Fabian Rauscher, BSc

GPU Cache Attacks

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisors

Daniel Gruss

Institute of Applied Information Processing and Communications

Graz, October 2022

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date, Signature

Acknowledgements

First, I would like to thank my supervisor Daniel Gruss and the other members of the CoreSec group for all the support and feedback they provided me with while working on this thesis.

Second, I want to thank my mother, Andrea Rauscher, for all the work and time she spent raising me and all the support she provided. Being a single mother of two is not easy, but I think she did an exceptionally good job.

Third, I want to thank Christina Ritter for emotionally supporting me and accepting that I sometimes have to work on weekends.

Finally, I thank all my friends and family for all the exciting and interesting past years.

Abstract

Modern GPUs are not only used for graphics processing but high-security applications such as high-volume encryption. Previous work has demonstrated side-channel attacks on GPUs using electromagnetic radiation, the last-level cache in multi-GPU systems, and shared memory bank conflicts. However, so far there are no practical cache attacks on single GPU systems.

In this thesis, we propose a Prime+Probe-based attack that targets single Nvidia GPUs in a shared environment and can recover the full AES-128 key on a GPU-accelerated AES implementation. We evaluate cache access timings and their relation to the particular addresses they are measured from and the location on the GPU. Using this relation, we reverse-engineer the cache layout, cache slices, and even the overall layout of modern Nvidia GPUs from both the Turing and Ampere architecture. We propose an efficient and fast way to build cache eviction sets by taking advantage of the exact cache access timings of memory locations. With our eviction sets, we build a Prime+Probe-based cross-user cache covert channel on modern Nvidia GPUs with a transfer speed of 2.5 kB/s. We present a first-round cache side-channel attack on an AES-128 T-table implementation running on a modern Nvidia GPU that can recover 48 bit of the key. Furthermore, we present a last-round attack on AES that can recover the full key requiring only ten thousand encryptions with the unknown key in $15.988 \pm 1.960s$.

Keywords: Cache Attacks · GPU Attacks · CUDA · Vulkan · Prime+Probe

Kurzfassung

Moderne GPUs werden nicht nur für Grafikverarbeitung, sondern auch für Hochsicherheitsanwendungen wie die Verschlüsselung von großen Datenmengen verwendet. Frühere Arbeiten haben Seitenkanalangriffe auf GPUs mit elektromagnetischer Strahlung, den Last-Level-Cache in Multi-GPU-Systemen und Konflikte zwischen gemeinsam genutzten Speicherbänken demonstriert. Bisher gibt es jedoch keine praktischen Cache-Angriffe auf Systeme mit nur einer GPU.

In dieser Arbeit schlagen wir einen Prime+Probe-basierten Angriff vor, der auf einzelne Nvidia-GPUs in einer gemeinsam genutzten Umgebung abzielt und den vollständigen AES-128-Schlüssel auf einer GPU-beschleunigten AES-Implementierung extrahieren kann. Wir schauen uns die Cache-Zugriffszeiten und ihre Verbindung zu den jeweiligen Adressen, von denen sie gemessen werden, und zum Speicherort auf der GPU an. Unter Verwendung dieser Informationen folgern wir Rückschlüsse auf das Cache-Layout, die Cache-Slices und sogar das Gesamtlayout moderner Nvidia-GPUs sowohl auf der Turing- als auch auf der Ampere-Architektur. Wir schlagen einen effizienten und schnellen Weg vor, um Cache-Eviction-Sets zu erstellen, indem wir die genauen Cache-Zugriffszeiten von Speicherorten ausnutzen. Mit unseren Eviction-Sets bauen wir auf modernen Nvidia-GPUs einen Prime+Probe-basierten benutzerübergreifenden Cache-Covert-Kanal mit einer Übertragungsgeschwindigkeit von 2.5 kB/s auf. Wir präsentieren einen Last-Round-Seitenkanalangriff auf eine AES-128-T-Table-Implementierung, die auf einer modernen Nvidia-GPU läuft und 48 bit des Schlüssels wiederherstellen kann. Darüber hinaus präsentieren wir einen Last-Round-Angriff auf AES, der den vollständigen Schlüssel wiederherstellen kann und nur zehntausend Verschlüsselungen mit dem unbekannten Schlüssel benötigt in $15.988 \pm 1.960s$.

