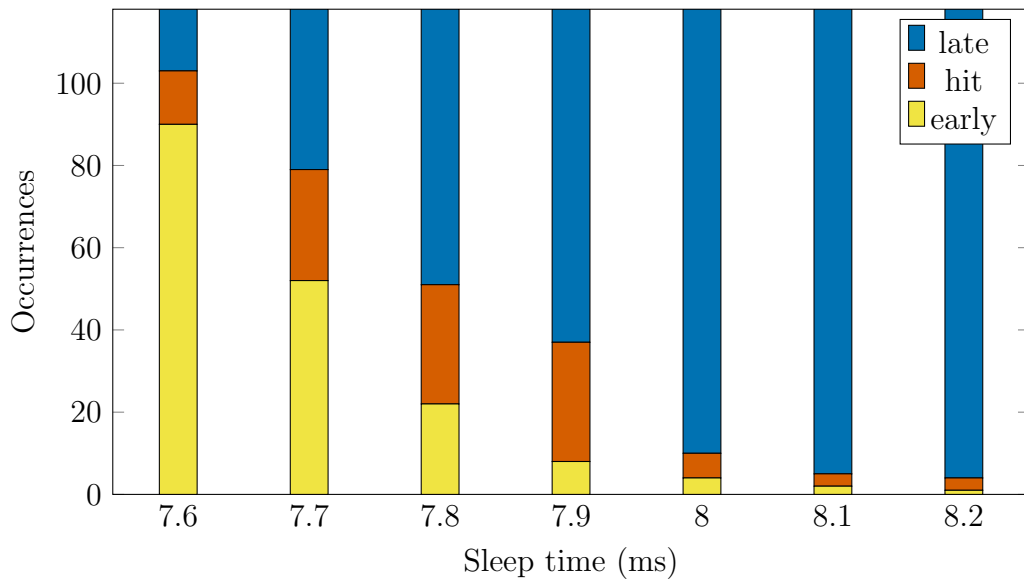


Figure 7.2: Attack window hit rates with for various sleep times between launching the victim and starting the attack, with SMC denial of service running. The bars represent the portion of early misses, late misses, and hits for each tested time. Sleep times of 7.7-7.9 ms achieve 22-25% hit rates.

command. As shown in fig. 7.2, sleep times of 0.0077, 0.0078, and 0.0079 all achieve window hit rates of 22-25%. SMC widened the attack window by about 10 times, from around 30 μ s to 300 μ s, and in doing so increased the timing accuracy of the attacker nearly 100-fold, from just 0.3% to as much as 25%.

7.3.2 DRAM Profiling

DRAM profiling determines the susceptibility of system hardware to our adjacent bit flip Rowhammer attack. The first step of DRAM profiling is the pre-processing phase, which includes templating on the same system as the victim. In this phase it is not necessary for the victim to be present or active. Rather, the information gathered during this phase will be used to inform the setup of the attack phase.

Memory Preparation

Contiguous Memory Detection Rowhammer attacks rely on the physical proximity of the attacker and victim rows in the DRAM chip. While we cannot measure this directly without probing the DRAM chip itself, proximity in physical address space tends to be a reliable alternative. Therefore, to construct the most effective hammering patterns, we first search a 512 MB buffer for continuous physical memory. To do this without simply sorting pages by physical address, which would be quick and error-free but requires root access (or another exploit), we use the SPOILER side-channel [96].

Bank Co-location For a successful rowhammer attack the attacker and the victim should be in the same bank. After finding continuous memory, we employ a row-conflict side-channel to identify the virtual addresses mapping to the same bank. It takes longer to access two addresses from the same bank than it does to access two addresses from separate banks. This is due to the fact that before loading another row, a loaded row in the row buffer must be written back to its original location. Another option, if physical addresses are known, is to do preliminary row layout profiling with this channel as described in [162] and reverse engineer the mappings from physical addresses to DRAM hardware layout, including channels, ranks, banks, and rows. This allows for deterministic sorting of rows by bank by simply decoding the physical address.

7.3.3 Hammering Techniques

Experiment Setup For our experiments we used an Intel Core i9-9900K CPU with a Coffee Lake microarchitecture. We chose the DDR4 DRAM chip with part number CMU64GX4M4C3200C16 and 16GB capacity. DRAM row refresh time was maintained at 64 ms, which is the standard setting in most systems. The operating system was Ubuntu 20.04.01 LTS with the 5.15.0-58-generic Linux kernel.

Method for Finding Adjacent Bit Flips An important component of this attack is finding adjacent bit flips. These are rows that are determined

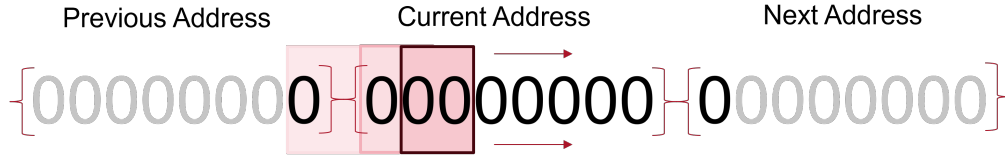


Figure 7.3: Using a sliding window approach to finding adjacent bit flips.

to have at least two faulty cells adjacent to each other. To detect these pages with adjacent bit flips we use a sliding window approach. After the page is hammered during the profiling stage, each 8-bit address is searched for a double-bit flip, as seen in fig. 7.3. When searching for bit flips 0 to 1, we iterate through each 8-bit address in the page, and compare the Least Significant Bit (LSB) with the Most Significant Bit (MSB) of the current address, and to determine if they form an adjacent bit flip pair. Then we compare each adjacent pair in the 8-bit address for a double flip, then finally we compare the LSB of the current address with the MSB of the next address.

Multi-sided Hammering For systems using DDR4 memory with TRR protection, a double-sided Rowhammer attack is generally insufficient to achieve bit flips. Instead, we use a multisided attack as first introduced by [51] to achieve flips in DDR4. We select a continuous block of rows within a single bank and use every other row as an attacker. The remaining rows between the attacker rows are the victim rows. Accessing a large number of attacker rows overwhelms DDR4’s TRR system so that it is unable to detect the attack and defensively refresh victim rows. Experimentally, we determined that at least 9 or 10 sided hammering is needed to observe

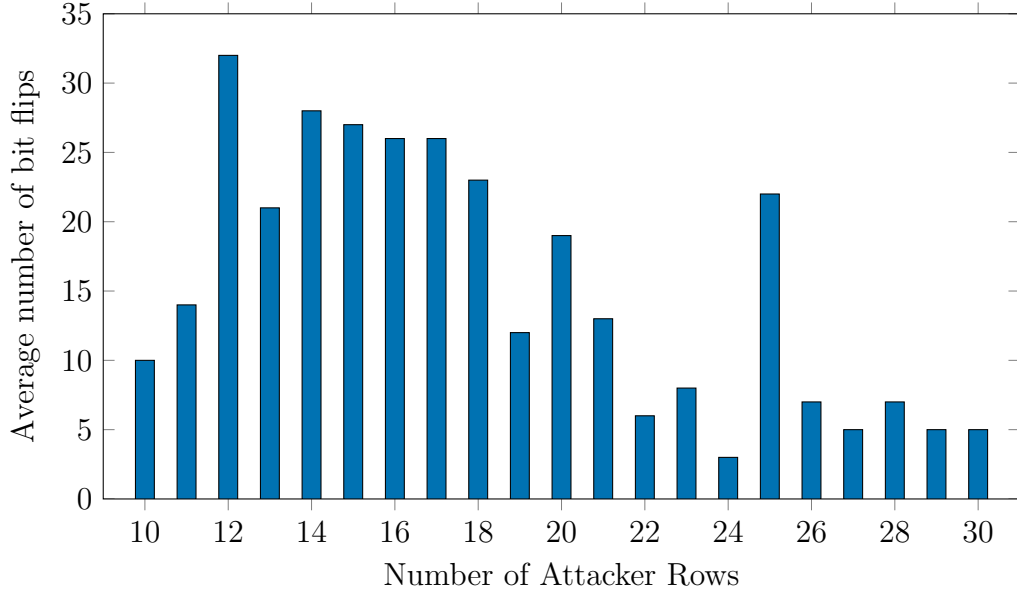


Figure 7.4: Average number of bit flips on an 128MB buffer vs the number of sides in a multi-sided Rowhammer attack.

adjacent two bit flips in the DDR4 system. Figure 7.4 shows the total number of adjacent bit flips found after hammering each set of attacker and victim rows in a 128MB buffer.

