UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Exploiting Cache Side-Channels on CPU-FPGA Cloud Platforms

Cache-basierte Seitenkanalangriffe auf CPU-FPGA Cloud Plattformen

Masterarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Thore Tiemann

ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth

Lübeck, den 09. Januar 2020

Abstract

Cloud service providers invest more and more in FPGA infrastructure to offer it to their customers (IaaS). FPGA use cases include the acceleration of artificial intelligence applications. While physical attacks on FPGAs got a lot of attention by researchers just recently, nobody focused on micro-architectural attacks in the context of FPGA yet.
We take the first step into this direction and investigate the timing behavior of memory reads on two Intel FPGA platforms that were designed to be used in a cloud environment: a Programmable Acceleration Card (PAC) and a Xeon CPU with an integrated FPGA that has an additional coherent local cache. By using the Open Programmable Acceleration Engine (OPAE) and the Core Cache Interface (CCI-P), our setup represents a realistic cloud scenario. We show that an Acceleration Functional Unit (AFU) on either platform can, with the help of a self-designed hardware timer, distinguish which of the different parts of the memory hierarchy serves the response for memory reads. On the integrated platform, the same is true for the CPU, as it can differentiate between answers from the FPGA cache, the CPU cache, and the main memory. Next, we analyze the effect of the caching hints offered by the CCI-P on the location of written cache lines in the memory hierarchy and show that most cache lines get cached in the LLC of the CPU. We use this behavior to construct a covert channel of 94.98 kBit/s from the FPGA to the CPU.

Zusammenfassung

Cloud Service Provider setzen immer mehr auf die Vermietung von FPGA-Infrastruktur (IaaS). FPGAs werden beispielsweise zur Beschleunigung von Anwendungen mit künstlicher Intelligenz eingesetzt. Während physikalische Angriffe auf FPGAs erst kürzlich viel Aufmerksamkeit in der Forschung bekamen, wurden Mikroarchitekturangriffe bisher in diesem Kontext nicht betrachtet.
Wir machen einen ersten Schritt in diese Richtung und untersucht das Zeitverhalten von Speicher-Schreibzugriffen auf zwei für den Cloud-Einsatz entworfenen FPGA-Plattformen von Intel: einer Programmable Acceleration Card (PAC) sowie einer Xeon CPU mit integriertem FPGA und eigenem kohärenten Cache. Durch die Verwendung der Open Programmable Acceleration Engine (OPAE) und des Core Cache Interface (CCI-P) entspricht unsere Versuchsumgebung einem realistischen Cloud-Szenario. Wir weisen nach, dass eine Acceleration Functional Unit (AFU) auf beiden Plattformen mittels eines selbst gebauten Hardware-Timers feststellen kann, aus welchem Teil der Speicher-Hierarchie die Antwort auf eine Lese-Anfrage geliefert wird. Auch die CPU des integrierten Systems ist in der Lage, anhand der Speicherladezeit zu unterscheiden, ob eine Cacheline aus dem FPGA-Cache stammt, aus dem Hauptspeicher geladen wurde oder bereits im Cache der CPU lag. Anschließend analysieren wir im Detail, wie sich die Caching Hints des CCI-P auf den Speicherort einer geschriebenen Cacheline auswirken und zeigen, dass diese in den meisten Fällen im LLC der CPU enden. Dieses Verhalten nutzen wir aus, um einen Cover Channel mit 94,98 kBit/s vom FPGA zur CPU zu konstruieren.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

_____

Lübeck, den 09. Januar 2020

Acknowledgements

An erster Stelle sind hier meine Freundin und meine Familie zu nennen. Vielen Dank, dass ihr mich best möglich über alle Jahre meines Studiums hinweg unterstützt habt. Sei es durch Finanzspritzen, das Korrekturlesen meiner Arbeiten oder einfach nur ein offenes Ohr oder eine Umarmung und schöne Worte.
Des Weiteren gilt mein Dank dem Institut für IT-Sicherheit, ohne das ich dieses Thema nie bearbeitet hätte. Vielen Dank Prof. Eisenbarth und Jan für die Betreuung und hilfreichen Diskussionen sowie das Korrekturlesen meiner Arbeit, auch noch auf den letzten Drücker.
And thank you Okan, for being a great office partner enduring my little freak-outs during the experimentation phase.
A great shout-out goes to the team from the Worcester Polytechnic Institute, Zane, Daniel, and Prof. Sunar for the weekly discussions, and Koksal for hosting me a whole month at your place.
Last but not least, a big thank you to Evan and Alpa from Intel for donating hardware and providing helpful background knowledge and clarifications in our monthly meetings.
Thanks to all of you!

Contents

1 Introduction

1.1 Motivation

Since the foundation of the internet, computers entered our everyday life on all levels with massive speed. Today, we have a rich diversity of devices connected to the internet like servers, computers, smartphones. Companies use technology to digitize their business processes and offer their products in online market places. To meet the companies' demand for computational powerful but yet cheap devices, leading tech companies like Google and Amazon started offering computational resources in the cloud to be rented at flexible rates. At first, the requested computation power was delivered by common server processors. But with the rise of artificial intelligence, accelerators were developed to be used in the cloud. Such accelerators may be hardwired special-purpose chips such as graphics cards or ASICs, or Field Programmable Gate Arrays (FPGAs) that can be reprogrammed to fit any use case. In direct comparison, ASICs will always outperform FPGAs when comparing computational speed. But since ASIC-development and production is far more costly than providing FPGAs, FPGAs win the performance/cost tradeof. That is why big cloud service providers such as Amazon AWS [Ama17] and Alibaba Cloud [Ali19] started offering FPGA as an Infrastructure as a Service (IaaS) product.
After a so far unknown vulnerability of the Windows operating system (OS) became public, a ransomware flood swept through many companies and institutions that did not patch their systems in time. This incident showed, that the security of computer systems is extremely important as critical infrastructure such as hospitals were taken offline due to a ransomware infection. Most attacks on computer systems happen because a vulnerable service realized in software is connected to the internet and therefore directly accessible for an attacker and for years, the underlying hardware was assumed to be secure and free of bugs. Experts knew this assumption to be wrong the latest since the Pentium FDIV bug. The majority, however, started realizing the threat of micro-architectural attacks when the vulnerabilities Meltdown and Spectre [Gib18] were published as they applied to the majority of processors that are currently used. Micro-architectural attacks are attacks exploiting bugs, vulnerabilities, and especially side-channels (SCs) leaking information from other processes usually running on the same server through shared resources. As such attacks unfold their power in multi-tenant scenarios, they pose a great threat to cloud platforms as it is their business model to rent out logically separated virtual environments

running on shared hardware.
Since FPGAs are meant to accelerate applications, this work takes a look at to what extent micro-architectural attacks can be accelerated by FPGAs in the cloud.

1.2 Scope

This work is meant to give a first assessment of the potential of micro-architectural SCs introduced to cloud platforms by making FPGAs available to cloud customers in an IaaS manner. To do so, we will examine two Central Processing Unit (CPU)-FPGA platforms designed to be used in a cloud environment. The Intel PAC with Arria 10 FPGA is already available to customers through Alibaba Cloud [Ali19]. The second platform is a prototype Xeon CPU that has an Arria 10 FPGA with coherent cache integrated into the processor package. As we use the Intel Acceleration Stack (IAS) to configure and access the FPGA, we have a realistic cloud scenario that we want to answer the following three questions in:

- Can a timer realized in hardware and running on an FPGA be used to find timing leakages introduced by the memory hierarchy? If yes, this would circumvent security measures that deny accessing precise timers from user-space.
- Does an FPGA with coherent memory access have an advantage over FPGAs without a coherent connection when it comes to launching cache attacks? If yes, this has to be considered when designing the emerging coherent accelerator interconnect protocols CCIX and CXL.
- To what extent does the current IAS configuration support or limit the possibilities of launching cache attacks from the FPGA? Intel is actively extending and improving the IAS, so findings to this question have a direct consequence on future product safety.

When finding leakage we think is exploitable, we will create proofs-of-concept whenever possible, considering the time constraints of this work. All findings will be reported and discussed with Intel personal in monthly meetings to make the platforms more secure.

1.3 Organization

We start with exhaustive literature research to give the background knowledge about FPGAs in the cloud and existing security frameworks before the memory architecture of modern CPUs and the historic development of the SCs in general and the most important

cache attacks, in particular, are explained during chapter 2. In chapter 3, we introduce important Intel terminology. The general specifications of the platforms used in this work are given and the IAS is explained, focusing on the OPAE and the CCI-P. Also, the exact specification of our setup is given. The main contribution follows in chapter 4. This is where we describe our experiments and the timing leakages we found. We reverse-engineer the behavior of caching hints available on the FPGAs before constructing a covert channel between an AFU on the FPGA and a cooperating software process on the CPU. Our results are concluded in chapter 5. We summarize our results and name countermeasures to minimize the impact of ou findings. We finish by discussing open problems of this work and other future fields of research, especially highlighting the field of making multi-tenancy possible on FPGAs.

We start by getting an overview of security frameworks and attacks that were already published in the context of FPGAs in the cloud. As we shall see, security mainly focuses on protecting intellectual property (IP) through bitstream encryption and integrity checks. Also, most works focus on the Hardware Acceleration as a Service (HAaaS) scenario where cloud provides offer access to Acceleration Functions (AFs) to their customers. However, we are interested in IaaS scenarios, where customers can design and run their own designs on the FPGA. Next, we give background about the memory and cache architecture used in modern server CPUs as this is crucial to successfully launch cache attacks. We give an overview of historic power and EM SCs and present works that apply such attacks to cloud environments with access to FPGAs. Those works make us believe that micro-architectural attacks and especially cache attacks could be accelerated by designing hardware optimized for such attacks. To complete the background chapter, we summarize different techniques for cache attacks and highlight those we think are applicable to our scenario.

Afterward, we present the FPGA platforms we are going to work on en detail, including the IAS that is used in cloud environments to configure the FPGA and connect software applications with AFUs. While reading the documentations [Int17b] and specifications [Int18a], we identify possible timing leakages and interface functionality we will later try to verify and exploit.

To do so and also answer our first research question, we then design a timer in hardware. The timer immediately enables us to verify the timing behavior described in [Int18a]. We further investigate the CCI-P available for AFU developers and the caching hints that can be passed to the interface when doing Direct Memory Access (DMA). Although documented otherwise, we find all caching hints sent with write requests to behave similarly and learn, that they are in fact ignored by the CCI-P at the moment. However, the configuration of the CPU makes the homing agents push most of the cache lines written to

by the AFU to the last level cache (LLC), enabling us to evict data from the cache. In the end, we use findings and the fact that the IAS helps to construct eviction sets to construct a covert channel from an AFU implemented in the FPGA to a process running on the CPU before concluding our work.

2 Background

This chapter presents existing works about FPGAs in the cloud and first security frameworks that mainly focus on bitstream protection. Afterward, the relevant part of a common memory hierarchy is explained including cache architecture and how memory addresses map to cache sets. The chapter closes with an overview of side-channel attacks (SCAs) and related countermeasures and the presentation of important cache attack techniques.

2.1 FPGAs in the Cloud

For a couple of years now FPGAs are used to speed up cloud computing. First cloud providers even make FPGA infrastructure available to their customers (e.g. Amazon EC2 F1 or Alibaba Cloud F1/F3 instances [Ama17, Ali19]). But while the security of multi-tenant CPU platforms is deeply researched for at least a decade now, secure multi-tenancy is still an open field with many unsolved problems on FPGA platforms.
In 2014, Chen et al. [CSZ+14] defined *abstraction*, *sharing*, *compatibility*, and *security* as the four major requirements for enabling FPGAs in the cloud. They proposed a framework for integrating FPGAs in the cloud through FPGA virtualization, thereby primarily addressing the first three requirements. The security requirement gets only a little attention as they mainly refer to Eguro and Venkatesan's work [EV12]. Eguro and Venkatesan apply techniques for bit-stream protection against IP piracy to implement a root of trust and defend client data and computation from external and hypervisor attackers.
Chen et al. discuss two scenarios: (i) users can run their own bit-streams on the FPGA provided (IaaS) or (ii) users can use AFs provided by the cloud provider (HAaaS). They achieve reasonable results for the second scenario. The first scenario is described as more complicated because running a user's bitstreams on the FPGA requires knowledge of the actual underlying FPGA hardware. The available space for a user's AF design depends on the amount and size of other AFs already running on the FPGA, which raises security concerns. They state that these concerns cannot be removed until FPGAs and their toolchains support hardware-independent bit-file generation.
Hategekimana et al. [HMMPB18] designed a new security framework for clouds integrating FPGAs that has less overhead than the one proposed by Chen et al. It allows an AF to inherit software security policies of the virtual machine calling the AF. But just like

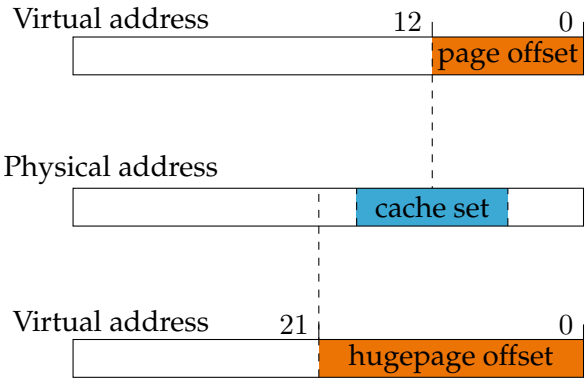
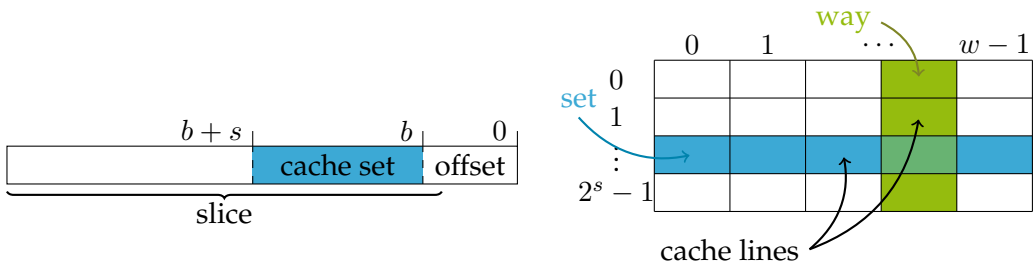


Figure 2.1: Virtual page addresses share the last 12 and 2 MiB hugepage addresses share the last 21 bits with their corresponding physical address. For hugepage addresses, the cache set index is fully contained within the shared offset bits.