Schlagwörter: Cache-Angriffe · GPU-Angriffe · CUDA · Vulkan · Prime+Probe

Contents

List of Figures

Chapter 1

Introduction

Over the past decade, cache side channels have proven to be an extremely powerful tool not only for direct attacks on applications but also as a building block for more powerful microarchitectural attacks such as Spectre-type attacks [24, 25, 44], Meltdown [27], and LVI [45]. Caches are a fundamental part of modern CPUs. They work around the low access speeds and high latency of DRAM and any other kind of memory-mapped I/O.

The advances and popularity of neural networks and the high availability of modern high-performance multi-purpose Graphics Processing Units (GPUs), even in the low-end consumer market, have led to a rise of General Purpose Graphics Processing Units (GPGPUs) throughout the past years. Nvidia, in particular, played a huge role in this by making every modern Nvidia GPU compatible with their GPGPU API CUDA, including low-end consumer GPUs [35]. More and more applications take advantage of the GPU not only by using the typical shader pipeline that such a GPU might have implemented but also for general-purpose computing such as artificial intelligence, video editing [41], data processing of enormous data sets [3], and even cryptography [30, 34, 14, 19]. This increase in GPU usage for general-purpose computation and therefore increase in the complexity of modern GPUs leads to a higher need for security on this hardware that was previously mainly used for low-security applications. In particular, in the case of cryptography, security is of the utmost importance.

Cache side channels have received a high amount of attention in the past on all kinds of CPUs [9, 24, 25, 44, 45, 47]. GPUs are often ignored when it comes to side-channel attacks due to their only recent rise in popularity for more general applications. Modern GPUs are designed for highly parallelized workloads that execute the same code on varying underlying data. While the workload of GPUs and CPUs is fundamentally different, GPUs incorporate caches similarly to modern CPUs to improve DRAM access timings and overall access speed.

In this thesis, we take a close look at the caches of modern Nvidia GPUs. We show not only that cache attacks are practical on GPUs but also that there is a lot of undocumented information about GPU designs that can be extracted by observing the timing behavior of cache accesses and fences. There is a clear correlation between the physical location an SM has on a GPU and the L2 cache access timings and memory access timings measured on it. By closely observing L2 cache access timings, we are able to recover the exact L2 cache slice that a memory location belongs to. We use this information to create the currently fastest and most efficient way for building eviction sets on Nvidia GPUs without access to physical addresses. The found eviction sets can enable further cache attacks such as Prime+Probe and Evict+Time [40]. To achieve this, we not only take advantage of the slice-related access timings to divide the L2 into multiple smaller search spaces but also that an application running on the GPU has exclusive access to the L2 and other resources, making timing measurements extremely accurate.

We create a simple Prime+Probe-based covert channel on a modern Nvidia GPU that can transfer data with up to 2.5 kB/s between a Vulkan and a CUDA application run by two completely independent users sharing a GPU. We demonstrate a first- and last-round attack on an AES-128 T-table implementation running on an Nvidia GPU. To build the eviction sets for the T-tables, we use our proposed algorithm for building L2 eviction sets to find an eviction set for each L2 cache set and filter through them by measuring encryptions with a known key and known plaintext. Our first-round attack on the AES-128 T-table implementation is able to recover 48 bits of the AES key, and our last-round attack is able to recover the full 128 bit AES key by monitoring only ten thousand encryptions.

Contributions.

1. We evaluate cache access timings and their relation with not only the particular addresses they are measured from but also the location on the GPU.
2. We determine cache timings and fence timings and reverse-engineer the cache layout, cache slices, and even the overall layout of modern Nvidia GPUs from both the Turing and Ampere architecture.
3. We propose an efficient and fast way to build eviction sets by taking advantage of exact cache access timings of memory locations.
4. We build a cross-user cache covert channel on modern Nvidia GPUs with a transfer speed of 2.5 kB/s.
5. We propose Prime+Count, a stronger version of the Prime+Probe attack that can recover the exact number of cache lines evicted by a victim.

6. We present a first and last-round cache side-channel attack on an AES-128 T-table implementation running on a modern Nvidia GPU that is able to recover 48 bits of the key in the case of the first-round attack and the full key in the last-round attack.

Outline. Chapter 2 gives an overview of cache architectures, modern cache attacks, and modern Nvidia GPU architectures. Chapter 3 explains our timing measurement methods, observations we made about cache access and fence timings, and an efficient algorithm to search for L2 cache eviction sets. In Chapter 4, we propose a Prime+Probe-based cross-application cache covert channel that runs on a modern Nvidia GPU. In Chapter 5, we present a first-round and a last-round attack on an AES-128 T-table implementation running on a GPU, including a full key recovery. In Section 2.5, we discuss work related to our findings and attacks. We summarize our results and conclude in Chapter 6.