Data Pattern Dependence

Since the Rowhammer vulnerability is based on the electrical phenomena underlying the storage of zeros and ones in DRAM, it is not safe to assume that Rowhammer faults are data-independent. Therefore, we tested a number of data patterns in the aggressor and victim rows and counted the frequency of adjacent bit flips. We tried filling all victim rows with ones (i.e., looking for

#Aggressor Rows	Flip Direction	#Adj. bit flips
12-sided	1→0	320
12-sided	0→1	114
13-sided	1→0	312
13-sided	0→1	177
14-sided	1→0	191
14-sided	0→1	80
15-sided	1→0	269
15-sided	0→1	173

Table 7.3: An experiment was conducted with diverse multi-sided access patterns to illustrate how the data pattern influences the occurrence of double bit flips.

one to zero flips) and all attacker rows with zeros and vice versa. In table 7.3 we compare the results. While running the experiments we observed a relation between data pattern and generation of bit flips. With each n-sided setting, we saw more 1 to 0 flips—in some cases more than twice as many.

7.4 Collection

In this section we demonstrate how an attacker who has profiled a system can collect faulty signatures where the nonce has had two adjacent bits flipped. We walk through the steps of physical co-location, allocation of the target page, and finally verification and collection of signatures. At each step, we explain our own test setup as well as how the attack would be modified to work in a realistic scenario.

7.4.1 Physical Co-location

In our tests, we are not using a distributed cloud environment, but a single computer. Physical co-location is not a problem, since any two programs running on this computer will use the same RAM. In an actual cloud environment, co-location is possible with cache side-channels and other techniques [80]. Alternatively, Rowhammer attacks can be carried out by malicious scripts running inside a web browser [64].

7.4.2 Allocation of a Target Page

As explained in section 7.3, the nonce must be stored in a physical page that meets very particular requirements. As we describe in section 7.3.2, there are quite few physical pages that incur adjacent bitflips with any regularity, and furthermore, they must occur within the right page offset. For our attack, since we performed profiling separately, we allocated a known target page by brute force; that is, allocating a large number of pages and then searching their physical addresses one at a time. We used root privileges to check physical addresses most efficiently, but they could also be checked by a single-thread exploit of the RIDL side-channel [201]. Alternatively, an attacker could construct a single program that profiles the memory for a usable page and then immediately uses it for the attack, without handling physical addresses at all.

7.5 Analysis

7.5.1 Signature Correction on Adjacent Nonce Faults

In our attack we make use of the signature correction algorithm as applied to ECDSA to recover bits from the nonces. We follow the fault injection attack given by Jolt [143] but instead of injecting faults into the decryption key d we target the Nonce k . The ECDSA signing and verification algorithms for the case of a faulty Nonce k are given below. All faulty parameters are marked by a bar. Note that the faults are injected after the nonce is used to compute $R = kP$ and before $\bar{s} = (\bar{k})^{-1}(H(M) + dr) \bmod n$ is computed. Hence, R and r is correctly computed and returned allowing signature correction.

```

1 Function sign( $M, d, P$ )
   input :  $M$  – message to be signed;  $d$  – private key;  $P$  – elliptic
           curve generator
   output : signature pair  $(r, s)$ 
2    $k \leftarrow \text{random} \in Z_n^*$ 
3    $R \leftarrow kP$  //  $R$  is an elliptic curve point generated with
        $k$ 
4    $r \leftarrow (kP)_x$  //  $r$  is the x coordinate of  $R$ 
5    $\bar{k} = k + \Delta_k$  // inject a fault into  $k$ 
6    $\bar{s} = (\bar{k})^{-1}(H(M) + dr) \bmod n$  // faulty  $s$  generated with  $\bar{k}$ 
7   return  $(r, \bar{s})$  // signature has correct  $r$ , faulty  $\bar{s}$ 

```

Algorithm 5: ECDSA Signing with a faulty nonce

Clearly the faulty nonce $\bar{k} = k + \Delta_k$ corrupts the half signature \bar{s} which then fails to verify $\bar{r} \neq r$ during Signature Verification. Note that the Verification function can be run by anybody by knowledge of the public

key and faulty signature. Hence all parameters shown including the faulty intermediate values are available to the attacker.

For Signature Correction we need to understand the impact of the faults on the parameters computed inside the verification function. As shown in Jolt (Appendix, Claim 2) [143], the following holds true for the faulty \bar{R} value that is computed during Signature Verification

$$\bar{R} = R + \Delta_k P$$

Note that both R and the generator P are public. The value \bar{R} is computed via knowledge of the faulty signature. Hence, the only unknown is Δ_k . As long as the faults are isolated to a few flips, Δ_k can be recovered by exhaustive search by checking if the equation is verified. For instance, for 256-bit nonces for single bit flips, we need to try $2 \times \binom{256}{1}$ (times two for both flip directions), $4 \times \binom{256}{2}$ values, etc. In any case, we only get very few isolated errors via Rowhammer anyway.

Decoding Δ_k to Nonce Bit Values Once the error pattern Δ_k is recovered, it needs to be decoded to the actual corresponding bit value in the nonce. This is best illustrated by a simple example: Assume we recover the additive fault value as $\Delta_k = -32$ in a setup where we can only inject single bit flips. This means the nonce $\bar{k} = k + \Delta_k = k - 32 = k - 2^5$. Or in other words the 6-th bit of k must have been flipped from a logic 1 to a 0. Hence the 6-th bit of k must have been a 1.

Impact of Adjacency to Signature Correction For lattice based key recovery to succeed, we require adjacent flips (and adjacent bits) from nonces. In this case the decoding is a bit more complicated. There are four possible error patterns

$$\begin{aligned} (00 \mapsto 11) : \Delta_k &= +32^i & (11 \mapsto 00) : \Delta_k &= -32^i \\ (01 \mapsto 10) : \Delta_k &= +2^i & (10 \mapsto 01) : \Delta_k &= -2^i \end{aligned}$$

Here i denotes the position where the adjacent faults start impacting the nonce. Note that during decoding our starting point is the Δ_k . The two error patterns $\Delta_k = \pm 2^i$ are somewhat problematic since they might have also been caused by single bit faults. Since Rowhammer is imperfect, we might indeed inadvertently inject single faults which might mistakenly be decoded as a double bit fault. The lattice reduction based key recovery step is intolerant to noise. Hence in our attack we discard such samples and only use samples with $\Delta_k = \pm 32^i$. If $\Delta_k = +32^i$ then we deduce nonce bits to be 00 starting at bit position i , and otherwise if $\Delta_k = -32^i$ then the nonce bits in the same position are decoded as 11.

Postprocessing Complexity Once the faulty signatures are collected, error patterns computed using signature correction, and the two nonce bits recovered, we formulate the information recovered into a HNP instance as explained in section 2.11. The instance is then processed using the BDD-

$ n $	$ k $	m	bkz-enum	bkz-sieve	enum-pred	sieve-pred
192	190	110	$2^{59.8}$	$2^{61.8}$	$2^{59.8}$	$2^{49.7}$
192	190	98	n/a	n/a	$2^{53.9}$	$2^{45.2}$
256	254	146	$2^{80.8}$	$2^{76.3}$	$2^{80.8}$	$2^{63.2}$
256	254	130	n/a	n/a	$2^{73.7}$	$2^{57.2}$

Table 7.4: Comparison of lattice reduction cost estimates for 2-bit nonce leakage without and with predicate optimization proposed in [4] for ECDSA key size $|n|$ -bits, with $|k|$ unknown nonce bits, with m collected signature samples.

predicate technique [4]. In table 7.4 we have listed complexity estimates obtained using the Git package `bdd-predicate`¹. We report the number samples m for optimal 192-bit and 256-bit cases, i.e. **$2^{45.2}$** and **$2^{57.2}$** time complexities, respectively, in rows 2 and 4. Rows 1 and 3, the optimal setting where enumeration and sieving still report results for comparison with the predicate versions provided in the last columns. In addition to these results, the estimator reports about 5.77 hours and 60K CPU hours, for the 192-bit and 256-bit sizes, respectively. For the 256-bit key size this may seem excessive but its attainable since the algorithm is easily parallelized².