Chen et al., they focus on the HAaaS scenario while claiming without proof that their framework is also securely applicable to the IaaS scenario.

2.2 Memory and Cache Architecture

In the main memory, buffers are allocated in *pages* of 4 KiB. Additionally, modern OSs support hugepages that can hold 2 MiB or even 1 GiB. `mmap` can be used to map a memory (huge)page into the virtual address space of a process. When a virtual address is accessed, the Memory Management Unit (MMU) does the virtual-to-physical address translation and sends the physical address to the memory controller. For 4 KiB pages, the virtual and corresponding physical address are equal in the last 12 least-significant bits, as they describe the offset within the memory page. Consequently, virtual and physical addresses for 2 MiB and 1 GiB hugepages share the last 21 and 30 bits, respectively (cf. Figure 2.1).
As CPUs became faster, caches were placed between the main memory and the CPU core to reduce memory access latency. Caches are organized in 2^s cache sets that hold up to w cache lines of size 2^b bytes (cf. Figure 2.2b). The total cache (slice) size computes as $2^{s+b} \cdot w$ bytes. w is called the associativity or way-ness of the cache. Caches with only one way per set ($w = 1$) are called *directly mapped* and caches with only one set ($s = 1$) are called *fully associative*. If neither $s = 1$ nor $w = 1$, the cache is called *w-way set associative*.
The b least significant bits of a physical memory address define the offset within a cache line to enable single-byte addressing. The following s address bits define the mapping of a memory address to a cache set (cf. Figure 2.2a). Also, modern Intel LLCs are partitioned into l slices, each holding 2^s sets. The amount of slices l is equal to the number of physical CPU cores. Which slice a memory address belongs to is determined by an undoc-



(a) A physical memory address determines the cache slice and set it is mapped to.

(b) A cache (slice) is organized in sets and ways.

Figure 2.2: Cache (slice) layout and physical memory address mapping. An address is mapped to any of the ways belonging to the same set.

umented deterministic slice selection function, that depends on a subset of the physical address bits. For many Intel CPUs, the slice selection function was reverse-engineered in the past [IES15b, MLSN+15].

Memory addresses that map to the same cache set are called *congruent*. If an address is to be loaded into a cache set that already contains w other cache lines, one of these cache lines gets *evicted* and the new cache line is placed in the previously occupied way. The cache line to be evicted is chosen depending on the *replacement policy* of the cache. One can think of many different replacement policies such as "first in first out" (FIFO) always replacing the oldest cache line, "least recently used" (LRU), "least frequently used" (LFU), or simply a random replacement policy. For average cache access patterns, LRU gives good results when comparing cache miss rates. However, with an increasing number of cache lines in the cache, LRU becomes more difficult to implement. That is why in practice often pseudo-LRU is used to approximate LRU [HP12]. The details of the "Quad-Age LRU" replacement policy used in Intel caches are undocumented and therefore unknown. However, the replacement policies used by some Intel processors were reverse-engineered just recently [BMME19, VGGK19].

A cache c_1 is called to be *inclusive* of another cache c_2, if the cache lines stored in c_2 are a subset of the cache lines present in c_1. c_1 is called *exclusive* of c_2, if the intersection of the cache lines stored in c_1 and c_2 is empty. If a cache is neither inclusive nor exclusive it is called *non-inclusive-non-exclusive* or for short *non-inclusive*. On Intel processors, the LLC usually is inclusive of the Level 1 cache (L1) and the Level 2 cache (L2). Since each core has its own L1 and L2, an inclusive LLC grows in the number of cores. That is why recent Intel server CPUs, starting with the Skylake architecture, have non-inclusive caches [Mul17].

A set of congruent addresses that fills an entire cache set when accessed and therefore evicts all other cache lines in the same set is called an *eviction set*. Eviction sets play an important role in the context of cache attacks, which are described later in subsection 2.3.2.

Finding eviction sets is easy when the cache replacement policy and the slice selection function are known. Otherwise, an attacker has to search for congruent addresses by timing memory accesses and thereby finding colliding addresses [VKM18].
On systems with multiple caches, a coherence protocol is used to keep the data in a coherent state that guarantees that the CPU never computes on stale data. A widely known and used protocol is the MESI protocol. The letters in MESI stand for the states a cache line can be in: Modifies, Exclusive, Shared, or Invalid. Reading a previously uncached cache line loads it in the Exclusive state. By reading the same cache line into another cache, the state of both cache lines is changed to Shared. If one of the shared cache lines is written to, the written cache line's state changes to Modified while the second occurrence gets invalidated (state Invalid) and therefore evicted. If now a read to the invalidated cache line happens, the cache line in the Modified state is written back to the main memory, effectively evicting it from the cache, before the cache line is sent to the reading CPU core [HP12]. Intel uses an extended version of the MESI protocol called MESIF with an extra Forward (F) state to organize cache coherence. If a cache line is in the Exclusive state and another cache requests the same cache line, the cache already holding the cache line can forward the data to the requesting cache, making a slow read from the main memory unnecessary. The receiving cache sets the state of the cache line to Shared and the sending cache changes the state to Forward [BFB+17].

2.3 Side Channel Attacks

Next, we give an overview of side-channels (SCs) and side-channel attacks (SCAs) in different scenarios. We start with the early timing and power SCs. We then introduce the most important cache attack variants and countermeasures, before finishing with recent results in the field of micro-architectural attacks in general.

2.3.1 Timing, Power, and EM Attacks

In the field of SCAs, Paul Kocher was one of the pioneers. In 1996 he broke implementations of RSA, DSA, and Diffie-Hellman using a timing SC [Koc96]. The attack focused on tamper-proof devices like smart-cards rather than common computers as they contain private key material that is not supposed to be extracted by the user. As a countermeasure Kocher proposed to use constant time implementations only, as they remove the timing SC.
Three years later he developed power SC-based attacks such as Simple Power Analysis (SPA) and Differential Power Analysis (DPA) [KJJ99]. Power SCs become available if the implementation contains secret dependent branches. As branches contain different in-

structions and instructions differ in their power consumption the secret can be extracted from the power trace. If the simple power trace does not reveal the secret directly, a differential approach may be taken. The power consumption also correlates with the data being processed as, depending on the underlying technology, storing a value containing many 1 bits consumes more energy than storing a value with many 0 bits. This difference becomes visible in differential power attacks again enabling the attacker to extract secret key material. To circumvent power analysis attacks, implementations have to be free of secret and data-dependent branches. In addition, masking techniques reduce the correlation between the processed data and the power consumption. In cases, where a power leakage is assumed to exist but the power consumption cannot be probed directly (e.g. because of a case physically protecting the chip), electromagnetic (EM) emissions can be measured and used instead in many cases.

The first timing, power, and EM SCs needed physical access to the device under attack. In the context of FPGAs, this changed lately. In 2017 Giechaskiel et al. [GRE17] developed a covert channel between two AFs. They showed that neighboring long wires in an interconnect routing channel can be used as a pair of sender and receiver as logic values carried on a long wire influence the delay of long wires placed next to it. This covert channel is applicable to FPGAs in the cloud and does not need physical access to the FPGA anymore. Just a year later they expanded their covert channel and were able to use it as an SC to leak information from neighboring AFs [GRE18]. While both works ran their experiments on Xilinx SRAM FPGAs, Ramesh et al. [RPD+18] showed that the threat of long wire leakages is also available on several Intel SRAM FPGAs. Additionally, they extracted an AES key from an AES hardware running next to the attacker. This was done by using long wire leakage from one S-box input combined with the analysis of the encrypted output. By doing so, they demonstrated that SCAs against AFs were indeed possible without physical access to the FPGA.

Attacks involving severe voltage fluctuation are hardly applicable in cloud scenarios. However, Gnad et al. [GOT17] showed that this kind of attack is possible for an attacker that can run self-designed AFs on an FPGA. The authors were able to cause voltage fluctuations causing the FPGA to crash within microseconds and only restart after manual power-cycling. Therefore, this attack may be used for efficiently running a Denial-of-Service (DoS) against FPGA resources on a cloud platform. They improved their attack to cause carefully chosen voltage fluctuations [KGT18], capable of placing voltage faults in a neighboring AF's AES computation that can be used for a Differential Fault Analysis (DFA) attack on AES.

At last, even the assumption that power analysis attacks such as SPA and DPA require physical access and specialized hardware was proven wrong in the case of FPGA-IaaS-

clouds. To do so, Schellenberg et al. [SGMT18] designed sensor AFs to be placed on the FPGA that are capable of measuring voltage fluctuations due to power consumption. Based on that work, Zhao and Suh [ZS18] showed the practicability of such measurements by running a successful power analysis attack against an RSA crypto-module placed on the same FPGA. On top, they present the ability of FPGA-based power-monitoring a CPU on the same chip and, based on that monitor, run a power analysis attack against an RSA program executed on the CPU.
As we can see from the literature given, introducing FPGAs to the cloud opens attack vectors that did not exist without them, such as remote DFA, SPA, and DPA. This is because FPGAs enable the attacker to provide the hardware needed to launch SC-based attacks. Depending on the interconnection between CPU and FPGA, an FPGA can be used to attach arbitrary hardware such as a USB or Thunderbolt device [MRG+19] to the cloud, again enlarging the attack surface.

2.3.2 Cache Attacks

When processors became faster than the main memory, caches were introduced to minimize the delay of memory accesses. As accesses to cached data return faster, caches introduced a potential timing SC. In 2004 Bernstein [Ber04] showed, that this SC was not only of theoretic interest but a real threat. He did so by fully recovering an AES key by monitoring which parts of memory get cached during cryptographic operations.
Since then a couple of available techniques exploiting cache SCs were developed. Osvik et al. [OST06] used two approaches to recover an AES key: *Evict+Time (E+T)* and *Prime+Probe (P+P)*. At first, E+T and P+P were applicable to the L1 only, so attackers and victims had to reside on the same core. Yarom et al. [YF13] proposed *Flush+Reload (F+R)* as a high-resolution cache SC applicable to the Level 3 cache (L3) which makes cache attacks possible across cores if memory de-duplication is turned on.
Two years later, in 2015, Liu et al. [LYG+15] and Irazoqui et al. [IES15a] adapted P+P for L3 attacks, thereby removing the need of memory de-duplication, and showed that the attack could be run cross-core and even across virtual machines (VMs). After another year, Irazoqui et al. were able to improve their attack to even work across packages [IES16]. A similar result was achieved by Lipp et al. [LGS+16] as they ported known cache attacks working on Intel CPUs to ARM processors and made them cross-CPU capable. Gruss et al. [GSM15] presented *Cache Template Attacks*, a technique that profiles a victim's cache accesses and automatically exploits cache-based data leakage, rendering cache attacks even more powerful. Also, Gruss et al. [GMWM16] proposed *Flush+Flush (F+F)* as a fast and stealthy alternative to F+R.
To speed-up data transfers between peripherals and the CPU, Intel invented Direct Data

I/O (DDIO). It gives peripherals the ability to directly write to a subset of cache ways per cache set of the CPU's LLC. While peripheral attackers cannot run P+P attacks against processes on the CPU due to only being able to prime parts of a cache set, this situation can still lead to attacks against other peripherals as all peripherals are limited to the same subset of ways per cache set [KGA+20].

Flush+Reload

Flush+Reload (F+R) [YF13] gives the attacker information about the victim's behavior with cache line granularity. To do so, the attacker cycles over three steps:

1. The attacker flushes the cache line that is to be monitored.
2. After flushing the monitored cache line, she waits for the victim to execute.
3. Later, she reloads the flushed line and measures the latency.

If the latency measured during step three is low, the cache line is served from the cache hierarchy. That means, that the cache line was accessed by the victim during its execution. If the access latency is high, the cache line was loaded from the main memory, meaning that the victim did not access it during its execution.
F+R works across cores and even across sockets, as long as the LLC is coherent, as is the case with many modern multi-CPU systems. These attacks are limited to shared memory scenarios, where victims and attackers share data or instructions, as is the case with shared libraries on systems where memory de-duplication is enabled.

Evict+Reload

Evict+Reload (E+R) [LGS+16] is similar to F+R. In an E+R attack, if the system does not have a flush instruction or disabled its execution from userspace, the attacker can, instead, evict the desired cache line by accessing congruent cache lines that form an eviction set during step one.
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 LLC is inclusive. If the LLC is non-inclusive the attacker can attack the inclusive directory structure used to ensure coherency instead [YSG+19].