Chapter 2

Background

2.1 Cache Architecture

Modern DRAM is very slow compared to CPUs. A typical DRAM access can take between 200 and 300 CPU cycles. This is a significant amount of time the CPU would need to stall to receive memory, given that there are many instructions that can be completed in just a single cycle. Due to this, memory accesses on modern CPUs do not directly access the DRAM but the on-chip cache. The CPU fetches data from DRAM into the cache if the accessed data is not present in the cache. By storing recently accessed data in this middle layer, it is possible to decrease the overall memory access time significantly, down to a few cycles depending on the cache layer the data has to be loaded from.

Figure 2.1 shows the typical cache hierarchy of a modern x86 CPU. L1 through L3 consist of fast on-chip memory increasing in size the further away from the CPU cores they are. The cores directly access the L1, which is often divided into a data cache (L1D) and an instruction cache (L1I).

L2 and L3 both store instructions and data. When a core accesses memory, the L1 is searched first. If the data is not present in the L1, the next cache level is searched. If the data is not present at any cache level, the CPU will fetch it from DRAM. Once the data reaches the CPU, it is propagated back up through the searched caches and forwarded to the core that sent the initial request. If the data is present in one of the higher levels, such as the L1, lower levels, such as the L2 and L3, do not have to be searched, decreasing access times if data is present in one of the faster cache levels. Data found in the L1 is immediately forwarded to the core without checking the L2, L3, or the DRAM.

Since there are caches shared between cores, such as the last-level cache (LLC), which in Figure 2.1 is the L3, memory access timings can vary due to the possible influence of other programs running on other cores. Data stored in the LLC by one

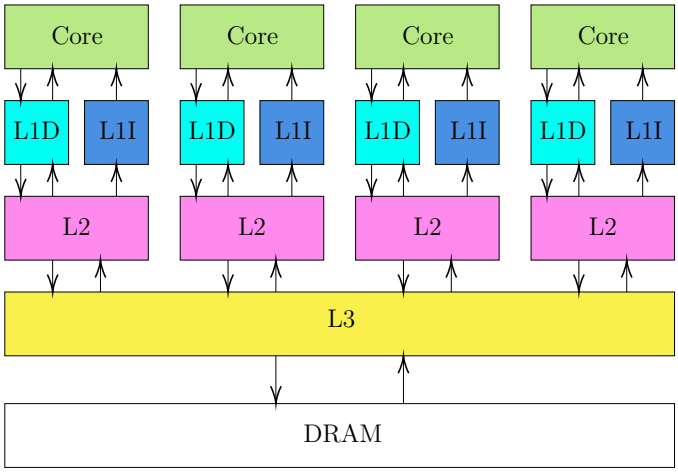


Figure 2.1: Typical cache hierarchy of a modern x86 CPU.

core running one program might get evicted by a different core running a different program to free up required cache space and load different data from DRAM. This can influence the runtime of programs in both a negative way, if needed data is evicted from the cache, and a positive way, if data that is needed by a program has already been loaded into the cache by a different program.

2.1.1 Cache Addressing

CPU caches have to be fast temporary memory storage with access times that only take a few cycles. The cache size is usually significantly smaller than the DRAM size, and every memory location needs to map to some location in the cache as quickly as possible. To achieve this mapping while still making it possible to store multiple memory locations that map to the same part of the cache at a time, modern processors employ set-associative caches [17]. For each cache access, the accessed address is split into three parts: the offset into the cache line, an index, and a tag. The offset is directly related to the cache-line size and consists of the least significant bits of the address. A cache-line size of 64 B would therefore mean 6 offset bits. The index is usually right after the offset and is the identifier of the cache set. The rest of the address is used as the tag. A cache set consists of multiple so-called ways, which are individual cache lines. A cache line usually consists of at least the data, a unique identifier called a tag, and a valid bit. The number of cache lines in a set is processor-dependent. To find the correct cache line in a set, the tag extracted from the address is simultaneously compared with all ways in the set. Since this is combinatorially for all cache lines in parallel, this lookup is almost instant. The corresponding cache line is returned if it contains

the matching tag and is valid. If no valid matching cache line can be found, the access results in a cache miss.