7.6 Countermeasures

There are some proposed hardware and software level solutions against Rowhammer attack. Unfortunately, newer attacks or technological advance-

¹<https://github.com/malb/bdd-predicate>

²The `bdd-predicate` software library supports scaling via more threads.

ments keep breaking these remedies.

7.6.1 Hardware Countermeasures

As of yet, rowhammer attacks on legacy systems have no immediate defense. In this section, we propose some hardware-only defense against rowhammer attacks that is both practical and effective.

Error Correcting Code (ECC)

Use of ECC, which is widely available in many chipsets, was initially advised as a defense against Rowhammer [109]. In general ECC works by including redundancy that can correct single bit flips and double bit flips. [109] proved that by identifying and flipping three or more bit locations within a word utilizing timing side-channels, it is possible to overcome ECC. In particular, it is anticipated that the ECC algorithm will fix or reveal any bits that they manage to flip in memory under actual conditions. In [42] the authors have practically shown that, ECC-equipped memory is still vulnerable to Rowhammer attack. Using ECCploit an attacker can reverse engineer the ECC function and trigger Rowhammer bit flips bypassing ECC memory without crashing the system.

Targeted Row-Refresh (TRR)

The TRR feature, which refreshes the neighbors of a select few frequently accessed rows, has been added by DRAM makers to DDR4 memory modules

and some memory controllers. It's a fact that the details of TRR working are not publicly available [50] and implementations vary chip wise. [51] showed that it is possible to cause Rowhammer attack even with TRR in the modern DDR4 system. According to Halddouble [115], by hammering distance-2 aggressor row it is possible to exploit the TRR refreshes which is caused in distance-1 aggressor rows and increases the probability of getting bit flips. Most recently, Jattke et al. [100] successfully used TRR to flip bits in all 40 of their newly acquired DDR4 DIMMs.

Alternatives to TRR

In [176] Saileshwar et al. suggested substituting an attacker-oriented defense for victim-oriented defenses like TRR. After a predetermined number of activations, they suggest utilizing a permutation layer to switch attacker rows with another row from the same bank. Using this mechanism it is possible to break the locality between aggressor and victim rows which consequently decreases the probability of hammering the same victim row repeatedly.

Cryptographic Security and Integrity against Rowhammer

[104] suggested that CSI:Rowhammer, a hardware-software hybrid that uses a cryptographic MAC for hardware error detection, completely counteract Rowhammer assaults. In 256 bits of data, CSI:Rowhammer can identify any number of bit flips and fix up to 8 bit flips in a reasonable amount of time. when it comes to performance measure CSI:Rowhammer outperforms

ECC and TRR protected DRAMs against rowhammer attack. With the capacity to rectify a single bitflip in less than 20 ns during the rowhammer attack, CSI:Rowhammer maintains a low latency, as opposed to standard ECC-DRAM that can take up to 63 μ s.

7.6.2 Software Countermeasures

[143] proposes careful fault checking especially to prevent leakage of faulty signatures. Indeed several vendors have already implemented fault checking, i.e. wolfSSL, LibreSSL while others such as OpenSSL did not, citing that fault injection attacks are outside their threat model.

Verify after Sign

A sender can identify the presence of an adversary inserting flaws using Rowhammer by using the verify after sign technique. The verification algorithm has the benefit of being around three times quicker than the double signing process. In addition, adding a verification step to libraries is simple. The disadvantage, however, is that employing instruction skips through opcode flipping [63], which skips the checking phase, increases the risk of the checking system itself falling prey to an attack.

Redundant Signing

Another way to check for injected faults is to sign many times with the identical values where each copy is stored in a separate region of memory

and then compare the results. To be successful, the attacker would need to repeatedly inject the same fault into the same places during all signing operations. The two downsides of the countermeasure are overhead and another one is the instruction skip attack which might target the check step. It's possible that other attacks, like RAMBleed [120], which target a single computation, will still be able to get past this basic defense.

Masking Sensitive Values

Masking [173] is a randomization technique which is used against strong side channel attacks. Often, the input to a cryptographic algorithm is concealed by employing random values that are hidden from an opponent. Because of this, the intermediate outcomes of the algorithm calculation are unrelated to the input, and the adversary is unable to use the side-channel to gather any meaningful information. As masking is randomized higher order masking can also be used to increase the scheme's resilience. As long as the initial mask randomization procedure is safe from fault injection, masking will offer safety. But the system will become vulnerable to the attack if the fault is injected before the initial step.

Chapter 8

Conclusion

In chapter 4, we show that modern FPGA-CPU hybrid systems can be more vulnerable to well-known hardware attacks that are traditionally seen on CPU-only systems. We show that the shared cache systems of the Arria 10 GX and its host CPU present possible CPU to FPGA, FPGA to CPU, and FPGA to FPGA attack vectors. For Rowhammer, we show that the Arria 10 GX is capable of causing more DRAM faults in less time than modern CPUs. Our research indicates that defense against hardware side-channels is just as essential for modern FPGA systems as it is for modern CPUs. Of course, the security of any device physically installed in a system, like a network card or graphics card, is important, but FPGAs present additional security challenges due to their inherently flexible nature. From a security perspective, a user-configurable FPGA on a cloud system needs to be treated with at least as much care and caution as a user-controlled CPU thread, as it can exploit

many of the same vulnerabilities.

In the IOTLB-SC chapter, we demonstrated a new side-channel attack against IOTLBs in such IOMMUs that works across virtual environments and threatens cloud tenants. We developed a new eviction set finding algorithm that works without prior assumptions of cache or TLB organization and a hardware module for an FPGA that implements the fundamentals necessary to exploit the IOTLB side-channel. We used these tools to record a side-channel trace from a GPU running a database acceleration library. The results prove that the IOTLB can be used as a side-channel to spy on co-located devices. We highlight this fact by showing a very reliable covert channel from the GPU to the FPGA where we use the database application running on the GPU to encode messages into the GPU's system memory access patterns. While we acknowledge the limitations of the IOTLB channel with current hardware and applications, we argue that with the upcoming PCIe 5.0 and CXL standards, IOMMU usage patterns will change and fine-grained IOTLB side-channel attacks will become practical. To overcome the threat of the side-channel, we suggest a variety of countermeasures that can be implemented on different system levels ranging from hardware modifications up to the implementation of applications. Many of these countermeasures fully eliminate the threat of IOTLB side-channels, but at the same time reduce the speed of peripherals or scalability of the systems that host them. Therefore, when designing or choosing hardware for large-scale, high-performance, secure services, IOTLB threats must be acknowledged and IOTLB isolation measures must be carefully

considered for the specific needs of the system. Furthermore, when designing security-critical peripherals or security-critical software or firmware that makes use of peripherals, timing leakages from peripheral memory accesses must be addressed with constant-time design practices.

We showed that the previously recommended countermeasures for the Firecracker VMM were incomplete and insufficient to meet its isolation goals and, with our disclosure to AWS, prompted a revision to the recommendations. Furthermore, many of the tested attack vectors showed leakage while countermeasures were in place. We identified the Medusa cache indexing/block write variant as an attack vector that only works across VMs, i.e. with additional isolation mechanisms in place. Additionally, we showed that disabling SMT—an expensive mitigation technique recommended and performed by AWS—does not fully protect against Medusa variants. The aforementioned Medusa variant, and Spectre-PHT are still capable of leaking information between VMs when SMT is disabled, if the attacker and target threads keep competing for hardware resources of the same physical CPU core. Unfortunately this is inevitably the case in high-density serverless environments. Furthermore, processor designs continue to evolve and speculative and out-of-order execution remain important factors in improving performance from generation to generation. So, it is unlikely that we have seen the last of new microarchitectural vulnerabilities, as the recent wave of newly discovered attacks [137, 196, 209] shows.

Finally, we demonstrated that adjacent bit flips—enabling incredibly

powerful lattice attacks—are possible with Rowhammer and addressed all of the practical challenges to carrying out a Rowhammer attack in a real world setting with minimal assumptions in the threat model. We explain the latest and most powerful techniques for finding continuous physical memory and sorting pages by bank. We use a CPU denial of service attack to slow the victim process and show that this widens the attack window enough for Bash’s sleep command to provide timing control for the attack. We discuss the many choices of hammering patterns that evade DDR4’s built-in defenses and find good parameters for achieving adjacent bit flips on our test system.