Flush+Flush

Flush+Flush (F+F) [GMWM16], similar to F+R, gives the attacker cache line granularity and consists of three steps. The only difference is in the third step where the attacker

flushes the cache line again and measures the execution time of the flush instruction instead of the memory access. F+F is faster than F+R as the second flush phase can be used as the first flush for another run. However, like F+R, F+F is limited to shared data/instruction scenarios with access to a flush instruction.

Prime+Probe

Prime+Probe (P+P) gives the attacker a more coarse cache set granularity 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 granularity is sufficient in many cases [OST06, RTSS09, ZJRR12, IES15a, OKSK15, LGS+16, MIE17].
Again there are 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 threshold, parts of the eviction set were evicted by the victim process, meaning that the victim accessed cache lines congruent to the eviction set [LYG+15]. Unlike F+R, E+R, and F+F, P+P does not rely on shared memory. However, the granularity is more coarse-grained, noisier, works only if the victim is located on the same socket as the attacker, and relies on inclusive caches. In non-inclusive cache scenarios, an attacker again has to focus on the directory structure rather than the cache itself [YSG+19].

2.3.3 Other Micro-architectural Attacks

Lipp et al. [LSG+18] revealed in 2018 how to use the cache as a covert channel to read arbitrary kernel space memory from userspace and Kocher et al. [KGG+18] used a similar technique to exfiltrate arbitrary data from a victim process. Both attacks rely on the out-of-order execution built into many high-end CPUs to speed up computations by speculatively executing branches and un-doing the computation if the speculation was faulty. As the un-doing only affects the registers within the CPU but not the caches, speculative executions leave traces in the cache an attacker can then extract by timing memory accesses. The latter attack works cross-core and cross-VM and therefore, poses a great threat to cloud environments where multiple users co-reside on the same CPU package. Van Schaik et al. presented indirect cache attacks. They did so by timing address translation operations performed by the MMU [vSGBR18]. Gras et al. [GRBG18]

developed TLBleed, an SCA exploiting the translation look-aside buffers (TLB) instead of the cache. Since then multiple other buffers were used to run different attacks just recently [CGG+19, vSMO+19, SLM+19].

2.3.4 Countermeasures

Countermeasures are divided into those that are realizable in software and those that use hardware support. For countermeasures realized in software, three main approaches are known in the literature. One is to partition the cache by cache ways and assigning processes to only one partition. Thereby, processes cannot evict other processes cache lines anymore. The second approach is similar to the first one as it partitions the cache by cache sets. The third approach uses flags to mark cache lines as "not evictable" to disable cache line evictions. Different specific countermeasure frameworks proposed in the literature usually combine two of the approaches [KPMR12, YWCL14, ZRZ16, KLA+18].
Countermeasures (partially) realized in hardware follow similar approaches as the ones implemented in software and include the usage of hardware transactional memory [GLS+17] or the Intel Cache Allocation Technology (CAT) [LGY+16]. On ARM cores, a hardware feature called AutoLock blocks evictions from inclusive caches if at least one of the included caches still hold the cache line to be evicted [GRLZ+17].

3 Intel Arria 10 Ecosystem and Setup

Before starting with our findings, we need to explain the platforms we are working on and introduce additional terminology used in the following. We describe the OPAE Application Programming Interface (API) and CCI-P used for FPGA-CPU communication. Also, we give detailed information about hardware and software versions we used to increase reproducibility.

3.1 FPGA Platforms

Intel's portfolio lists different FPGA cloud platforms to fit a variety of needs. One of them is the *Intel Programmable Acceleration Card (PAC)*. It is a PCI Express (PCIe) expansion card containing an Arria 10 GX FPGA, 8 GB DDR4 random access memory (RAM) and 128 MB flash memory. The FPGA is connected with the host CPU and main memory over one PCIe Gen3x8 bus (cf. Figure 3.1) [Int18b]. In addition, the FPGA has direct access to one 40 GbE Enhanced Quad Small Form-factor Pluggable (QSFP+) port that can operate in either 1x40 or 4x10 Gigabit mode. Physical addresses are used to access the host memory. If the PAC is attached to virtual environments, the Input/Output Memory Management Unit (IOMMU) of the CPU translates the physical addresses (I/O Virtual Addresses (IOVAs)) of the virtual environment to physical addresses of the host [Int18a].
The *Intel Xeon Processor with integrated Arria 10 FPGA (BDX)* is a platform that combines a server CPU and an Arria 10 GX FPGA in one package. They are interconnected by one UltraPath Interconnect (UPI) and two PCIe Gen3x8 links (cf. Figure 3.1) [Huf18]. The FPGA has a 128 KiB directly mapped cache that is kept coherent with the CPU caches and accessible only over UPI. Just like on the PAC, memory accesses from the FPGA over PCIe and UPI use I/O virtual addressing when attached to a virtual environment and physical addresses otherwise. As the UPI bus bypasses the IOMMU of the CPU, the FPGA Interface Unit (FIU) implements an IOMMU and TLB in the Blue Region to translate addresses for memory requests over UPI [Int18a].

3.2 Intel Acceleration Stack

The Intel Acceleration Stack (IAS) is a framework of drivers, Application Programming Interface (API), and intellectual property (IP) developed by Intel to simplify the cooperation

of Acceleration Functions (AFs) implemented on Arria 10 FPGAs and software applications. Drivers, API and part of the IP contained in the IAS are developed open-source as part of the Open Programmable Acceleration Engine (OPAE).
On the software side, the IAS exposes an API to software developers and Linux drivers are provided for Xeon CPUs to manage the FPGA and route communication between the CPU and the FPGA.
The FPGA is sub-divided into the so-called *Blue Region* that is static at runtime and the *Green Region* that is reconfigurable. The Blue Region contains the FIU and the FPGA Interface Manager (FIM). The FIU manages the interfaces towards the CPU and the Core Cache Interface (CCI-P) towards the Acceleration Functional Unit (AFU)(s) on the FPGA, and the FIM, that monitors the FPGA's state like temperature and power consumption. Therefore, the Blue Region can be thought of as the "OS" of the FPGA. The bitstream used to configure the Blue Region is consequently called *bluestream*.
The Green Region contains the Partially Reconfigurable Unit (PRU) that is reconfigurable at runtime. The PRU enables a software application to dynamically exchange AFUs as needed during computation. In the future, the number of AFUs running inside the Green Region is supposed to be limited only by the space available on the FPGA; currently, only one AFU per Green Region is supported. A bitstream used to configure the Green Region is called *Green Stream*.
An abstract overview of the setup is given in Figure 3.1. We give more details on OPAE and CCI-P in the following.

3.2.1 Open Programmable Acceleration Engine (OPAE)

The Open Programmable Acceleration Engine (OPAE) is a software framework for managing and accessing FPGAs [Int17b]. It consists of an OPAE Software Development Kit (SDK), Linux drivers for both, the PAC and BDX platform, and the Basic Building Block (BBB) library to ease AFU development. The OPAE SDK and the BBB library are developed open-source on Github[1]. Software developers are given access to the AFU Simulation Environment (ASE) as well as a neatly arranged C, C++, and Python API as part of the SDK. The ASE is used for end-to-end simulation of an AF design cooperating with a software application using the C API. The API provides two main methods for exchanging data between the CPU and the FPGA: *Memory-mapped I/O (MMIO)* and *DMA*.

Memory-mapped I/O (MMIO) Software applications can send MMIO requests of 64-bit (and optionally 32-bit) width to the AFU. To do so, the C-API provides functions to map,

[1] https://github.com/OPAE/opae-sdk and https://github.com/OPAE/intel-fpga-bbb

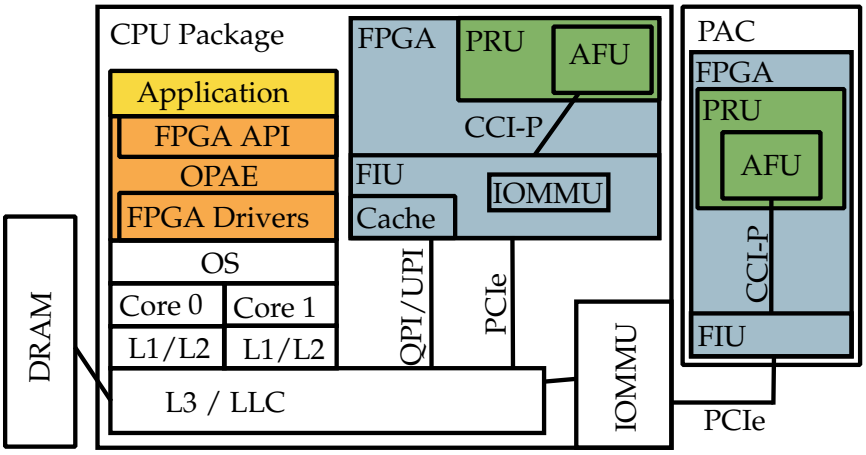


Figure 3.1: Architectural and hardware overview of Intel Arria 10-based cloud platforms. The software part of the IAS called OPAE is highlighted in orange. Its API is used by applications (yellow) to communicate with the AFU. The Green Region marks the part of the FPGA that is reconfigurable from user-space at runtime. The Blue Region describes the static soft core of the FPGA. It exposes the CCI-P interface to the AFU.

unmap, read and write MMIO memory spaces exposed by an AFU [Int17b]. The MMIO spaces available on an AFU are defined by the accelerator designer and must be known to the software developer.

For our experiments, we use `fpgaReadMMIO64` and `fpgaWriteMMIO64` API calls to configure the AFU and monitor its state. MMIO addresses passed to the API need to be 64-bit (32-bit) aligned, meaning that all addresses are multiples of 8 (4) for 64-bit (32-bit) MMIO reads and writes.

Direct Memory Access (DMA) An AFU can be given DMA to parts of the systems main memory. To do so, the software application first uses the API call `fpgaPrepareBuffer` to allocate a buffer and configure the IOMMU to grant the AFU access to that memory region. The AFU uses physical addresses for DMA, or IOVAs if connected to a virtual environment on a system using Intel Virtualization Technology for Directed I/O (VT-d). The address of the allocated buffer must be manually passed to the AFU (e.g. via MMIO) and `fpgaGetIOAddress` is used to the virtual-to-physical address translation. Note that user-level processes usually don't know the physical addresses because, in addition to other security concerns, they give information about the cache set the address is mapped to. This knowledge can be used to ease cache attacks like P+P. Intel is aware of the fact that making physical addresses available to userspace through OPAE is a security risk, as a note in the documentation [Int17b] of the `fpgaGetIOAddress` function shows. That is why

this API-call will be removed in the future.
Because the AFU uses physical addressing for DMA, allocated buffers must be physically contiguous in memory. To fit this need, the `fpgaPrepareBuffer` call allocates[2] hugepages if the requested buffer size exceeds a single memory page. If the software application needs to grant the AFU access to a memory region previously allocated by some other API, the `FPGA_BUF_PREALLOCATED` can be passed to `fpgaPrepareBuffer`. In this case, the software application has to ensure that the buffer is physically contiguous in memory and that its size is a non-zero multiple of the memory page size (4 KiB).
It is well known that disabling hugepages increases the barrier of finding eviction sets [IES15a, LYG+15] which in turn makes cache attacks more difficult. We suggest disabling OPAE's usage of hugepages. To do so, the AFU address space has to be virtualized, no matter whether the AFU is attached to a virtual machine or the host itself.

3.2.2 Core Cache Interface (CCI-P)

The Core Cache Interface (CCI-P) [Int18a] is part of the BBB library [Int17a] and can be understood as the "API" for accelerator designers. It provides a simple and hardware-independent way of handling MMIO and DMA requests. Therefore, AFUs designed to work with CCI-P can be synthesized for any CCI-P compatible platform without the need for changing the design.
To enable the OPAE Linux drivers to identify an AFU, the AFU implements the "Device Feature Header" (DFH). It contains a unique identifier for the AFU, the supported CCI-P version number, information about the functionality provided, and the MMIO space mapping. On the accelerator side, MMIO requests are always processed in blocks of 64 (or 32) bits. That is why the two least-significant MMIO address bits are truncated. CCI-P defines 16 address bits for the MMIO space, so its size is 2^{16} 32-bit words, or 256 KiB. It is left to the AFU designer to decide in what order and speed incoming MMIO requests are processed. However, their answer must be sent within 65536 cycles of the primary FPGA clock.
An AFU cannot send MMIO requests towards the software application. If the accelerator needs to send data to its software counterpart, it can do so via DMA. CCI-P offers memory reads, writes, and write fences; each read/write request can process either one, two, or four cache lines. Memory addresses processed by the CCI-P are 46 bits wide as the 6 least-significant intra cache line offset bits are truncated[3]. An AFU can issue DMA requests to a specific bus (UPI or PCIe) or leave the choice to the FIU. The latter is recommended

[2] https://github.com/OPAE/opae-sdk/blob/1.1.2-PAC/libopae/src/buffer.c#L73
[3] On `x86_64` architectures, only 48 bits of the 64-bit physical addresses are used. Bits 63-47 are either 1 or 0 indicating kernel and userspace addresses, respectively.

Table 3.1: Overview of the caching hints configurable over CCI-P on an integrated Arria 10. `*_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.