With virtual memory enabled, a memory location can have both a physical and a virtual address. Both of these addresses can be used for cache lookups. The two important parts for cache lookups, the tag, and the index, can each be derived from either the physical or the virtual address. The offset stays the same in both cases since address translation is done at page size granularity, and the smallest page size is usually significantly larger than the cache line size. There are three common ways to split up the index and tag calculations:

- Virtually Indexed Virtually Tagged (VIVT)
- Virtually Indexed Physically Tagged (VIPT)
- Physically Indexed Physically Tagged (PIPT)

VIVT is fast since it does not require a translation from the virtual to the physical address. PIPT requires a full address translation before the cache can be searched. This extra translation slows down cache accesses significantly compared to VIVT but has the advantage that shared memory has to be stored in the cache only once, even if the memory area is mapped to different virtual addresses across different address spaces. Due to this, PIPT is a very appealing option for the LLC, which stores data accessed by multiple cores that might not share a virtual address space. VIPT computes the index from the virtual address and reads the tag from the physical address. While this might seem slow compared to VIVT, it usually has minimal performance overhead if the address translation is cached since the address translation can be done in parallel to indexing the cache set. If all of the index bits lie within the page offset of the smallest page size of the architecture, both PIPT and VIPT behave identically since the page offset part of an address is always the same in the virtual and the corresponding physical address.

2.1.2 Cache Slices

The LLC is usually shared across all cores of a CPU. This can lead to a high number of parallel accesses to the LLC and, therefore, congestion [31]. Due to this, on Intel CPUs, the LLC is divided into so-called cache slices [17]. Each cache slice consists of a set-associative cache and is directly connected to one CPU core. The slices are connected to each other to enable a core to access a cache slice to which it is not directly connected. Intel employs a ring, or a mesh interconnect for their CPUs [17]. Physical addresses are uniformly distributed across all cache slices. This uniformity is achieved with a hash function.

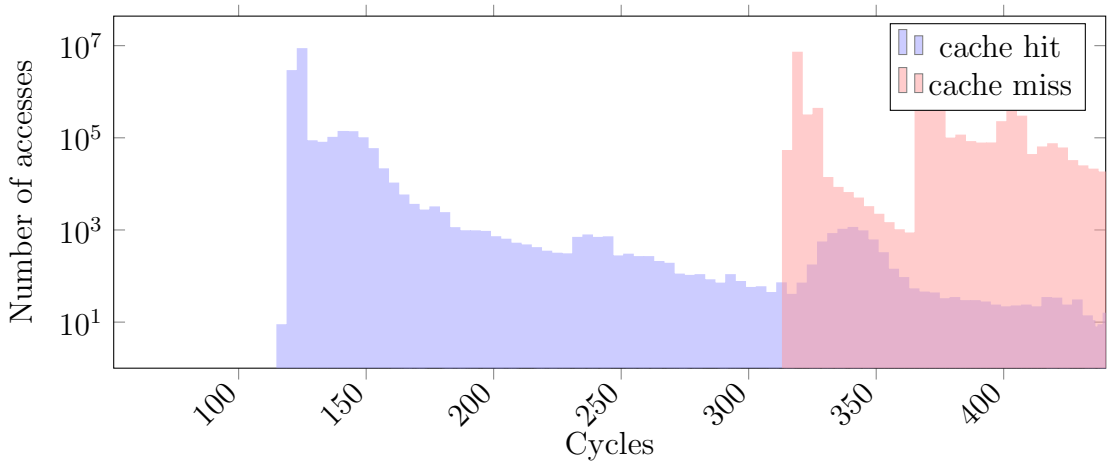


Figure 2.2: Cache hit and miss timings of an i7-3820. The cache hits are at around 125 cycles, and the cache misses at around 325 cycles. There is a clear difference between the cycle counts of a cache miss and a cache hit with a difference of over 150 cycles.

The hash function takes the physical address bits, excluding the cache line offset bits, and computes the slice ID. These functions are not officially available and can depend on the CPU manufacturer and sometimes the CPU generation. Slice functions for various processors have already been reverse-engineered [15, 18, 28]. Slice functions usually consist of multiple XOR functions applied to the physical address, resulting in a fast but predictable way to compute slice IDs.

2.2 Cache Attacks

Caches provide a considerable performance boost to modern processors, but they also provide the opportunity for side-channel attacks due to their shared nature. Since caches are not process-dependent, two completely independent workloads from two different processes can load data into the cache for each other and evict data cached from a different process. While data cannot be directly leaked this way, the knowledge that certain data was cached or evicted can leak sensitive information. In this section, we discuss existing cache attacks and explain how they make it possible to leak information from other processes that run on the same processor.