In combination, these contributions make a multi-faceted but surely incomplete picture of the microarchitectural threat landscape for cloud and heterogeneous computing systems. We analyze several specific vulnerabilities, including ones novel to these systems, and focus on how the specific hardware needs of cloud and heterogeneous paradigms present new threats and challenges. Both computing models require the integration at the hardware level of seemingly disparate computational processes: in the case of cloud computing, multiple users’ programs are executed on shared hardware; in the case of heterogeneous computing, radically different hardware components share memory and other computational resources. Wherever these disparate processes meet, there is potential for microarchitectural leakage or interference. The scale and complexity of modern cloud and heterogeneous systems makes it extremely difficult to identify every attack surface, and every new microarchitectural feature that introduces increased performance (or even

increased security) to these systems is also a potential source of a new threat. To look at these systems more optimistically, their depth and complexity also provides opportunities for microarchitectural defenses at every level, from hardware and microcode to kernel and user code. This work provides new perspectives which we hope will help others to identify new vulnerabilities, reassess old ones, and implement effective defenses in heterogeneous and cloud systems.

Bibliography

- [1] Leonard M Adleman and Jonathan DeMarrais. A subexponential algorithm for discrete logarithms over all finite fields. *Mathematics of Computation*, 61(203):1–15, 1993.
- [2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *NSDI*, pages 419–434. USENIX Association, 2020.
- [3] Martin R Albrecht, Amit Deo, and Kenneth G Paterson. Cold boot attacks on ring and module lwe keys under the ntt. *Cryptology ePrint Archive*, 2018.
- [4] Martin R. Albrecht and Nadia Heninger. On bounded distance decoding with predicate: Breaking the “lattice barrier” for the hidden number problem. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021*, pages 528–558, Cham, 2021. Springer International Publishing.
- [5] Martin R Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.
- [6] Alibaba Cloud. FPGA-based compute-optimized instance families, 2019. Access: 2019-10-15.
- [7] Alibaba Cloud ECS. Introducing the sixth generation of Alibaba Cloud’s elastic compute service, 2020. Access: 2022-01-31.
- [8] Amazon AWS. Amazon EC2 F1 instances, 2017. Access: 2019-10-12.

- [9] Amazon AWS. AWS Nitro System, 2018. Access: 2022-01-31.
- [10] Amazon Web Services. AWS Lambda features, 2023. accessed: Aug 17, 2023.
- [11] Amazon Web Services. Firecracker design, 2023.
- [12] Amazon Web Services. The Firecracker jailer, 2023. accessed: August 14, 2023.
- [13] Amazon Web Services. Production host setup recommendations, 2023. accessed: May 22, 2023.
- [14] Amazon Web Services. Production host setup recommendations (updated), 2024. accessed: Mar 29, 2024.
- [15] AMD. *AMD I/O Virtualization Technology (IOMMU) Specification*, 3.06-pub edition, 2021.
- [16] AMD. Offering unmatched performance, leadership energy efficiency and next-generation architecture, AMD brings 4th gen AMD EPYC processors to the modern data center, 2022. Access: 2022-12-04.
- [17] X9 ANSI. 62: public key cryptography for the financial services industry: the elliptic curve digital signature algorithm (ecdsa). *Am. Nat'l Standards Inst*, 1999.
- [18] Diego F. Aranha, Pierre-Alain Fouque, Benoît Gérard, Jean-Gabriel Kammerer, Mehdi Tibouchi, and Jean-Christophe Zapalowicz. GLV/GLS decomposition, power analysis, and attacks on ECDSA signatures with single-bit nonce bias. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*, volume 8873 of *Lecture Notes in Computer Science*, pages 262–281. Springer, 2014.
- [19] Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. Ladderleak: Breaking ecdsa with less than one bit of nonce leakage. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, page 225–242, New York, NY, USA, 2020. Association for Computing Machinery.

- [20] Christian Aumüller, Peter Bier, Wieland Fischer, Peter Hofreiter, and Jean-Pierre Seifert. Fault attacks on RSA with CRT: Concrete results and practical countermeasures. In Burton S. Kaliski, Çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 260–275, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
- [21] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamaric, and Leonid Ryzhyk. System programming in Rust: Beyond safety. In *HotOS*, pages 156–161. ACM, 2017.
- [22] Ramachandran Balasubramanian and Neal Koblitz. The improbability that an elliptic curve has subexponential discrete log problem under the menezes—okamoto—vanstone algorithm. *Journal of cryptology*, 11:141–145, 1998.
- [23] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. “Ooh Aah... Just a Little Bit” : A Small Amount of Side Channel Can Go a Long Way. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, pages 75–92, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
- [24] Daniel J Bernstein. *Introduction to post-quantum cryptography*. Springer, 2009.
- [25] Sarani Bhattacharya and Debdeep Mukhopadhyay. Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES 2016*, pages 602–624, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.
- [26] Sarani Bhattacharya and Debdeep Mukhopadhyay. Advanced Fault Attacks in Software: Exploiting the Rowhammer Bug. In Sikhar Patranabis and Debdeep Mukhopadhyay, editors, *Fault Tolerant Architectures for Cryptography and Hardware Security*, pages 111–135. Springer Singapore, Singapore, 2018.
- [27] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In Walter

- Fumy, editor, *Advances in Cryptology — EUROCRYPT '97*, pages 37–51, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.
- [28] Dan Boneh and Ramarathnam Venkatesan. Hardness of computing the most significant bits of secret keys in diffie-hellman and related schemes. In *Advances in Cryptology—CRYPTO'96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings*, pages 129–142. Springer, 2001.
 - [29] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. \mathcal{A} EPIC leak: Architecturally leaking uninitialized data from the microarchitecture. In *USENIX Security Symposium*, pages 3917–3934. USENIX Association, 2022.
 - [30] Stefan Brenner and Rüdiger Kapitza. Trust more, serverless. In *SYS-TOR*, pages 33–43. ACM, 2019.
 - [31] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. CacheShield: Detecting Cache Attacks through Self-Observation. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, CODASPY '18*, pages 224–235, New York, NY, USA, 2018. Association for Computing Machinery.
 - [32] Billy Bob Brumley and Risto M. Hakala. Cache-Timing Template Attacks. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, pages 667–684, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
 - [33] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *Computer Security—ESORICS 2011: 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12–14, 2011. Proceedings 16*, pages 355–371. Springer, 2011.
 - [34] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: hijacking transient execution through microarchitectural load value injection. In *IEEE Symposium on Security and Privacy*, pages 54–72. IEEE, 2020.

- [35] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *SysTEX@SOSP*, pages 4:1–4:6. ACM, 2017.
- [36] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtvushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security Symposium*, pages 249–266. USENIX Association, 2019.
- [37] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant CPUs. In *CCS*, pages 769–784. ACM, 2019.
- [38] Claudio Canella, Jo Van Bulck, Michael Schwarz, Daniel Gruss, Catherine Easdon, and Saagar Jha. Transient fail [code], 2019.
- [39] Sébastien Carré, Matthieu Desjardins, Adrien Facon, and Sylvain Guilley. OpenSSL Bellcore’s Protection Helps Fault Attack. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 500–507, August 2018.
- [40] Marco Chiappetta, Erkan Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 49:1162 – 1174, 2016.
- [41] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, 1998.
- [42] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting correcting codes: On the effectiveness of ECC memory against rowhammer attacks. In *IEEE S&P*, pages 55–71. IEEE, 2019.
- [43] Dana Dachman-Soled, Léo Ducas, Huijing Gong, and Mélissa Rossi. Lwe with side information: attacks and concrete security estimation. In *Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part II*, pages 329–358. Springer, 2020.