	DMA Read		DMA Write		
Hint	`RdLine_I`	`RdLine_S`	`WrLine_I`	`WrLine_M`	`WrPush_I`
Description	No FPGA caching intent	Intent to cache in FPGA cache in shared (S) state	No FPGA caching intent	Intent to cache in FPGA cache in modified (M) state	Intent to cache in LLC
Available	UPI, PCIe	UPI	UPI, PCIe	UPI	UPI, PCIe

as it leaves the AFU design hardware-independent and auto-balances workloads on the different buses available.
DMA requests driven over UPI are first processed by the local cache of the FPGA. On a cache miss, the requests are forwarded to the LLC of the CPU before it eventually reaches the main memory, if another cache miss occurred. Requests over PCIe are directly passed to the CPU's LLC, therefore bypassing a potentially available local cache on the FPGA. Again, on a miss, the requests is eventually answered by the main memory. CCI-P marks every DMA response with a cache hit/miss bit, indicating whether the local cache of the FPGA returned the requested data or not. On a PAC, this indicator is always '0' because it does not have a local cache. The same is true for requests sent over PCIe as it bypasses any cache provided by the Blue Region.
If the corresponding FPGA features a local cache, the AFU has control over what data is to be cached locally when doing DMA by adding caching hints to the requests. Available hints are summarized in Table 3.1. 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 not cache data locally, `WrLine_M` leaves written data in the local cache in the modified state, and `WrPush_I` does not cache data locally but requests to cache data in the CPU's LLC.
The CCI-P documentation [Int18a] lists all caching hints as available for memory requests over UPI. When running requests over PCIe, `RdLine_I`, `WrLine_I`, and `WrPush_I` are supported while all other hints are ignored.

3.3 Experimentation Platforms

We analyze two distinct FPGA-CPU platforms with Intel Arria 10 FPGA:

1. CPU and FPGA integrated into one package (BDX) and

2. Programmable Acceleration Card (PAC).

Access to prototype versions of the platforms is kindly provided by Intel through donation and their Intel Lab (IL) Academic Compute Environment[4]. The IL gives academics access to Intel technology to support and advance research. Besides the FPGA platforms named above, access to Xeon, Xeon Phi, and Stratix 10 systems is provided and artificial intelligence frameworks such as Caffee and TensorFlow come pre-installed and optimized for the usage with Intel products.

BDX Platform The BDX platform is a prototype based on an E5-2600v4 CPU. The IL environment gives access to two different versions with either 12 or 14 physical cores, both being clocked at 1.2 - 3.4 GHz. The CPU has a Broadwell architecture in which the LLC is inclusive of the L1/L2 caches. The CPU's LLC has $l = 12$ slices, each holding 2048 sets ($s = 11$) with an associativity of $w = 20$ cache lines á 64 bytes ($b = 6$). For the 12-core CPU, this results in a total size of 30720 KiB for the L3 and on the 14-core CPU, the L3 can hold 35840 KiB (2.56 MiB per core). Both CPUs have one 256 KiB L2 and two 32 KiB L1s, one for data and one for instructions, per core.
The integrated Arria 10 GX 10AX115U3F45E2SGE3 FPGA is running at 400 MHz. The Blue Region contains a 128 KiB direct-mapped cache. Intel's proprietary QuickPath Interconnect (QPI), instead of the announced UPI, coherently connects the FPGA with the processor and the main memory. QPI was chosen over UPI, as this is the standard interconnect for processors of the Broadwell architecture. Additionally, two PCIe buses connect the FPGA with the main memory through the LLC. However, they bypass the cache of the FPGA.
The OS is a 64-bit Enterprise Linux 7 with Kernel version 3.10. It allows access to the `/proc/self/pagemap` file, so we know the physical addresses used by our applications. Also, the OS has 4096 2 MiB-hugepages enabled to meet the needs of the OPAE memory allocation. The OPAE version that is installed for usage with the BDX platform was compiled and installed on July 15th, 2019 and Board Management Controller (BMC) firmware version 5.0.3 manages the integrated FPGA. Hardware design synthesis and the placing and routing on the FPGA is done by Quartus 16.0.0 and takes 60 to 90 minutes on average. This is even true for small designs because all of the Green Region design is re-synthesized, placed and routed against the Blue Region every time.
Unless noted otherwise, we run our experiments on one of the BDX platforms. That is because it has the same and more connection options as a PAC and an extra cache. Also, the cache of the CPU on a BDX platform has an inclusive LLC, which makes common tech-

[4] https://wiki.intel-research.net/

niques for finding eviction sets applicable. Constructing eviction sets for non-inclusive caches is possible [YSG+19], but not researched in the same dimension yet.

PAC Platform The IL environment provides access to platforms with two PACs. The FPGA on each PAC is an Arria 10 GX 10AX115N2F40E2LG FPGA running at 200 MHz[5]. Both PACs are connected to each other via the QSFP+ ports to facilitate research on networked FPGAs.

The PAC platforms are driven by two Intel Xeon Platinum 8180 processors with 28 physical cores and clocked at 1.0 - 3.8 GHz. The CPUs have separate 32 KiB L1 caches for data and instructions, and one L2 of size 1024 KiB per core. As a trade-of for the increased size of the L2, the size of the L3 is 39424 KiB (1.408 MiB per core). This trade-of is possible, because the L3 is non-inclusive of the L1 and L2 caches. The L3 has $l = 28$ slices, 2048 sets ($s = 11$), and an associativity of $w = 11$. One cache line is 64 bytes wide ($b = 6$).

The OS running in the IL is a 64-bit Enterprise Linux 7 with Kernel version 3.10. Like on the BDX platforms, access to the `/proc/self/pagemap` is granted and 4096 hugepages are available. The OPAE version was compiled and installed on July 15th, 2019, and the bitstream version of the BMC is 1.1.3. Synthesis and the place and route operations are done with Quartus 17.1.1 and take around 30 to 40 minutes in total. This is significantly less, compared to the BDX platform, and most likely because of the more limited number of interfaces, which makes the place and route operation less complex.

Additionally, we have root access to two PACs with the same FPGA as in the IL environment that were kindly donated by Intel. One of them is plugged into a server driven by a Xeon E5-2670 CPU (10 physical cores) at the Worcester Polytechnic Institute (WPI). The CPU runs at 1.2 - 2.5 GHz. The inclusive L3 has 25600 KiB, that are subdivided into $l = 10$ slices á 2.56 MiB. The cache layout within the slices is identical to that of the CPUs on the BDX platform ($s = 11$, $w = 20$, $b = 6$). The server has 32 GB system memory installed; we configure hugepages as needed.

The second PAC was later inserted into a server with two Xeon Silver 4114 CPUs at the Institute for IT-Security (ITS). The CPUs have 10 physical cores each that are clocked at 2.20 GHz. The L3 is non-inclusive of the L1/L2 caches and 14080 KiB in size; that is 1.408 MiB per slice. Each of the $l = 10$ slice's layout is identical to that on the Platinum 8180 CPUs in the IL ($s = 11$, $w = 11$, $b = 6$). The server has 96 GB of RAM installed and the number of available hugepages can be configured as needed.

Ubuntu with Kernel version 4.4 is running both servers. To complete the cloud scenario, we installed IAS version 1.2, which includes OPAE 1.1.2-1 and Quartus 17.1.1. The Blue

[5]The PAC is intended to support 400 MHz clock speed, but the current version of the IAS has a bug that halves the clock speed.

Region on the FPGAs contains BMC firmware version 0x7F8300030201.

4 CPU-FPGA Platform Investigation

We reverse-engineered parts of the memory subsystem and its behavior on current Arria 10-based FPGA-CPU cloud systems. In this chapter, we first reveal several leakages that are observable by an AFU or software application. To do so on an FPGA, we construct a precise timer in hardware. We continue by deeply analyzing the caching hint behavior to find ways for an AFU to evict cache lines from the LLC. We combine our findings to construct a covert channel with a bandwidth of 94.98 kBit/s from an AFU to a co-operating software process in the last section of this chapter.

4.1 Designing a Hardware Timer

To measure memory access latency from the FPGA, we designed a timer module that exposes the interface defined in Listing 4.1. Besides a clock, the timer is configurable via a one-bit input flag `hpc_run` that enables the timer while driven high and disables it otherwise. When disabled, the timer writes the measured clock cycles to the `hpc_value` output register before resetting the counter. The `hpc_done` flag indicates the availability of the measured value in the `hpc_value` register. In fact, the counter value in `hpc_value` is only guaranteed to contain meaningful data while `hpc_done` is high, as the output register is not initialized.

Listing 4.1: Interface definition of our timer module.

```
1 module high_performance_counter
2 (
3         input              clk,
4         input              hpc_run,
5         output reg         hpc_done,
6         output reg [63:0] hpc_value
7 );
```

Internally, the timer implements a state machine with the three states Idle, Running, and Done; with Idle being the initial state. While `hpc_run` is driven high, the state changes to Running. When `hpc_run` goes low, the state machine switches to the Done state for one clock cycle, before returning to Idle. As long as the timer is in state Running, a 64-bit counter register gets incremented with every clock cycle. When the module changes to Done, the counter value is written to the `hpc_value` register and the `hpc_done` flag is driven

high. The counter register is reset to zero whenever the timer is in state Idle.
In all our AFU designs, we will connect the primary CCI-P clock to the `clk` input of the timer as it is the fastest clock available. The advantage of a timer realized in hardware is, that it runs un-interruptible in parallel to all other sub-modules of the AFU and no signals to start or stop a timer on the CPU need to be sent. Therefore, the timer precisely counts FPGA clock cycles, while timers on the CPU, such as `rdtsc`, may yield more noisy measurements due to interruptions by the OS, the CPU's out-of-order pipeline, or delays introduced by the QPI or PCIe bus. However, we keep using `rdtsc` for timing measurements on the CPU. This is because its usage is easy and again we do not need to send messages between CPU and FPGA. Also, the hardware timer only yields measurements with a resolution of 200 or 400 MHz. This is not enough in the context of CPUs running about 7.5 times faster.

4.2 Timing Leakages Visible on a PAC

The CCI-P documentation [Int18a] mentions a measurable timing difference on the PAC for memory requests served by the CPU's LLC and those served by the main memory. We designed an AFU capable of issuing memory read requests and timing their latency with our hardware timer. The AFU exposes a simple interface to enable memory address configuration and measurement transmission to MMIO space. A software application prepared a memory buffer to be accessible by the AFU. It then uses the `clflush` and `mov` instructions to locate the first cache line of the buffer either in the main memory or the cache hierarchy. We executed our experiment on the WPI PAC, which has an inclusive LLC. Therefore, a cache line is present in the LLC after reading it using the `mov` instruction. After the cache line is in place, the software application starts the AFU, which then reads the first cache line of the prepared buffer and returns the measured latency. The access time distribution is shown in Figure 4.1. The peaks are close but distinct, meaning that an AFU attacker can tell whether an address was cached in the LLC or not by measuring the access latency.
As we know from subsection 2.3.2, a way to distinguish the locations in the memory hierarchy is required in order to run cache attacks. Additionally, a way of altering the shared cache either by flushing or evicting is needed. While memory reads on a CPU always result in the data being cached, we figured that this is not the case with memory reads originating from the PAC. This fact reduces the attack surface for cache attacks from a PAC; but at the same time is a great advantage for rowhammer attacks as memory will stay un-cached on consecutive reads [WTM+19].
As we shall see in section 4.5 in greater detail, the PAC can alter the LLC by memory

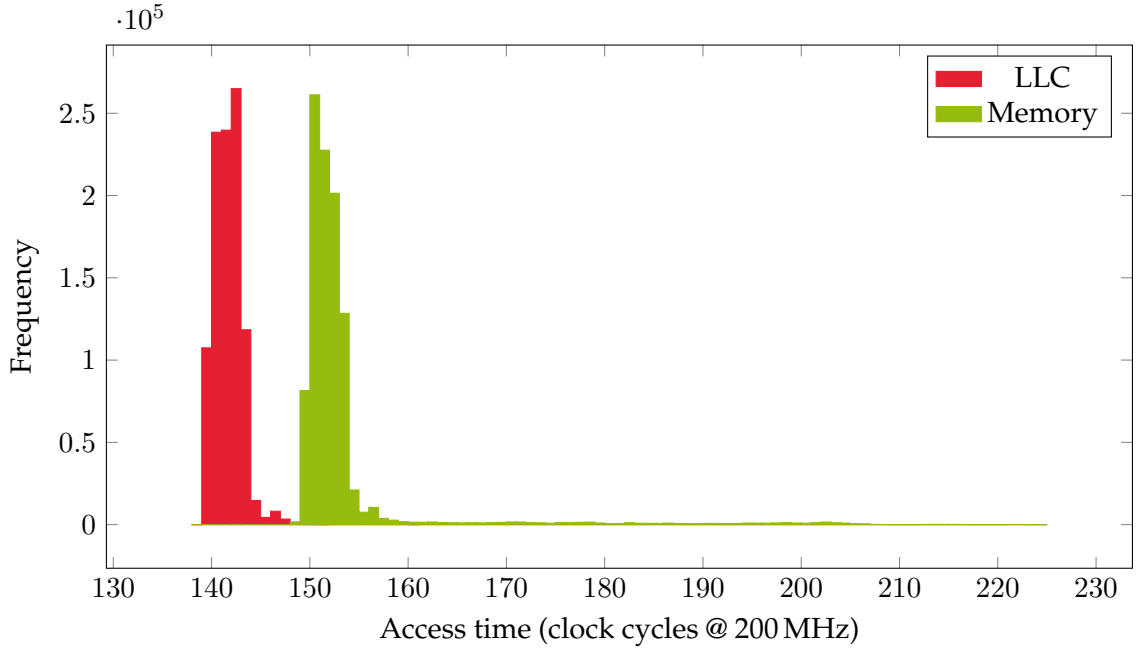


Figure 4.1: Data access time distribution measured on a PAC for data being present or absent in the CPU LLC.

writes. Therefore, the potential risk of cache attacks still emanates from PACs.