2.2.1 Flush+Reload

Flush+Reload is a cache-based side-channel attack that takes advantage of shared memory and the often physically-indexed and physically-tagged last-level cache [47]. Figure 2.2 shows the cycle difference between a cache miss and a cache hit on an i7-3820. The exact instructions measured are an `mfence`, a `mov`, and another `mfence`. The `mfence` instructions provide serialization of memory operations and take up most of the cycles in our plot in the case of a cache hit. There is a clear difference between a cache hit at around 130 cycles and a cache miss at around 320 cycles. The exact cycle counts are highly hardware-dependent, but there is always a clear difference between a hit and a miss.

Flush+Reload leverages the time difference between a cache hit and a cache miss to detect if a memory region has been recently accessed and infers information from it. The last-level cache of modern CPUs usually has physically-indexed and physically-tagged (PIPT) cache lines. This means that the index that specifies the cache set that a memory chunk is cached in and the unique tag that identifies the correct cache line in the set are both derived from only the physical address. Physical memory accessible through multiple different virtual addresses and even from different address spaces is only cached once while being accessible through all of these separate mappings.

A typical Flush+Reload attack requires some form of shared memory between the attacker and the victim and information that can be extracted through accesses to this shared region. An attacker can create shared memory with other applications on modern operating systems by loading data from the disk, such as shared libraries or other files, and through active deduplication through the operating system. Both of these methods result in shared copy-on-write pages between processes. Shared libraries, in particular, are commonly used for this kind of attack since it is easy to reliably get shared memory with the victim if the library the victim application uses is accessible to the attacker. To set up this kind of shared memory, the attacker maps the shared library or file the victim uses into its own address space.

Shared memory through deduplication is rarely used since the attacker has to create an exact copy of the victim's page that it wants to monitor, and the operating system has to support page deduplication. While many modern hypervisors [46, 5], and operating systems [2] support this, it often requires more information about the victim's memory than mapping the same library into the attacker's address space.

Once the attacker has a shared memory region containing an area of interest with the victim, they flush the memory area they want to monitor. The victim is now being scheduled. Once the victim is done, the attacker accesses the memory

location and measures the access time. If the access time indicates a cache hit, the victim has recently accessed the memory location. If the access time indicates a cache miss, it was most likely not accessed by the victim.

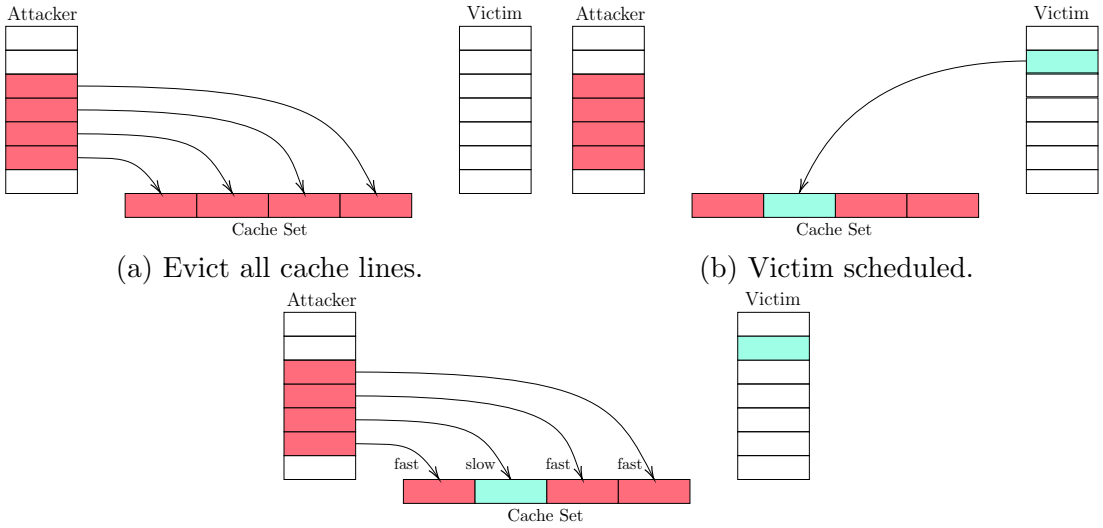
Initially proposed as an attack on shared libraries [47], due to its simplicity and the information one can gain about the microarchitectural state by knowing if data is cached, Flush+Reload is now used as an essential building block for microarchitectural attacks [27, 24, 43, 42].