- [44] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. Smash: Synchronized many-sided rowhammer attacks from javascript. In *USENIX Security Symposium*, pages 1001–1018, 2021.
- [45] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean M. Tullsen. Prime+Abort: A timer-free high-precision L3 cache attack using intel TSX. In *USENIX Security Symposium*, pages 51–67. USENIX Association, 2017.
- [46] Marie Doleželová, Milan Navrátil, Eva Majoršínová, Peter Ondrejka, Douglas Silas, Martin Prpič, and Rüdiger Landmann. *Red Hat Enterprise Linux 7 Resource Management Guide—Using cgroups to manage system resources on RHEL*. Red Hat, Inc, Dec 2020. accessed: Aug 17, 2023.
- [47] T El Gamal. A public key cryptosystem abd a signature scheme based on discrete logarythms. In *Proceedings of “Advances in Cryptology—CRYPTO*, volume 84, 1985.
- [48] Daniel Firestone, Andrew Putnam, Sambrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian M. Caulfield, Eric S. Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert G. Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *NSDI*, pages 51–66. USENIX Association, 2018.
- [49] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating microarchitectural attacks with the GPU. In *IEEE S&P*, pages 195–210. IEEE, 2018.
- [50] Pietro Frigo, Emanuele Vannacc, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Trtrespass: Exploiting the many sides of target row refresh. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 747–762, 2020.

- [51] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the many sides of target row refresh. In *IEEE S&P*, pages 747–762. IEEE, 2020.
- [52] Jacob Fustos, Michael Garrett Bechtel, and Heechul Yun. SpectreRewind: Leaking secrets to past instructions. In *ASHES@CCS*, pages 117–126. ACM, 2020.
- [53] Nicolas Gama and Phong Q Nguyen. Finding short lattice vectors within mordell’s inequality. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 207–216, 2008.
- [54] Romain Gay, Dennis Hofheinz, Eike Kiltz, and Hoeteck Wee. Tightly cca-secure encryption without pairings. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8–12, 2016, Proceedings, Part I*, pages 1–27. Springer, 2016.
- [55] Silvia E. Gianelli. Xilinx announces general availability of Virtex Ultra-Scale+ FPGAs in Amazon EC2 F1 instances, 2017. Access: 2019-10-15.
- [56] Ilias Giechaskiel, Ken Eguro, and Kasper B. Rasmussen. Leakier Wires: Exploiting FPGA Long Wires for Covert- and Side-Channel Attacks. *ACM Trans. Reconfigurable Technol. Syst.*, 12(3), August 2019.
- [57] Dennis R. E. Gnad, Fabian Oboril, and Mehdi B. Tahoori. Voltage drop-based fault attacks on FPGAs using valid bitstreams. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–7, September 2017.
- [58] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating cache side-channel protections with TLB attacks. In *USENIX Security Symposium*, pages 955–972. USENIX Association, 2018.
- [59] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *NDSS*. The Internet Society, 2017.

- [60] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. AutoLock: Why Cache Attacks on ARM Are Harder Than You Think. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1075–1091, Vancouver, BC, August 2017. USENIX Association.
- [61] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, István Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security Symposium*, pages 217–233. USENIX Association, 2017.
- [62] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is dead: Long live KASLR. In *ESSoS*, volume 10379 of *Lecture Notes in Computer Science*, pages 161–176. Springer, 2017.
- [63] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O’Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 245–261, May 2018.
- [64] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 300–321, Cham, 2016. Springer International Publishing.
- [65] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 300–321, Cham, 2016. Springer International Publishing.
- [66] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A fast and stealthy cache attack. In *DIMVA*, volume 9721 of *LNCS*, pages 279–299. Springer, 2016.
- [67] Berk Gülmemoğlu, Thomas Eisenbarth, and Berk Sunar. Cache-Based Application Detection in the Cloud Using Machine Learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Com-*

- munications Security*, ASIA CCS '17, pages 288–300, New York, NY, USA, 2017. Association for Computing Machinery.
- [68] Berk Gülmezoğlu, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. FortuneTeller: Predicting Microarchitectural Attacks via Unsupervised Deep Learning, 2019.
 - [69] Berk Gülmezoğlu, Andreas Zankl, Thomas Eisenbarth, and Berk Sunar. PerfWeb: How to Violate Web Privacy with Hardware Performance Events. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *Computer Security – ESORICS 2017*, pages 80–97, Cham, 2017. Springer International Publishing.
 - [70] Berk Gülmezoğlu, Andreas Zankl, Caner Tol, Saad Islam, Thomas Eisenbarth, and Berk Sunar. Undermining user privacy on mobile devices using AI. *CoRR*, abs/1811.11218, 2018.
 - [71] Pawan Gupta. *TAA - TSX Asynchronous Abort*. The Linux Kernel Organization, Dec 2020. accessed: Aug 17, 2023.
 - [72] Junichiro Hayata, Jacob C. N. Schuldt, Goichiro Hanaoka, and Kanta Matsuura. On private information retrieval supporting range queries. In *ESORICS (2)*, volume 12309 of *LNCS*, pages 674–694. Springer, 2020.
 - [73] Jamie Hayes and George Danezis. k-fingerprinting: A robust scalable website fingerprinting technique. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1187–1203, Austin, TX, August 2016. USENIX Association.
 - [74] Gottfried Herold, Elena Kirshanova, and Alexander May. On the asymptotic complexity of solving lwe. *Designs, Codes and Cryptography*, 86:55–83, 2018.
 - [75] Tyler Hicks. *MDS – Microarchitectural Data Sampling*. The Linux Kernel Organization, November 2019. accessed: Aug 17, 2023.
 - [76] Benjamin Holmes, Jason Waterman, and Dan Williams. KASLR in the age of MicroVMs. In *EuroSys*, pages 149–165. ACM, 2022.
 - [77] Jann Horn. Speculative execution, variant 4: speculative store bypass, 2018. accessed: Aug 17, 2023.

- [78] Jennifer Huffstetler. Intel processors and FPGAs – better together. <https://itpeernetwork.intel.com/intel-processors-fpga-better-together/>, 2018. accessed: 21-05-19.
- [79] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *IEEE S&P*, pages 191–205. IEEE, 2013.
- [80] Mehmet Sinan Inci, Berk Gülmezoglu, Thomas Eisenbarth, and Berk Sunar. Co-location detection on the cloud. In *COSADE*, volume 9689 of *LNCS*, pages 19–34. Springer, 2016.
- [81] Mehmet Sinan İnci, Berk Gülmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. *IACR Cryptology ePrint Archive*, 2015.
- [82] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In *CHES*, volume 9813 of *LNCS*, pages 368–388. Springer, 2016.
- [83] Intel. *Open Programmable Acceleration Engine*, 1.1.2 edition, 2017. <https://opae.github.io/1.1.2/index.html>.
- [84] Intel. *Acceleration Stack for Intel Xeon CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual*, 1.2 edition, December 2018. <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl-ias-ccip.pdf>.
- [85] Intel. Accelerator cards that fit your performance needs. <https://www.intel.com/content/www/us/en/programmable/solutions/acceleration-hub/platforms.html>, 2018. accessed: 21-05-19.
- [86] Intel. Speculative execution side channel mitigations, 2018. rev. 3.0 accessed: Mar 22, 2023.
- [87] Intel. Intel transactional synchronization extensions (Intel TSX) asynchronous abort. Technical report, Intel Corp., 2019. accessed: Aug 17, 2023.