4.3 Timing Leakages Visible on an Integrated Arria 10

For FPGA-based cache attacks against the CPU, the integrated Arria 10 platform is much more promising as it has a local cache that is coherent with the CPU's LLC. We started by verifying the memory location-dependent latency differences proposed in [Int18a]. As we wanted to know the latency of the local cache of the FPGA as well, we expanded the AFU used in section 4.2 by the option to configure the caching hint sent with memory read requests. Also, we made the bus used to send the requests configurable. Making the AFU configurable from software drastically reduces the time needed for synthesis, place, and route operations. Then, we ran our experiment for each PCIe and QPI bus separately.
We found, that read requests sent over one of the two PCIe lanes shows an access time distribution similar to the one observed on the PAC (recall Figure 4.1). Also, issuing memory requests over one of the PCIe lanes again does not alter the state of the cache line being read, neither in the local cache of the FPGA nor in the LLC. So, we assume the integrated Arria 10 FPGA to behave the same way a PAC does when using the PCIe lanes. As we later learned from an Intel employee, the Blue Stream designed for PACs is derived from the Blue Stream used on the BDX platform. This fact strengthens our assumption.

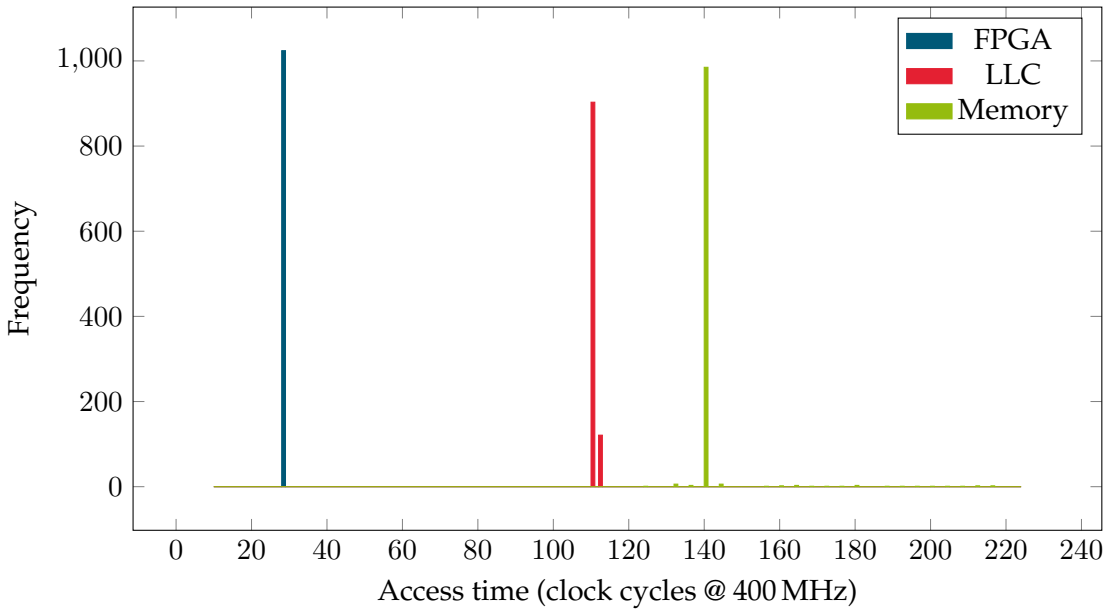


Figure 4.2: Memory access latency on the integrated Arria 10 when loading data from the FPGA local cache, CPU LLC, or main memory.

When sending memory requests over the QPI link, the FPGA's local cache is queried first, before sending the request off to the CPU. In this case, the memory access latency distribution shows three distinct peaks, as depicted in Figure 4.2.
Issuing memory reads over QPI does not measurably alter the LLC. Memory reads cached in the FPGA cache do not result in shorter latency on the CPU, as we see in section 4.4. Instead, the latency goes up compared to a read from the main memory. This is confusing as inclusive caches like the LLC on the Broadwell CPU usually hold every cache line that is present in any cache it is inclusive of. This means that the LLC either does something undocumented, or it is non-inclusive of the FPGA cache. In the latter case, it is unclear, how cache coherence is maintained.
Again, the possibilities of writing to the LLC are discussed in section 4.5.

4.4 Timing Leakages Introduced by an Integrated Arria 10

This section investigates the CPU's capabilities to run cache attacks against the coherent cache on the integrated Arria 10 FPGA. Again, we measured the memory access latency depending on the location of the address accessed. The measurements were gathered by using the AFU and software application developed during section 4.3 We got addresses placed in the right locations (memory, last level cache, FPGA local cache) before measuring the access time, this time from the CPU using the `rdtsc` instruction. The results are

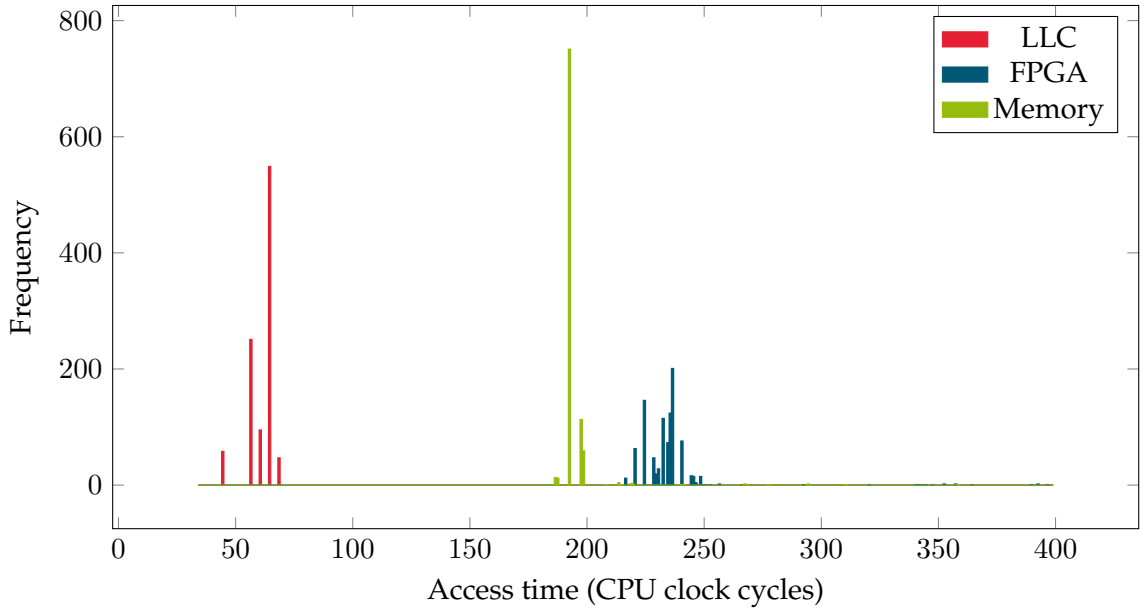


Figure 4.3: Memory access latency distribution on the Broadwell CPU with data being present in FPGA local cache, CPU LLC, or main memory.

presented in Figure 4.3.

They show, that the CPU can clearly distinguish where a cache line was located. As all known cache attacks have a probing phase, this capability is a good step in the direction of having a fully working cache attack from the CPU against the FPGA cache. It is interesting to note, that answers from the main memory return faster than those coming from the FPGA cache. This may be explained by the slow clock speed of the FPGA (400 MHz) compared to the CPU clock speed (1.2 - 3.4 GHz). Also, the platform is an early prototype so the coherency protocol implementation of the Blue Region might still be buggy.

Besides the capability of probing the FPGA cache, we also need a way of flushing, priming, or evicting cache lines. While the AFU can control which data is cached on the FPGA, there is no such option documented for the CPU. Therefore, the CPU cannot directly prime the FPGA cache and use this to evict cache lines. As the CPU has a `clfush` instruction, it can flush a cache line from the FPGA cache. That is, because the cache is coherent with the CPU caches meaning that flushes from the LLC result in flushes from the FPGA cache.

In total, we have the capability of probing and flushing cache lines located in the FPGA cache. So the CPU attacker may run a “Flush+Probe” attack against the victim AFU where the addresses used by the AFU get flushed before the execution of the AFU. Afterward, the attacker probes all previously flushed addresses to learn, which addresses were used during the AFU execution.

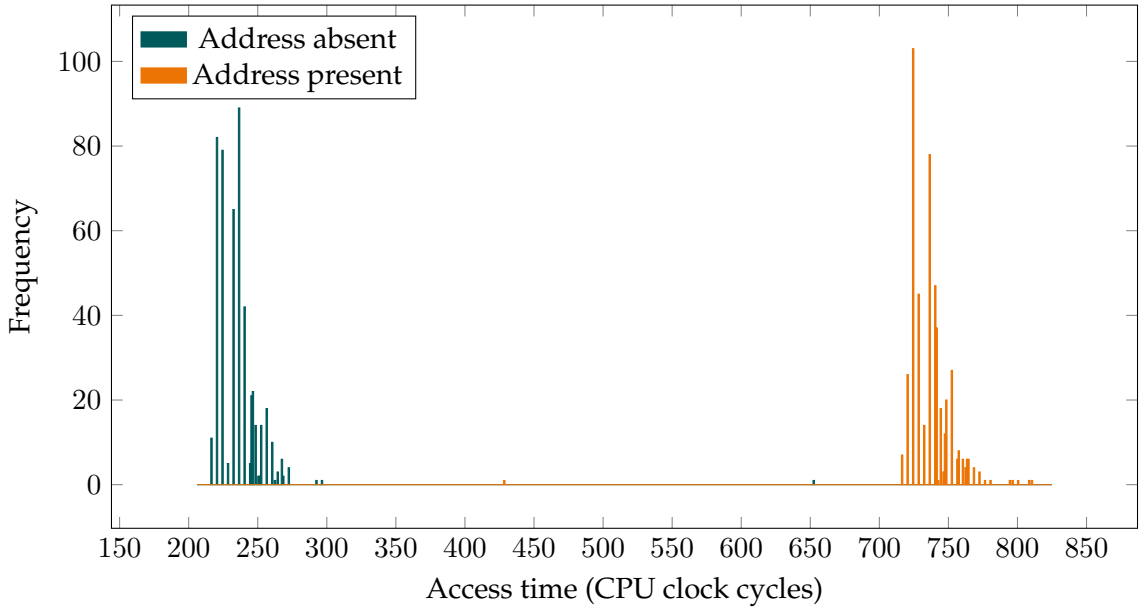


Figure 4.4: The `clflush` instruction execution time on the Broadwell CPU depends on whether the flushed address is absent and present in the cache of the integrated Arria 10.

Another possible cache attack is F+F (recall section 2.3.2). We assume this attack to be more efficient as the probing phase already executes the next flush phase. 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. Figure 4.4), while the latency difference between memory and FPGA cache accesses adds up to only about 50 - 70 CPU clock cycles.
A rather unrealistic scenario for the F+R and F+F attacks is a cloud service provider spying on an AFU. More realistic is a scenario where a spying process and the AFU share data, e.g. an AES T-table. Then, the CPU would flush entries of the table to main memory, let the AFU compute encryption or decryption, and then flush the entries again. Cache lines resulting in high flush execution times were cached in the FPGA cache. This means, that they were used during the encryption/decryption. An attacker can now use that knowledge to mount a cache-collision timing attack and recover the secret encryption/decryption key [IİES14].

4.5 Reverse-engineering Caching Hint Behavior

We reverse-engineered the behavior of the caching hints. In our experiments, the memory read caching hints `RdLine_I` and `RdLine_S` show no useful effect on the LLC, neither

over PCIe, nor over QPI. The `RdLine_S` flag asks the Blue Region to cache the cache line in the FPGA's local cache if the request is sent over QPI. For requests over PCIe, this caching hint is ignored. Setting `RdLine_I` in a memory request over QPI or PCIe requests an uncacheable snapshot of a cache line. If the cache line requested lies in the LLC in the modified state, this request results in a write-back to the main memory and eviction from LLC before forwarding the cache line to the AFU. In common attack scenarios, data is either shared by two entities or writable but usually not both at the same time. Therefore, the situation of evicting a modified cache line from the LLC only occurs for cache lines owned and used by the AFU and its corresponding software application and cannot be exploited to mount cache attacks.

When writing data, three caching hints are available over QPI. The `WrLine_M` caching hint caches the cache line before writing to it, leaving the cache line in the local cache in a modified state. The `WrLine_I` flag behaves as a write-back flag, evicting the cache line from the local cache to the main memory. `WrPush_I` writes to a cache line, evicts it from the FPGA cache, and hints the CPU to store the cache line in the LLC. Because the FPGA's cache is kept coherent with the CPU's LLC, writing to a cache line from the AFU must result in invalidating the cache line in the LLC or updating it. For the `WrLine_M` caching hint, the cache line must be evicted from the LLC as the cache line is left in the FPGA cache in a modified state. In the case of `WrLine_I` either invalidating or updating may be true; the CCI-P documentation [Int18a] explains that invalidating is what happens. When `WrPush_I` is used, we expect the cache line to be updated in the LLC at least every once in a while.