- [88] Intel. Microarchitectural data sampling. Technical report, Intel Corp., 2019. ver. 3.0, accessed: Aug 17, 2023.
- [89] Intel. Vector register sampling. Technical report, Intel Corp., 2020. accessed: Aug 17, 2023.
- [90] Intel Corporation. *Intel Data Direct I/O Technology (Intel DDIO): A Primer*, 1 edition, 2012.
- [91] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Jun 2016.
- [92] Intel Corporation. *Intel Virtualization Technology for Directed I/O – Architecture Specification*, 3.1 edition, 2019.
- [93] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S\$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing – and Its Application to AES. In *2015 IEEE Symposium on Security and Privacy (SP)*, pages 591–604, May 2015.
- [94] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in intel processors. In *DSD*, pages 629–636. IEEE, 2015.
- [95] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross Processor Cache Attacks. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS ’16, pages 353–364, New York, NY, USA, 2016. Association for Computing Machinery.
- [96] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gülmezoglu, Thomas Eisenbarth, and Berk Sunar. Spoiler: Speculative load hazards boost rowhammer and cache attacks. In *USENIX Security Symposium*, pages 621–637, 2019.
- [97] Saad Islam, Koksai Mus, Richa Singh, Patrick Schaumont, and Berk Sunar. Signature correction attack on dilithium signature scheme. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 647–663. IEEE, 2022.
- [98] Saad Islam, Koksai Mus, Richa Singh, Patrick Schaumont, and Berk Sunar. A signature correction attack on the post-quantum scheme

- dilithium. In *Proceedings of the IEEE European Workshop on Security and Privacy*, 2022.
- [99] J. Jancar, M. Fourné, D. De Almeida Braga, M. Sabt, P. Schwabe, G. Barthe, P. Fouque, and Y. Acar. “they’re not that hard to mitigate”: What cryptographic library developers think about timing attacks. In *2022 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 755–772, Los Alamitos, CA, USA, may 2022. IEEE Computer Society.
 - [100] Patrick Jattke, Victor Van Der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. Blacksmith: Scalable rowhammering in the frequency domain. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 716–734. IEEE, 2022.
 - [101] JC-42.6 Low Power Memories Committee. Low Power Double Data Rate 4 (LPDDR4). Standard JESD209-4B, JEDEC Solid State Technology Association, March 2017.
 - [102] Brian Johannesmeyer, Jakob Koschel, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Kasper: Scanning for generalized transient execution gadgets in the linux kernel. In *NDSS*. The Internet Society, 2022.
 - [103] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International journal of information security*, 1(1):36–63, 2001.
 - [104] Jonas Juffinger, Lukas Lamster, Andreas Kogler, Maria Eichlseder, Moritz Lipp, and Daniel Gruss. Csi: Rowhammer-cryptographic security and integrity against rowhammer. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 236–252. IEEE Computer Society, 2022.
 - [105] Cameron F Kerry and Patrick D Gallagher. Digital signature standard (dss). *FIPS PUB*, pages 186–4, 2013.
 - [106] Salman Abdul Khaliq, Usman Ali, and Omer Khan. Timing-based side-channel attack and mitigation on PCIe connected distributed embedded systems. In *HPEC*, pages 1–7. IEEE, 2021.
 - [107] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael B. Abu-Ghazaleh.

SafeSpec: Banishing the Spectre of a Meltdown with leakage-free speculation. In *DAC*, page 60. ACM, 2019.

- [108] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTH-MEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 189–204, Bellevue, WA, 2012. USENIX.
- [109] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 361–372. IEEE Press, 2014.
- [110] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 974–987, October 2018.
- [111] Vladimir Kiriansky and Carl A. Waldspurger. Speculative buffer overflows: Attacks and defenses. *CoRR*, abs/1807.03757, 2018.
- [112] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.
- [113] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting speculative execution. In *IEEE S&P*, pages 1–19. IEEE, 2019.
- [114] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO’99*, pages 388–397, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.
- [115] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. {Half-Double}: Hammering from the next row over. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3807–3824, 2022.

- [116] Esmail Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *WOOT @ USENIX Security Symposium*. USENIX Association, 2018.
- [117] Jonas Krautter, Dennis R. E. Gnad, and Mehdi B. Tahoori. FPGA-hammer: Remote Voltage Fault Attacks on Shared FPGAs, suitable for DFA on AES. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):44–68, August 2018.
- [118] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical cache attacks from the network. In *IEEE S&P*, pages 20–38. IEEE, 2020.
- [119] Eyal Kushilevitz and Rafail Ostrovsky. Replication is NOT needed: SINGLE database, computationally-private information retrieval. In *FOCS*, pages 364–373. IEEE, 1997.
- [120] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. Rambleed: Reading bits in memory without accessing them. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 695–711. IEEE, 2020.
- [121] Arjen K Lenstra, Hendrik Willem Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische annalen*, 261(ARTICLE):515–534, 1982.
- [122] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, Austin, TX, August 2016. USENIX Association.
- [123] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*, pages 973–990. USENIX Association, 2018.
- [124] Moritz Lipp, Michael Schwarz, Lukas Raab, Lukas Lamster, Misiker Tadesse Aga, Clémentine Maurice, and Daniel Gruss. Nethammer: Inducing rowhammer faults through network requests. In *2020*

IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), pages 710–719. IEEE, 2020.

- [125] Fangfei Liu, Qian Ge, Yuval Yarom, Frank McKeen, Carlos V. Rozas, Gernot Heiser, and Ruby B. Lee. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *HPCA*, pages 406–418. IEEE, 2016.
- [126] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE S&P*, pages 605–622. IEEE, 2015.
- [127] Mingjie Liu and Phong Q Nguyen. Solving bdd by enumeration: An update. In *Topics in Cryptology–CT-RSA 2013: The Cryptographers’ Track at the RSA Conference 2013, San Francisco, CA, USA, February 25–March 1, 2013. Proceedings*, pages 293–309. Springer, 2013.
- [128] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2109–2122. ACM, 2018.
- [129] A. Theodore Marketos, Colin Rothwell, Brett F. Gutstein, Allison Pearce, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals. In *NDSS*. The Internet Society, 2019.
- [130] Debora T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):4–15, 2002.
- [131] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the other side: SSH over robust cache covert channels in the cloud. In *NDSS*. The Internet Society, 2017.
- [132] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer.

Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *24th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2017. Internet Society.

- [133] Victor S Miller. *Use of elliptic curves in cryptography*. Springer, 1986.
- [134] CVE-2019-19962. Available from MITRE, CVE-ID CVE-2019-19962., December 2019.
- [135] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 69–90, Cham, 2017. Springer International Publishing.
- [136] Daniel Moghimi. Medusa [code], 2020.
- [137] Daniel Moghimi. Downfall: Exploiting speculative data gathering. In *USENIX Security Symposium*, pages 7179–7193. USENIX Association, 2023.
- [138] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In *USENIX Security Symposium*, pages 1427–1444. USENIX Association, 2020.
- [139] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL: TPM meets timing and lattice attacks. In *USENIX Security Symposium*, pages 2057–2073. USENIX Association, 2020.
- [140] Benoît Morgan, Eric Alata, Vincent Nicomette, and Mohamed Kaâniche. Bypassing IOMMU protection against I/O attacks. In *LADC*, pages 145–150. IEEE, 2016.
- [141] Benoît Morgan, Eric Alata, Vincent Nicomette, and Mohamed Kaâniche. IOMMU protection against I/O attacks: a vulnerability and a proof of concept. *J. Braz. Comput. Soc.*, 24(1):2:1–2:11, 2018.
- [142] David Mulnix. Intel Xeon processor scalable family technical overview. <https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview>, 2017. Accessed: 2019-07-10.

- [143] K. Mus, Y. Doröz, M. Tol, K. Rahman, and B. Sunar. Jolt: Recovering tls signing keys via rowhammer faults. In *2023 2023 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 1719–1736, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.
- [144] Koksall Mus, Saad Islam, and Berk Sunar. Quantumhammer: a practical hybrid attack on the luov signature scheme. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1071–1084, 2020.
- [145] Onur Mutlu and Jeremie S Kim. Rowhammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(8):1555–1571, 2019.
- [146] Hoda Naghibijouybari, Esmail Mohammadian Koruyeh, and Nael Abu-Ghazaleh. Microarchitectural attacks in heterogeneous systems: A survey. *ACM Comput. Surv.*, 55(7), dec 2022.
- [147] Matus Nemec, Marek Sys, Petr Svenda, Dusan Klinec, and Vashek Matyas. The return of coppersmith’s attack: Practical factorization of widely used rsa moduli. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1631–1648, 2017.
- [148] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding PCIe performance for end host networking. In *SIGCOMM*, pages 327–341. ACM, 2018.
- [149] Nguyen and Shparlinski. The insecurity of the digital signature algorithm with partially known nonces. *Journal of Cryptology*, 15:151–176, 2002.
- [150] Khang T. Nguyen. Usage models for Cache Allocation Technology in the Intel Xeon Processor E5 v4 family, 2016.
- [151] Phong Q Nguyen and Igor E Shparlinski. The insecurity of the elliptic curve digital signature algorithm with partially known nonces. *Designs, codes and cryptography*, 30:201–217, 2003.

- [152] National Institute of Standards and Technology. Digital signature standard (DSS). Technical Report Federal Information Processing Standards Publications (FIPS PUBS) 186-5, U.S. Department of Commerce, Washington, D.C., 2023.
- [153] Wakaha Ogata and Kaoru Kurosawa. Optimum secret sharing scheme secure against cheating. In *Advances in Cryptology—EUROCRYPT’96: International Conference on the Theory and Application of Cryptographic Techniques Saragossa, Spain, May 12–16, 1996 Proceedings 15*, pages 200–211. Springer, 1996.
- [154] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *CCS*, pages 1406–1418. ACM, 2015.
- [155] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006.
- [156] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.
- [157] Andriy Panchenko, Fabian Lanze, Jan Pennekamp, Thomas Engel, Andreas Zinnen, Martin Henze, and Klaus Wehrle. Website fingerprinting at internet scale. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21–24, 2016*. The Internet Society, 2016.
- [158] Alexandra Parkegren and Melker Veltman. Trust in lightweight virtual machines: Integrating TPMs into Firecracker, 2023. Chalmers University of Technology, University of Gothenburg, master thesis.
- [159] PCI-SIG. *PCI Express Base Specification*, 2006. Rev. 2.0.
- [160] PCI-SIG. *Address Translation Services*, 2009. Rev. 1.1.
- [161] Christoph Peglow. Security analysis of hybrid Intel CPU/FPGA platforms using IOMMUs against I/O attacks. Master’s thesis, University of Lübeck, 2020.

- [162] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: exploiting DRAM addressing for cross-cpu attacks. In *USENIX Security Symposium*, pages 565–581. USENIX Association, 2016.
- [163] George Provelengios, Chethan Ramesh, Shivukumar B. Patil, Ken Eguro, Russell Tessier, and Daniel Holcomb. Characterization of Long Wire Data Leakage in Deep Submicron FPGAs. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, pages 292–297, New York, NY, USA, 2019. Association for Computing Machinery.
- [164] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the observer effect for high-precision cache contention attacks. In *CCS*, pages 2906–2920. ACM, 2021.
- [165] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Double Trouble: Combined heterogeneous attacks on Non-Inclusive cache hierarchies. In *USENIX Security Symposium*, pages 3647–3664. USENIX Association, 2022.
- [166] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *USENIX Security Symposium*, pages 1451–1468. USENIX Association, 2021.
- [167] Chethan Ramesh, Shivukumar B. Patil, Siva Nishok Dhanuskodi, George Provelengios, Sébastien Pillement, Daniel Holcomb, and Russell Tessier. FPGA Side Channel Attacks without Physical Access. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 45–52, April 2018.
- [168] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1–18, Austin, TX, August 2016. USENIX Association.
- [169] RedHat, Inc. `/sys/class/iommu/iommu#/devices/`, 2014.
- [170] Eric Rescorla. The transport layer security (tls) protocol version 1.3. Technical report, 2018.

- [171] Vera Rimmer, Davy Preuveneers, Marc Juárez, Tom van Goethem, and Wouter Joosen. Automated feature extraction for website fingerprinting through deep learning. *CoRR*, abs/1708.06376, 2017.
- [172] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS*, pages 199–212. ACM, 2009.
- [173] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of aes. In *Cryptographic Hardware and Embedded Systems, CHES 2010: 12th International Workshop, Santa Barbara, USA, August 17-20, 2010. Proceedings 12*, pages 413–427. Springer, 2010.
- [174] Thomas Rokicki, Clémentine Maurice, Marina Botvinnik, and Yossi Oren. Port contention goes portable: Port contention side channels in web browsers. In *AsiaCCS*, pages 1182–1194. ACM, 2022.
- [175] Thomas Rokicki, Clémentine Maurice, and Michael Schwarz. CPU port contention without SMT. In *ESORICS (3)*, volume 13556 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 2022.
- [176] Gururaj Saileshwar, Bolin Wang, Moinuddin Qureshi, and Prashant J Nair. Randomized row-swap: mitigating row hammer by breaking spatial correlation between aggressor and victim rows. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1056–1069, 2022.
- [177] Falk Schellenberg, Dennis R. E. Gnäd, Amir Moradi, and Mehdi B. Tahoori. An Inside Job: Remote Power Analysis Attacks on FPGAs. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1111–1116, March 2018.
- [178] Falk Schellenberg, Dennis R. E. Gnäd, Amir Moradi, and Mehdi B. Tahoori. Remote Inter-Chip Power Analysis Side-Channel Attacks at Board-Level. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–7, November 2018.
- [179] Claus-Peter Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theoretical computer science*, 53(2-3):201–224, 1987.

- [180] Claus-Peter Schnorr and Martin Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical programming*, 66:181–199, 1994.
- [181] Michael Schwarz. PTEditor: A small library to modify all page-table levels of all processes from user space for x86_64 and ARMv8. <https://github.com/misc0110/PTEditor>, April 2019.
- [182] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*, pages 753–768. ACM, 2019.
- [183] Michael Schwarz, Cl  mentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript. In Aggelos Kiayias, editor, *Financial Cryptography and Data Security*, pages 247–267, Cham, 2017. Springer International Publishing.
- [184] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat*, 15, 2015.
- [185] Debendra Das Sharma. Compute express link. Whitepaper, Compute Express Link Consortium, 2019.
- [186] Anton Shilov. Intel’s Sapphire Rapids formal launch date revealed, 2022. Access: 2020-12-04.
- [187] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 639–656, Santa Clara, CA, August 2019. USENIX Association.
- [188] Olin Sibert, Phillip A. Porras, and Robert Lindell. The Intel 80×86 processor architecture: pitfalls for secure systems. In *IEEE S  P*, pages 211–222. IEEE, 1995.
- [189] Mingtian Tan, Junpeng Wan, Zhe Zhou, and Zhou Li. Invisible probe: Timing attacks with PCIe congestion side-channel. In *IEEE S  P*, pages 322–338. IEEE, 2021.

- [190] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Packet Chasing: Spying on Network Packets over a Cache Side-Channel. *arXiv preprint arXiv:1909.04841*, 2019.
- [191] Andrei Tatar, Radhesh Krishnan, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA, 2018. USENIX Association.
- [192] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. TLB;DR: Enhancing TLB-based attacks with TLB desynchronized reverse engineering. In *USENIX Security Symposium*, pages 989–1007. USENIX Association, 2022.
- [193] Thore Tiemann, Zane Weissman, Thomas Eisenbarth, and Berk Sunar. Iotlb-sc: An accelerator-independent leakage source in modern cloud systems. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*, ASIA CCS ’23. ACM, July 2023.
- [194] Thore Tiemann, Zane Weissman, Thomas Eisenbarth, and Berk Sunar. *Microarchitectural Vulnerabilities Introduced, Exploited, and Accelerated by Heterogeneous FPGA-CPU Platforms*, pages 203–237. Springer International Publishing, Cham, 2024.
- [195] M. Tol, S. Islam, A. J. Adiletta, B. Sunar, and Z. Zhang. Don’t knock! rowhammer at the backdoor of dnn models. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 109–122, Los Alamitos, CA, USA, jun 2023. IEEE Computer Society.
- [196] Daniël Trujillo, Johannes Wikner, and Kaveh Razavi. Inception: Exposing new attack surfaces with training in transient execution. In *USENIX Security Symposium*, pages 7303–7320. USENIX Association, 2023.
- [197] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES Implemented on Computers with Cache. In Colin D. Walter, Çetin K. Koç, and Christof Paar,

- editors, *Cryptographic Hardware and Embedded Systems - CHES 2003*, pages 62–76, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
- [198] Paul Turner. Retpoline: a software construct for preventing branch-target-injection, 2018. accessed: Mar 22, 2023.
 - [199] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1675–1689, 2016.
 - [200] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16*, pages 1675–1689, New York, NY, USA, 2016. Association for Computing Machinery.
 - [201] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: rogue in-flight data load. In *IEEE S&P*, pages 88–105. IEEE, 2019.
 - [202] Stephan van Schaik, Alyssa Millburn, genBTC, Paul Menzel, jun1x, Stephen Kitt, pit fr, Sebastian Österlund, and Cristiano Giuffrida. Ridl [code], 2020.
 - [203] Pepe Vila, Boris Köpf, and José F. Morales. Theory and practice of finding eviction sets. In *IEEE S&P*, pages 39–54. IEEE, 2019.
 - [204] Pepe Vila, Boris Köpf, and José Francisco Morales. Theory and Practice of Finding Eviction Sets. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 39–54, May 2019.
 - [205] Zane Weissman, Thore Tiemann, Thomas Eisenbarth, and Berk Sunar. Microarchitectural security of aws firecracker vmm for serverless cloud platforms, 2023.