To validate our assumptions and the documentation of the write behavior, we timed the CPU's access to cache lines that were previously written to by an AFU. So, we designed an AFU that can write to memory using a caching hint and communication bus configurable via MMIO. The corresponding software application configures the AFU, activates it, and afterward measures the access time for each cache line that was just written.

On the BDX platform, we tested all bus/hint combinations including the combinations that are supposed to be ignored. The resulting CPU access times are plotted in Figure 4.5. We can see, that cache lines written to over PCIe as well as over QPI with `WrLine_M` always end up in the LLC, as their access time is strictly below 100 CPU clock cycles. For most cache lines written to over QPI using flag `WrLine_I` or `WrPush_I` this is true as well. But we see a small number of cache lines that return after around 225 clock cycles. Those cache lines come from the main memory, as we see by including the distribution of simple memory accesses at the bottom of Figure 4.5.

On the PAC, we see a very similar distribution; the concrete result is shown in Figure 4.6. We again measured the `WrLine_M` flag, even though it is ignored if used over PCIe. Most

of the cache lines were present in the LLC after the write request (high peak around 100 cycles) and only a very small portion of the cache lines were written back to the main memory (small peak around 300 cycles). The small peak is observable on the BDX platform as well, but can hardly be seen in Figure 4.5 because of the scale that was needed to have all measurements fit the page.
For the `WrPush_I` flag, the observed distribution fits our expectation because only some cache lines end up in the main memory. Since `WrPush_I` is just a hint to the CPU, the homing agent in charge may either forward the cache line to some LLC cache agent or write it back to memory. Though, it is interesting to note that the chance for cache lines written over PCIe to be forwarded to the LLC is very close to 100%.
The results for the `WrLine_M` over QPI[6] hint are interesting because all access times are below 100 cycles while the read latency from the FPGA cache measured in section 4.4 distributed around 230 cycles. We re-ran both experiments right after each other and got the same results again. So it seems like the coherence protocol delays cache line access for cache lines in the FPGA cache which are in the modified state. Finding the reason for the delay is left for future work.
We found unexpected behavior when using the `WrLine_I` flag. It is supposed to write the cache line back to the main memory, independent of whether it is used over QPI or PCIe [Int18a]. But instead, we see LLC hits in the great majority of measurements.
We asked in a related Intel forum for implementation details concerning the caching hints in the IL environment and learned that the PAC, and most likely the BDX platform as well, ignore caching hints in write requests. The CPU, however, handles all write requests as if `WrPush_I` is set.
This information explains most of the behavior observed, but leaves the question, why `WrLine_I` and `WrPush_I` writes over QPI end up in the main memory significantly more often than all other writes.
As the CPU decides to cache (nearly) all write requests in the LLC, we assume a DDIO like behavior which gives the AFU access to a reduced number of ways per cache set. This would reduce the attack surface for a P+P attack as only parts of the cache sets can be primed. However, attacks against other peripherals are still possible as recent results [KGA+20] show.

4.6 Constructing a Covert Channel from Integrated Arria 10 to CPU

We complete our work by realizing the promised covert channel. To do so, we designed an AFU that takes a message `i_data` and a memory address as input. It then writes the

[6] `WrLine_M` is ignored by PCIe requests and `WrLine_I` is used instead [Int18a].

constant string "Hello world" to the memory address over PCIe whenever a '1' occurs in the message and stays quiet when a '0' is to be sent. This way, the AFU encodes the configured message in a memory write pattern which can be read by any software process running on the CPU monitoring the cache set the configured memory address maps to. For the rest of this section, we will refer to the memory address the AFU writes to as the *target address*. Because we did not know in advance whether the AFU or the receiver will be the bottleneck of the transmission, we added an interface to the AFU that takes a number `i_stuf_bits` and waits the corresponding amount of clock cycles between each bit transmission. This is also useful to empirically determine the time to wait between two memory writes that is needed to not fill the CCI-P output buffer. Last but not least, the AFU can be configured to send each bit of the message several times (`i_bit_rep`) to ease message decoding through redundancy. The module header definition is shown in Listing 4.2 and the complete core logic of the AFU creating the write pattern can be read in Listing 1 of the Verilog Code appendix.

Listing 4.2: Header definition of the sending pattern generator contained in the covert channel AFU.

```
1  module covert_channel
2          #(
3                  parameter DATA_WIDTH=64,
4                  parameter CNTR_BITS=10
5          )
6          (
7                  input                   clk,
8                  input                   rst,
9                  input                   clk_en,
10                 input                   i_load,
11                 input [CNTR_BITS-1:0]   i_stuf_bits,
12                 input [CNTR_BITS-1:0]   i_bit_rep,
13                 input [DATA_WIDTH-1:0]  i_data,
14                 output logic            o_data,
15                 output logic            o_ready
16         );
```

The receiving process first constructs an eviction set for the set/slice-pair the target address belongs to. To construct an eviction set, we run a slightly modified version of Algorithm 1 using Test 1 from [VKM18]. Vila et al. construct their eviction set bottom-up, starting from an empty set. While this approach is valid in theory, it did not work for us. Instead, we take the top-down approach, reducing the size of the candidate set, until we reach an eviction set of size w[7]. In general, the candidate set is a huge set of mem-

[7] Recall, that w is the associativity of the cache.

ory addresses. This necessary, because virtual addresses that point to normal memory pages do not contain all cache set bits to learn the LLC cache set the address is mapped to. Therefore, the set initially contains addresses not belonging to the right set. They can be discarded if they do not collide with the target address.
However, we use the OPAE API to allocate buffers, which are backed by hugepages if the buffer size exceeds 4 KiB (cf. subsection 3.2.1), which we did. Therefore, we can construct the eviction set from a rather small set of candidate addresses all belonging to the same set as this means that the virtual addresses of the buffer share all those bits with the physical address that describe the corresponding cache set. But since we do not know the slice selection function of the CPUs, we still have to start with a set larger than w and then reduce it.
To ease the candidate set reduction in our case, the receiver process has access to the target address through shared memory. This way, the receiver can test its eviction set against the target address directly and we avoid the need of explicitly identifying the LLC slice the target address is mapped to. In a real-world scenario, either the slice selection function has to be known [HWH13, IES15b, İGI+15] or eviction sets for all slices have to be constructed by seeking conflicting addresses [LYG+15, OKSK15]. In the latter case, having one thread per slice monitoring the slice prevents the time penalty that would be introduced by monitoring all slices sequentially.
After finding an eviction set, the receiver primes the LLC with the eviction set and probes the set in an endless loop, basically running a P+P. At the same time, the AFU starts sending the configured message by issuing write requests with the corresponding pattern. Note that the AFU must not start sending as long as the eviction set finding step is in progress. With every write, the target address is cached in the LLC, making the eviction set finding algorithm believe that the eviction capability is not given. This leads to a failure of the eviction finding step.
Whenever the execution time of a probe is above a threshold, the receiver assumes the (partial) eviction of the eviction set addresses being the result of the AFU writing to the target address. If three evictions are measured within a fixed time window, the receiver interprets this as receiving a '1'. If the probe execution time stays below the threshold three times in a row, a '0' is detected as no eviction of the eviction set addresses occurred. An example of transmission measurement and its decoding steps are depicted in Figure 4.7.
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 1.509 - 1.8329 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 [WTM+19]. In theory, this[8]

[8] This is a worst-case scenario where every transmitted bit is a '1'-bit. For a random message, this estimation

makes the AFU capable of sending 40 MBit/s over the covert channel when clocked at 400 MHz.
The covert channel currently keeps breaking down regularly for an undefined time after every 800 - 1100 probes. This hinders us from transmitting long and pseudo-random bit-sequences to compute reliable error rates. Knowing error rates would yield even more information about the percentage of written cache lines that are pushed to the LLC as every two to three cache lines that are written back to the main memory will lead to a wrong bit being detected on the receiving end. The reason for the breakdown is still unknown to us. We assumed a change in the way of the CPU handling writes after a certain number of memory-writes. So we let the experiment described in section 4.5 run for a long time to see whether the number of cache lines written back to the main memory increases more than linearly. However, we did not see such behavior. Also, we noticed that sometimes our eviction set finding algorithm would not find an eviction set for several minutes. This repeatedly occurring symptom seems to correlate to the breakdowns of the covert channel. We searched the operating system for jobs running regularly and producing many evictions, but could not find any.
For the beginning of transmission until the first channel breakdown, the channel achieves a throughput of about 94.98 kBit/s on the BDX platform using PCIe for writing. This is because the AFU sends every bit thrice to ease decoding. Also, the AFU and the receiver are not well synchronized. After improving single cache set throughput, 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 [TVT19].

goes up again as '0'-bits do not fill the buffer, allowing for faster transmission.

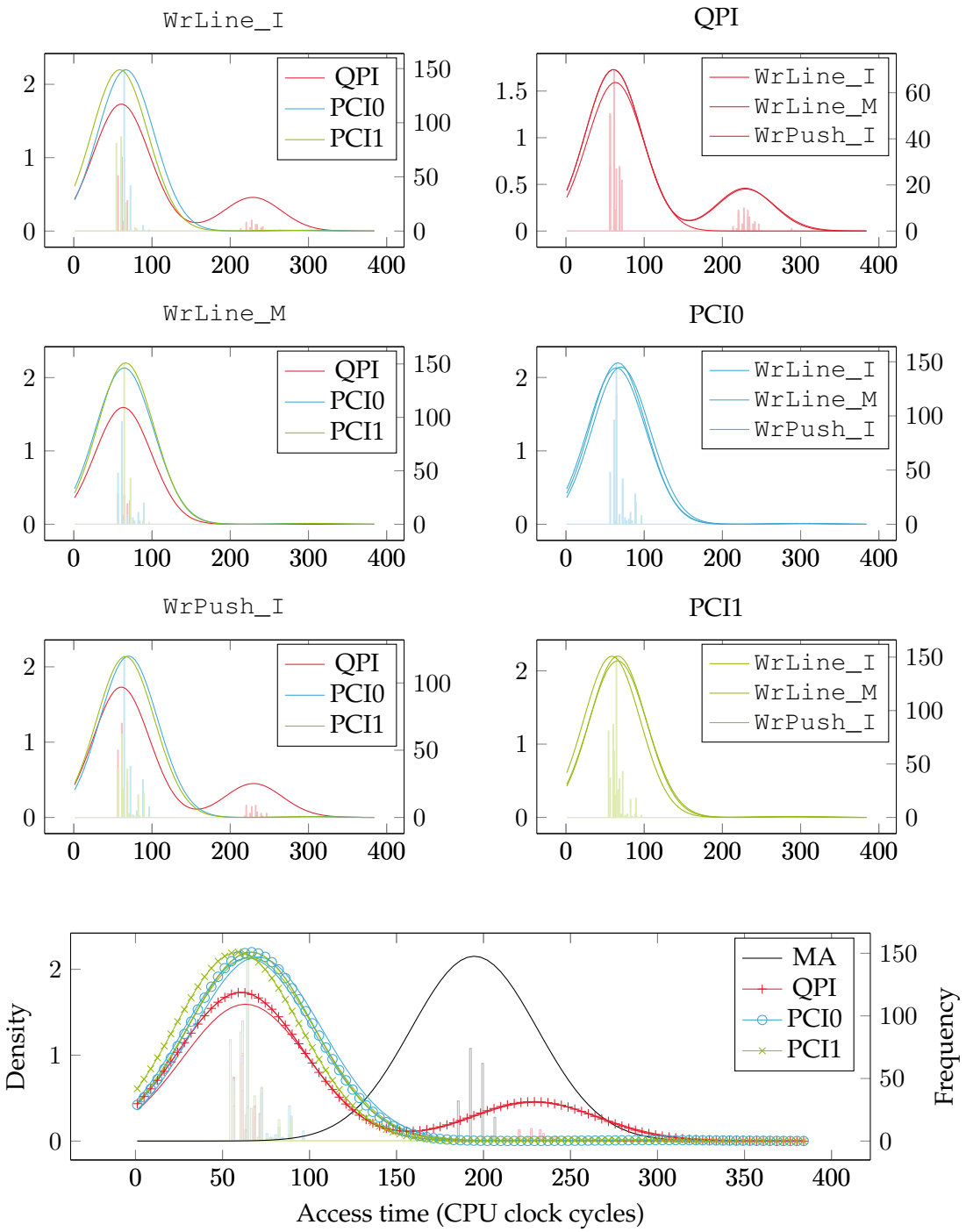


Figure 4.5: Histograms of memory access latency and their density for cache lines accessed by the CPU after being written to by an AFU running on an integrated Arria 10. Plots in the left column fix the caching hint while plots for a fixed interface are shown on the right. All measurements are combined in the bottom plot. Additionally, the distribution of simple main memory access latency (MA) is given.