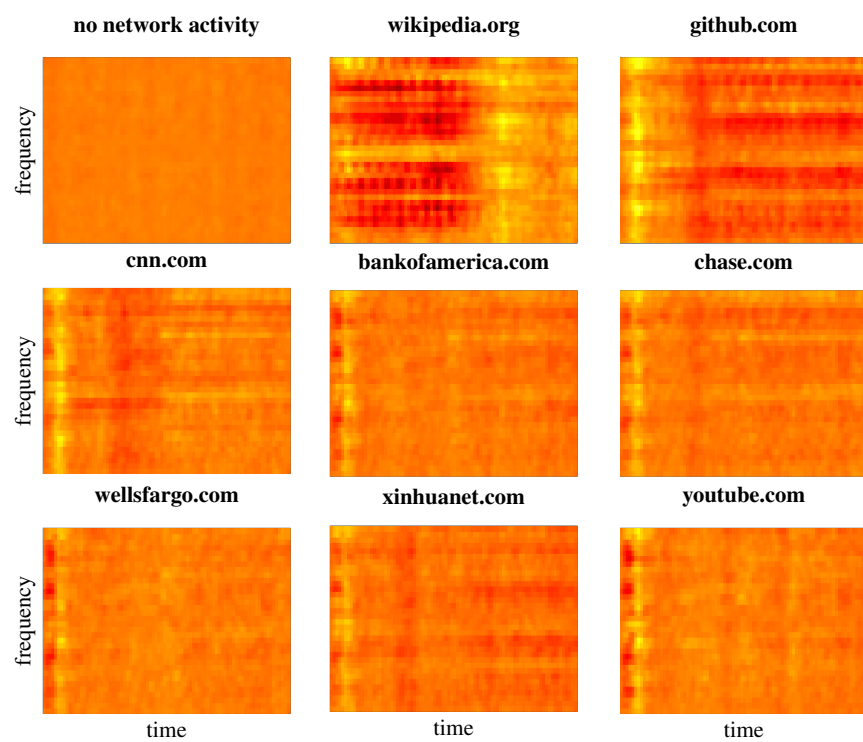
- [206] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. Jackhammer: Efficient rowhammer on heterogeneous fpga-cpu platforms. *arXiv preprint arXiv:1912.11523*, 2019.
- [207] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. JackHammer: Efficient rowhammer on heterogeneous FPGA-CPU platforms. *IACR TCHES*, 2020(3):169–195, 2020.
- [208] Johannes Wikner and Kaveh Razavi. RETBLEED: arbitrary speculative code execution with return instructions. In *USENIX Security Symposium*, pages 3825–3842. USENIX Association, 2022.
- [209] Johannes Wikner, Daniël Trujillo, and Kaveh Razav. Phantom: Exploiting decoder-detectable mispredictions. In *MICRO (to appear)*. IEEE, 2023.
- [210] Marc F. Witteman, Jasper G. J. van Woudenberg, and Federico Menarini. Defeating RSA Multiply-Always and Message Blinding Countermeasures. In Aggelos Kiayias, editor, *Topics in Cryptology – CT-RSA 2011*, pages 77–88, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [211] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyperspace: High-speed Covert Channel Attacks in the Cloud. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 159–173, Bellevue, WA, 2012. USENIX.
- [212] Jietao Xiao, Nanzi Yang, Wenbo Shen, Jinku Li, Xin Guo, Zhiqiang Dong, Fei Xie, and Jianfeng Ma. Attacks are forwarded: Breaking the isolation of microVM-based containers through operation forwarding. In *USENIX Security Symposium*, pages 7517–7534. USENIX Association, 2023.
- [213] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One Bit Flips, One Cloud Flops: Cross-VM row hammer attacks and privilege escalation. In *USENIX Security Symposium*, pages 19–35, 2016.

- [214] Xilinx. Accelerator Cards. <https://www.xilinx.com/products/boards-and-kits/accelerator-cards.html>, 2019. Accessed: 2019-10-15.
- [215] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. InvisiSpec: Making speculative execution invisible in the cache hierarchy. In *MICRO*, pages 428–441. IEEE Computer Society, 2018.
- [216] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 888–904, May 2019.
- [217] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium*, pages 719–732. USENIX Association, 2014.
- [218] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium*, pages 719–732. USENIX Association, 2014.
- [219] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. COLORIS: a dynamic cache partitioning system using page coloring. In *PACT*, pages 381–392. ACM, 2014.
- [220] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The true cost of containing: A gVisor case study. In *HotCloud*. USENIX Association, 2019.
- [221] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds. In Fabian Monrose, Marc Dacier, Gregory Blanc, and Joaquin Garcia-Alfaro, editors, *Research in Attacks, Intrusions, and Defenses (RAID)*, pages 118–140, Cham, 2016. Springer International Publishing.
- [222] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS '12*, pages 305–316, New York, NY, USA, 2012. Association for Computing Machinery.

- [223] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In *CCS*, pages 990–1003. ACM, 2014.
- [224] Mark Zhao and G. Edward Suh. FPGA-Based Remote Power Side-Channel Attacks. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 229–244, May 2018.
- [225] Shixuan Zhao, Pinshen Xu, Guoxing Chen, Mengya Zhang, Yinqian Zhang, and Zhiqiang Lin. Reusable enclaves for confidential serverless computing. In *USENIX Security Symposium*, pages 4015–4032. USENIX Association, 2023.
- [226] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. A Software Approach to Defeating Side Channels in Last-Level Caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16*, pages 871–882, New York, NY, USA, 2016. Association for Computing Machinery.

Appendix A

Mean Spectral Data Per Site



Mean of all training observations for each class in the model, after all pre-processing.

Appendix B

Extended RIDL Mitigations

Table

Table [B.1](#). Mitigations necessary to protect the host vs. Firecracker victims from RIDL and other MDS attacks. The recommended `nosmt` mitigation protects against most but not all of these variants. All proof of concepts were tested on Firecracker v1.0.0, v1.4.0, and v1.5.0 with identical results.

Exploit	Details			Bare Metal		Firecracker			TSX required?
	Common Name	Target Buffer	Fault Type	nosmt	mds	nosmt (H)	mds (H)	mds (VM)	
alignment.write		Fill Buffer	Alignment	+	+	+	+	✗	no
pgtable.leak.notsx		Fill Buffer	Page	✗ ^b	✓	✗ ^b	✓	✓	no
ridl.basic	RIDL	Fill Buffer	Page	✓	✓	N/A ^c	N/A ^c	N/A ^c	no
ridl.invalidpage		Fill Buffer	Page	✓	✗	N/A ^c	N/A ^c	N/A ^c	no
pgtable.leak		Fill Buffer	TSX abort	✓	✗	✓	✗	✗	yes
taa.read	RIDL/TAA	Fill Buffer	TSX abort	✓	✗	✓	✗	✗	yes
taa.basic		Fill Buffer	TSX abort	✓	✗	N/A ^c	N/A ^c	N/A ^c	yes
verw.bypass.1ides	RIDL/TAA	Fill Buffer	TSX abort	✓	✗	✓	✗	✗	yes
loadport	RIDL	Load Port	Page	✓	✗	✓	✗	✗	no
vsrs	RIDL/VRS	Store buffer	Page	✓	✗	✓	✗	✗	no
cpuid.leak	Crosstalk	Fill Buffer	Page	✓	✗	N/A ^a	N/A ^a	N/A ^a	no

- ✓ – mitigation prevents side-channel attack.
- ✚ – mitigation prevents attack only in combination with other mitigation(s) marked ✚.
- ✗ – mitigation has no effect on this attack.
- ^a CPUID instruction is emulated by virtual machine and has no microarchitectural effect.
- ^b This attack leaks information about pages used in its own thread.
- ^c PoCs had to be modified slightly to run in two processes before they could be tested in the virtual machine. These PoCs did not work on bare metal or in virtual machines when split into two processes.