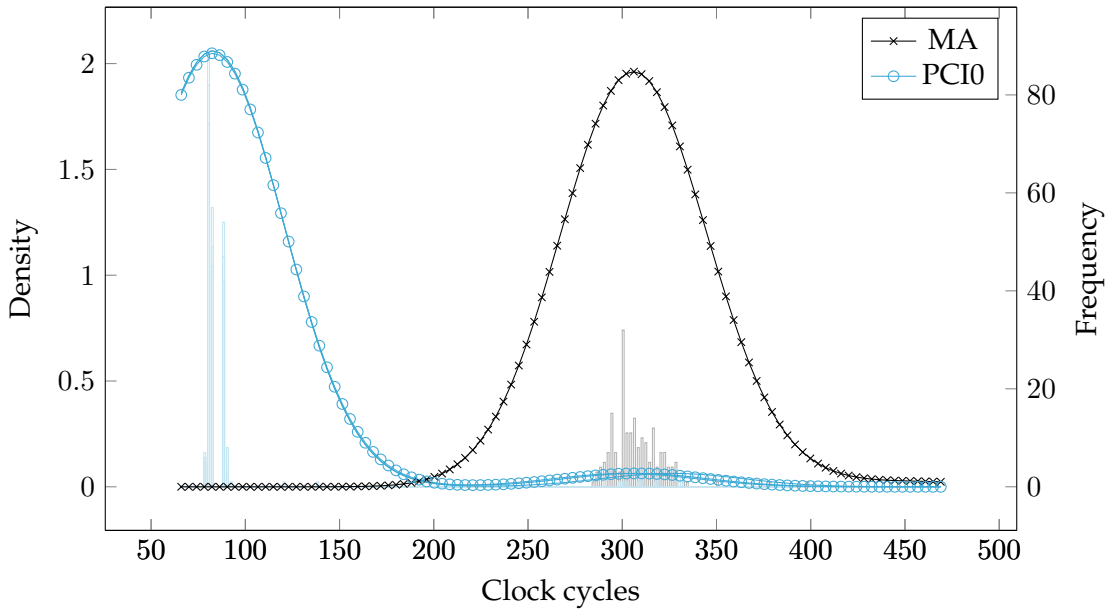


Figure 4.6: Histograms of memory access latency and their density for cache lines accessed by the CPU after being written to by an AFU running on a PAC.

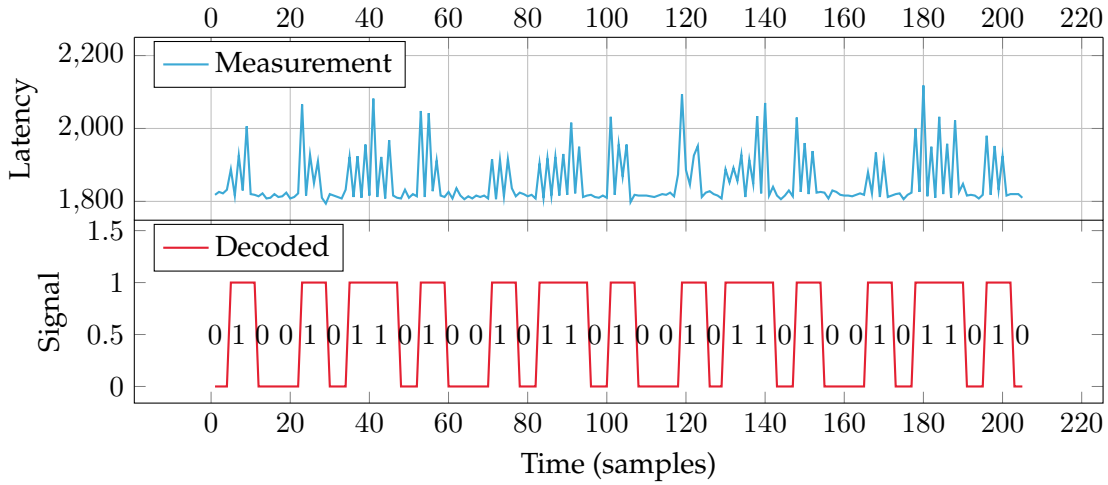


Figure 4.7: Covert channel measurements and decoding. The AFU sends every bit thrice, which results in three peaks at the receiver if a '1' is transmitted (top). Measurements are normalized by a threshold of 1875 and then classified based on a moving average every three measurements (bottom).

5 Conclusions

5.1 Summary

Bringing FPGAs to the cloud is an easy solution to meet the demand for highly specialized hardware to e.g. accelerate artificial intelligence algorithms. That is because FPGAs can be reprogrammed to realize arbitrary digital circuits, only limited by the size of the FPGA. Cloud providers offering HAaaS may protect their platform through the hypervisor. We claim that, when offering IaaS, the situation becomes more complicated as an attacker can implement any circuit on the FPGA that may help to launch an attack against the system.
To proof our claim, we investigated two prototypes of FPGA cloud platforms offered by Intel. One is the PAC that is realized as a PCI extension cart and the other integrates the FPGA into the CPU package. Our focus was on micro-architectural attacks. As these usually rely on precise timers, we started by developing a hardware timer module. Timers realized in hardware yield precise measurements with only little noise as they run uninterruptible and in parallel to any other circuit. Their resolution is bound by the primary clock speed of the FPGA, which is about 5 to 10 times slower than the CPU frequency.
Nonetheless, our timer is capable of detecting timing leakages that are introduced by the memory hierarchy. On a PAC, this gives an attacker the ability to differentiate between LLC hits and misses. On a BDX platform, the differentiation also includes the local cache on the FPGA. This basic capability is required to attack shared caches.
Another requirement is the ability to alter the shared cache. Since CCI-P does not include a flush instruction, we have to rely on evictions. CCI-P provides caching hints that can influence the coherency protocol. We investigated those hints for read and write memory accesses and found that written cache lines are pushed to the LLC, independent of the caching hint and even when the hint suggests to not do so. We learned that the FPGA currently ignores any caching hint on write requests, but the CPU is configured to handle all requests as if it was hinted to push the cache lines to the LLC. We used this circumstance to construct a covert channel from an AFU to a software application. The cover channel achieved a bandwidth of 94.98 kBits/s.
In total, we learn, that accelerators may be abused to accelerate micro-architectural attacks in the cloud, as arbitrary hardware components can be realized. A limiting factor is the lower clock frequency of the FPGA when compared to the CPU. When defining interfaces, security should be in mind, as the option for giving caching hints from user-space enables

an attacker to mess with hardware internal properties. Additionally, configurations need to be chosen carefully. A concrete example is the CPU pushing all cache lines written by an AFU to the LLC even when it is not supposed to.

5.2 Open Problems

We leave several problems for future work. For the covert channel, the reason for its breakdown needs to be further investigated. Afterward, reliable error rates could be determined by sending pseudo-random messages and computing the editing distance between the received bit sequence and the one that was sent [LYG+15, TVT19]. To generate such pseudo-random messages, a linear feedback shift register may be used as they are easily implemented in hardware.
The construction of a covert channel in the reverse direction would complete the covert channel scenario as well as demonstrating the channel on further caching hint/bus/platform combinations, as measurements from section 4.5 suggest their existence. Also, all the experiments were done only on systems with inclusive LLC should be tested on platforms with non-inclusive LLC.
In sections 4.2, 4.3, and 4.4, we only focused on the measurable timing differences for memory reads. Because we know from section 4.5 that an AFU can evict data from the LLC, being able to detect cache line locations by memory-writes could speed-up fully working end-to-end cache attacks against the LLC.
We did not discuss the number of ways per cache set an AFU can write to. It is reasonable to assume, that an AFU cannot place cache lines in all ways of a set, but is limited to a subset of ways as is the case for peripherals supporting direct cache access through DDIO. But even with limited access to the LLC, attacks against other peripherals are possible as the NetCAT paper shows [KGA+20]. Therefore, a meaningful continuation of this work would be to implement an attack like NetCAT as an AFU. Also, future work should determine the actual number of ways that are accessible by an AFU. In the unlikely case of having access to all ways per cache set, an AFU would be in the position to mount end-to-end P+P attacks against software applications running on the CPU.
When it comes to the CPU attacking an AFU, the F+F attack proposed in section 4.4 should give an easy start. Next, one could focus on the coherence protocol and search for other attack vectors that allow more realistic attack scenarios.

5.3 Future Research Topics

Just recently, the BDX platform in the IL environment was updated to always use IOVAs. This means, that the TLB contained in the Blue Region is now actually used, making it an

interesting target for the *TLBleed* [GRBG18] attack. This scenario is generalizable. Because FPGAs are highly configurable, they can be used to further investigate components like the IOMMU and corresponding buffers to find new attack vectors. The IOMMU is designed to limit the access peripherals have to the main memory. Accesses performed by the CPU, however, are not processed by the IOMMU, which makes it difficult to investigate the IOMMU from the CPU.

In 2018 and 2019, two new protocols were introduced: Cache Coherent Interconnect for Accelerators (CCIX)[9] and Compute Express Link Interconnect (CXL)[10]. Both are designed to coherently connect accelerators to a system's CPU and main memory over PCIe. Xilinx already ships Alveo Acceleration Cards with Virtex UltraScale+ FPGAs that are similar to the Intel PAC and feature coherent PCIe Gen4 connections with CCIX [Xil19]. Intel announced a new platform to compete with Alveo cards. They will be driven by Intel Agilex FPGAs and coherently connected to the main memory by the CXL [Int19]. Both, CCIX and CXL, are pushed by industry-leading companies. This circumstance makes these protocols interesting targets for future research projects.

When thinking about multi-tenant scenarios, intra-FPGA attacks come to mind. In the context of this work, research should focus on shared resources such as CCI-P. On the BDX, the cache is a shared resource and the PAC comes with onboard RAM. Some accelerators like the Alveo Cards or PACs with Stratix 10 FPGA also have private access to High Bandwidth Memory (HBM) that would be shared.

FPGA developers currently start modeling threats in the context of multi-tenancy and develop first frameworks to test. Until they are deployed to the cloud, research can focus on networked FPGAs as they are available today and are close to the scenario of having two AFUs on the same FPGA.

On top of the micro-architectural attacks, FPGAs open another interesting research field. Physical attacks in the cloud got a lot of attention just recently [GOT17, KGT18, RPD+18, SGMT18, ZS18]. Following this track and combining it with micro-architectural attacks may result in even more powerful attack classes that are unknown so far.

[9] https://www.ccixconsortium.com/
[10] https://www.computeexpresslink.org/

Glossary

CCI-P Core Cache Interface. 1, 11, 12, 15, 20, 25

OPAE Open Programmable Acceleration Engine. 1, 11, 12, 14, 20, 21, 25, 26

AES Symmetric encryption algorithm called Advanced Encryption Standard. 6, 7

AF Acceleration Function. 3, 4, 6, 7, 11, 12

AFU Acceleration Functional Unit. 1, 11, 12, 14–23, 25, 26, 28

API Application Programming Interface. 11, 12

ASE AFU Simulation Environment. 12

BBB Basic Building Block. 11, 12

Blue Region At boot configured FPGA softcore provided by Intel encapsulating the Green Region. 1, 11–13, 27

Blue Stream The bitstream that is used to configure the Blue Region on startup. 12

BMC Board Management Controller. 14

CCIX Cache Coherent Interconnect for Accelerators. 1, 25, 26

CPU Central Processing Unit. 1–5, 7, 8, 10–23, 26

CXL Compute Express Link Interconnect. 1, 25

DDIO Direct Data I/O. 15, 20

DFA Differential Fault Analysis. 2, 7

DMA Direct Memory Access. 12

DoS Denial-of-Service. 6

DPA Differential Power Analysis. 2, 6, 7

DRAM Dynamic RAM. 12

DSA Asymmetric digital signature algorithm called Digital Signature Algorithm. 5

E+R Evict+Reload. 8, 9

E+T Evict+Time. 7

EM electromagnetic. 6

F+F Flush+Flush. 7–9

F+R Flush+Reload. 7–9

FIM FPGA Interface Manager. 11, 25

FIU FPGA Interface Unit. 11–13

FPGA Field Programmable Gate Array. 1–4, 6, 7, 11–19, 25, 26, 28

Green Region The region of the FPGA that is reconfigurable at runtime. It contains one or more AFUs. 11, 12, 27, 28

Green Stream The bitstream that is used to configure the Green Region on reconfiguration. 11, 12

HAaaS Hardware Acceleration as a Service. 3, 4

HBM High Bandwidth Memory. 26

IaaS Infrastructure as a Service. 3, 4, 7

IAS Intel Acceleration Stack. 11, 12, 14

IL Intel Lab. 13, 14

IOMMU Input/Output Memory Management Unit. 12, 13

IP intellectual property. 3, 11, 14

ITS Institute for IT-Security. 14

. 22, 23

L1 Level 1 cache. 5, 7, 12, 13

L2 Level 2 cache. 5, 12, 13

L3 Level 3 cache. 7, 12

LLC last level cache. 1, 4, 5, 8, 12–21

MMIO Memory-mapped I/O. 12

MMU Memory Management Unit. 10

OS operating system. 11, 12, 14, 15

P+P Prime+Probe. 7, 9, 26

PAC Programmable Acceleration Card. 1, 11–16, 18, 23

PCI Peripheral Component Interconnect. 18–23

PCIe PCI Express. 12, 13, 26

PRU Partially Reconfigurable Unit. 11, 12

QPI QuickPath Interconnect. 12–14, 18–20, 22

RAM random access memory. 13

RSA Asymetric encryption scheme invented by Rivest, Shamir, and Adleman. 5, 7

SC side-channel. 1, 2, 5–7

SCA side-channel attack. 1, 5, 6, 10

SDK Software Development Kit. 11, 12

SoC System on Chip. 1

SPA Simple Power Analysis. 2, 6, 7

TLB translation look-aside buffers. 10, 13

UPI UltraPath Interconnect. 1, 12–14, 18

VM virtual machine. 7, 10

WPI Worcester Polytechnic Institute. 14

References

[Ali19] Alibaba Cloud. FPGA-based compute-optimized instance families. https://www.alibabacloud.com/help/doc-detail/108504.html, 2019. Access: 15-10-19.

[Ama17] Amazon Web Services. Amazon EC2 F1 instances. https://aws.amazon.com/ec2/instance-types/f1/, 2017. Access: 12-10-19.

[Ber04] Daniel J. Bernstein. Cache-timing attacks on AES. *https://cr.yp.to/antiforgery/cachetiming-20050414.pdf*, 2004.

[BFB+17] Robert G. Blankenship, Bahaa Fahim, Robert H. Beers, Yen-Cheng Liu, Vedaraman Geetha, Herbert H. Hum, and Jeff Willey. High performance interconnect coherence protocol, Apr 2017. US Patent US20170109286A1.

[BMME19] Samira Briongos, Pedro Malagón, José M. Moya, and Thomas Eisenbarth. RELOAD+REFRESH: Abusing cache replacement policies to perform stealthy cache attacks, 2019.

[CGG+19] 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 *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 769–784, New York, NY, USA, 2019. ACM.

[CSZ+14] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. Enabling FPGAs in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, pages 3:1–3:10, New York, NY, USA, 2014. ACM.

[EV12] Ken Eguro and Ramarathnam Venkatesan. FPGAs for trusted cloud computing. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 63–70, Aug 2012.

[Gib18] Samuel Gibbs. Meltdown and Spectre: 'worst' ever CPU bugs affect virtually all computers. https://www.theguardian.com/technology/

2018/jan/04/meltdown-spectre-worst-cpu-bugs-ever-found-affect-computers-intel-processors-security-flaw, 2018. Accessed: 08-01-20.

[GLS+17] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 217–233, Vancouver, BC, 2017. USENIX Association.

[GMWM16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: a fast and stealthy cache attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016.

[GOT17] Dennis R. E. Gnad, Fabian Oboril, and Mehdi Baradaran Tahoori. Voltage drop-based fault attacks on FPGAs using valid bitstreams. In *27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–7. IEEE, 2017.

[GRBG18] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating cache side-channel protections with TLB attacks. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 955–972, Baltimore, MD, 2018. USENIX Association.

[GRE17] Ilias Giechaskiel, Kasper B. Rassmussen, and Ken Eguro. A robust covert channel on FPGAs based on long wire delays. *CoRR*, 2017.

[GRE18] Ilias Giechaskiel, Kasper B. Rasmussen, and Ken Eguro. Leaky Wires: Information leakage and covert communication between FPGA long wires. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, pages 15–27, New York, NY, USA, 2018. ACM.

[GRLZ+17] 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, Aug 2017. USENIX Association.

[GSM15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating attacks on inclusive last-level caches. In *USENIX Security Symposium*, 2015.

[HMMPB18] Festus Hategekimana, Joel Mandebi Mbongue, Md Jubaer Hossain Pantho, and Christophe Bobda. Secure hardware kernels execution in CPU+FPGA heterogeneous cloud. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 182–189, Dec 2018.

[HP12] John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*, chapter B.1. Elsevier, 5 edition, 2012.

[Huf18] Jennifer Huffstetler. Intel processors and FPGAs – better together. https://itpeernetwork.intel.com/intel-processors-fpga-better-together/, 2018. Accessed: 21-05-19.

[HWH13] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205. IEEE, 2013.

[IES15a] 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*, pages 591–604, May 2015.

[IES15b] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in Intel processors. In *2015 Euromicro Conference on Digital System Design (DSD)*, pages 629–636. IEEE, 2015.

[IES16] 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. ACM.

[İGI+15] Mehmet Sinan İnci, Berk Gülmezoğlu, 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(1-15), 2015.

[İİES14] Gorka Irazoqui, Mehmet Sinan İnci, Thomas Eisenbarth, and Berk Sunar. Wait a Minute! a fast, cross-vm attack on aes. In Angelos Stavrou, Herbert Bos, and Georgios Portokalidis, editors, *Research in Attacks, Intrusions and Defenses*, pages 299–319. Springer International Publishing, 2014.

[Int17a] Intel. Intel FPGA basic building blocks (BBB). https://github.com/OPAE/intel-fpga-bbb, 2017. Access: 11-01-19.

[Int17b] Intel. *Open Programmable Acceleration Engine*, 1.3.0 edition, 2017. https://opae.github.io/.

[Int18a] Intel. *Acceleration Stack for Intel Xeon CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual*, 1.2 edition, Dec 2018. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl-ias-ccip.pdf.

[Int18b] 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.

[Int19] Intel. Intel Agilex FPGAs and SoCs. https://www.intel.com/content/www/us/en/products/programmable/fpga/agilex.html, 2019. Accessed: 21-05-19.

[KGA+20] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical cache attacks from the network. In *S&P*, May 2020. Intel Bounty Reward.

[KGG+18] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting speculative execution. *CoRR*, abs/1801.01203, 2018.

[KGT18] Jonas Krautter, Dennis R. E. Gnad, and Mehdi Baradaran Tahoori. FPGAhammer: Remote voltage fault attacks on shared FPGAs, suitable for DFA on AES. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):44–68, 2018.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 388–397. Springer Berlin Heidelberg, 1999.

[KLA+18] 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, Oct 2018.

[Koc96] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113. Springer Berlin Heidelberg, 1996.

[KPMR12] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 189–204, Bellevue, WA, 2012. USENIX.

[LGS+16] 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, 2016. USENIX Association.

[LGY+16] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pages 406–418. IEEE, 2016.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Michael Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*, 2018.

[LYG+15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, May 2015.

[MIE17] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 69–90. Springer, 2017.

[MLSN+15] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering Intel last-level cache complex addressing using performance counters. In *International Symposium on Recent Advances in Intrusion Detection*, pages 48–65. Springer, 2015.

[MRG+19] A. Theodore Markettos, 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 *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, 2 2019.

[Mul17] 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: 10-07-19.

[OKSK15] 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 *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1406–1418, New York, NY, USA, 2015. ACM.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, pages 1–20. Springer Berlin Heidelberg, 2006.

[RPD+18] Chethan Ramesh, Shivukumar B. Patil, Siva Nishok Dhanuskodi, George Provelengios, Sébastien Pillement, Daniel Holcomb, and Russell Tessier. FPGA side channel attacks without physical access. *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 45–52, 2018.

[RTSS09] 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 *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

[SGMT18] Falk Schellenberg, Dennis R. E. Gnad, Amir Moradi, and Mehdi Baradaran Tahoori. An inside job: Remote power analysis attacks on FPGAs. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1111–1116. IEEE, 2018.

[SLM+19] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*, 2019.

[TVT19] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Packet Chasing: Spying on network packets over a cache side-channel. *arXiv preprint arXiv:1909.04841v1*, 2019.

[VGGK19] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. CacheQuery: Learning replacement policies from hardware caches, 2019.

[VKM18] Pepe Vila, Boris Köpf, and José Francisco Morales. Theory and practice of finding eviction sets. *arXiv preprint arXiv:1810.01497*, 2018.

[vSGBR18] Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious Management Unit: Why stopping cache attacks in software is harder than you think. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 937–954, Baltimore, MD, Aug 2018. USENIX Association.

[vSMO+19] 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 *S&P*, May 2019.

[WTM+19] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. JackHammer: Efficient rowhammer on heterogeneous FPGA-CPU platforms, 2019.

[Xil19] Xilinx. Accelerator cards. https://www.xilinx.com/products/boards-and-kits/accelerator-cards.html, 2019. Accessed: 15-10-19.

[YF13] Yuval Yarom and Katrina E. Falkner. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. *IACR Cryptology ePrint Archive*, 2013:448, 2013.

[YSG+19] 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)*, volume 1, pages 56–72, Los Alamitos, CA, USA, May 2019. IEEE Computer Society.

[YWCL14] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. COLORIS: A dynamic cache partitioning system using page coloring. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 381–392, Aug 2014.

[ZJRR12] 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*, pages 305–316, New York, NY, USA, 2012. ACM.

[ZRZ16] 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*. ACM, 2016.

[ZS18] 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. IEEE, 2018.

Verilog Code

Listing 1: Code of the sending pattern generator contained in the covert channel AFU.

```
1   `include "module_serializer.sv"
2
3   module covert_channel
4           #(
5                   parameter DATA_WIDTH=64,
6                   parameter CNTR_BITS=10
7           )
8           (
9                   input                   clk,
10                  input                   rst,
11                  input                   clk_en,
12                  input                   i_load,
13                  input [CNTR_BITS-1:0]   i_stuf_bits,
14                  input [CNTR_BITS-1:0]   i_bit_rep,
15                  input [DATA_WIDTH-1:0] i_data,
16                  output logic            o_data,
17                  output logic            o_ready
18          );
19
20          localparam DATA_WIDTH_LOG = $clog2(DATA_WIDTH-1);
21
22          typedef enum logic [2:0] {
23                  IDLE,
24                  ENC,
25                  REP,
26                  STUF
27          } state_t;
28          state_t                 state;
29
30          logic                   ser_en;
31          logic                   ser_data;
32          logic                   ser_busy;
33
34          logic                   stuffing_done;
35          logic [CNTR_BITS-1:0]   int_stuf_bits;
36          logic [CNTR_BITS-1:0]   int_stuf_cnt;
37
```

```
38              logic                       repetition_done;
39              logic [CNTR_BITS-1:0]       int_bit_rep;
40              logic [CNTR_BITS-1:0]       int_rep_cnt;
41
42              logic                       encoding_done;
43              logic [DATA_WIDTH_LOG:0] enc_bits_cnt;
44
45              logic                       rep_stuf_done;
46              logic                       int_done;
47
48              // state flag driver
49              always_comb
50              begin
51                      stuffing_done = (int_stuf_cnt == int_stuf_bits);
52                      repetition_done = (int_rep_cnt == int_bit_rep);
53                      encoding_done = (enc_bits_cnt == DATA_WIDTH);
54                      rep_stuf_done = repetition_done && stuffing_done;
55                      int_done = encoding_done && rep_stuf_done;
56              end
57
58              // read input
59              always_ff @(posedge clk)
60              begin
61                      if ((state == IDLE) && i_load)
62                      begin
63                              if (i_stuf_bits < CNTR_BITS'(1))
64                                      int_stuf_bits <= i_stuf_bits;
65                              else
66                                      int_stuf_bits <= i_stuf_bits - 1;
67                              if (i_bit_rep < CNTR_BITS'(2))
68                                      int_bit_rep <= CNTR_BITS'(0);
69                              else
70                                      int_bit_rep <= i_bit_rep - 1;
71                      end
72              end // always_ff @
73
74              serializer #(DATA_WIDTH) ser (
75                      .i_clk(clk),
76                      .i_rst(rst),
77                      .i_clk_en(ser_en),
78                      .i_load((state == IDLE) && i_load),
79                      .i_data(i_data),
80                      .o_data(ser_data),
81                      .o_busy(ser_busy)
82              );
83
```

```
84              // ser_en driver
85              assign ser_en = (clk_en && rep_stuf_done);
86
87              // enc_bits_cnt driver
88              logic encoding;
89              assign encoding = (state == ENC);
90              always_ff @(posedge clk)
91              begin
92                      if (rst || (state == IDLE))
93                              enc_bits_cnt <= DATA_WIDTH_LOG'(0);
94                      else if (clk_en && encoding)
95                              enc_bits_cnt <= enc_bits_cnt + 1;
96                      else
97                              enc_bits_cnt <= enc_bits_cnt;
98              end
99
100             // int_rep_cnt driver
101             logic repreating;
102             assign repeating = (state == REP);
103             always_ff @(posedge clk)
104             begin
105                     if ( rst || (clk_en && encoding) )
106                             int_rep_cnt <= CNTR_BITS'(0);
107                     else if (clk_en && repeating)
108                             int_rep_cnt <= int_rep_cnt + 1;
109                     else
110                             int_rep_cnt <= int_rep_cnt;
111             end
112
113             // int_stuf_cnt driver
114             logic stuffing;
115             assign stuffing = (state == STUF);
116             always_ff @(posedge clk)
117             begin
118                     if ( rst || (clk_en && (encoding || repeating)) )
119                             int_stuf_cnt <= CNTR_BITS'(0);
120                     else if (clk_en && stuffing)
121                             int_stuf_cnt <= int_stuf_cnt + 1;
122                     else
123                             int_stuf_cnt <= int_stuf_cnt;
124             end
125
126             // o_ready driver
127             assign o_ready = (state == IDLE);
128
129             // o_data driver
```

```
130             always_comb
131             begin
132                     if (clk_en && (repeating || encoding))
133                             o_data = ser_data;
134                     else
135                             o_data = 1'b0;
136             end
137
138             // state machine
139             always_ff @(posedge clk)
140             begin
141                     if (rst)
142                             state <= IDLE;
143                     else
144                     begin
145                             case (state)
146                                     IDLE: begin
147                                             if (i_load)
148                                                     state <= ENC;
149                                     end
150                                     ENC: begin
151                                             if (clk_en)
152                                                     state <= STUF;
153                                     end
154                                     REP: begin
155                                             if (clk_en)
156                                                     state <= STUF;
157                                     end
158                                     STUF: begin
159                                             if (clk_en && int_done)
160                                                     state <= IDLE;
161                                             else if (clk_en && rep_stuf_done)
162                                                     state <= ENC;
163                                             else if (clk_en && stuffing_done)
164                                                     state <= REP;
165                                     end
166                                     default: begin
167                                             state <= state;
168                                     end
169                             endcase // case (state)
170                     end // else: !if(rst)
171             end // always_ff @
172
173             endmodule // covert_channel
```