2.2.2 Flush+Flush

The Flush+Reload attack requires the attacker to access the targeted shared memory and measure the access time to determine if the victim has loaded the data [13]. Once the attacker checks the memory location, they flush it from the cache. The flush is required to reset the cache to a state where the next time the victim is scheduled they can load the data back into the cache, which the attacker can, later on, observe again. Due to this, the attacker code always consists of a timed memory access and a flush to reset the state for future measurements. The data fetched is not required by the attacker, but whether it was recently cached. The memory access introduces a considerable amount of overhead by loading data from the slow DRAM only to evict it right afterward.

The Flush+Flush attack removes the memory access by the attacker by instead timing the flush operation. A typical cache-line flush on a multi-core processor does not only consist of the eviction of the corresponding cache line from all caches directly accessible by the core. Since the LLC on modern x86 CPUs is often inclusive [17], cache lines flushed from it must be removed from the L1 and L2 of all cores that share the LLC. Due to this extra work that only has to be done if the memory is cached, a small timing difference can be observed between a flush on memory that is cached and memory that is not cached. Memory that is not cached can be flushed from the cache faster since the flush instruction only has to look up whether the memory is cached, and if it is not cached, the CPU does not have to search and evict cache lines from other L1 and L2 caches.

The removal of the memory access increases the performance of the attack significantly. This way of checking for cached data removes one of the primary time sinks of other cache-based side-channel attacks and led to the fastest cache covert channel at the time Flush+Flush was introduced [13].



(a) Evict all cache lines.

(b) Victim scheduled.

(c) Check evictions.

Figure 2.3: Overview of the Prime+Probe attack. First, the cache set is filled by the attacker. Second, the victim loads memory into the same cache set evicting cache lines of the attacker. Last, the attacker checks if their cache lines were evicted by timing the memory accesses.

2.2.3 Prime+Probe

The Prime+Probe attack, as described by Osvik et al. [40], allows an attacker to monitor a victim's cache set usage through cache evictions. Contrary to other attacks, such as Flush+Reload and Flush+Flush, Prime+Probe does not extract information from cache hits and does not require shared memory. Prime+Probe takes advantage of a shared cache and how evictions work in modern caches. Since the size of a cache set is limited, multiple addresses map to the same cache set, even from different processes. Once a set is full and memory that maps to this set is loaded from DRAM, cache lines that are currently present in the cache set have to be evicted to free up space. If such an eviction occurs and the evicted cache line is reaccessed afterward, it must be reloaded from either lower cache levels or the DRAM.

Prime+Probe exploits this eviction behavior to detect if a victim loaded a cache line. This monitoring of cache-line loads is achieved with a so-called eviction set. Members of an eviction set have to occupy the whole cache set to detect all accesses to the cache set. For a cache set that can hold W cache lines, an eviction set consists of at least W different memory lines that map to the same cache set in order for it to occupy the whole cache set. When the members of an eviction set are loaded into the L3, they fill up the cache set it is responsible for.

In an attack, the attacker first loads the members of an eviction set, or even from multiple eviction sets, if the goal is to monitor multiple cache sets at a time, into memory, as shown in Figure 2.3a. The monitored cache set now only consists of the eviction set members, and every new load to it will evict a member of the eviction set. Next, the victim is scheduled. If the victim accesses addresses that map to the monitored cache set, parts of the eviction set are evicted, as shown in Figure 2.3b. When the attacker is scheduled again, they can measure the access timings to all memory lines of an eviction set, as shown in Figure 2.3c. The detected evictions inform an attacker about if a cache set was accessed by the victim. The exact cache line evicted depends on the replacement policy employed by the CPU. Popular replacement policies include Least Recently Used (LRU), Not Recently Used (NRU), and Random. Usually, an approximation of LRU, pseudo LRU (pLRU), is used since accurate LRU is difficult to implement.

Compared to Flush+Reload, Prime+Probe does not monitor one specific memory location. Instead, it looks at whole cache sets, making it significantly more flexible in its application because it does not require shared memory but only a shared cache.

2.2.4 Evict+Time

Evict+Time, proposed by Osvik et al. [40], takes a very similar approach to the previously discussed Prime+Probe attack. The attack focuses on observing the timing behavior cache evictions have on the runtime of a victim application. Similar to Prime+Probe, the first step in Evict+Time is to build an eviction set for the cache set to be monitored. The attacker measures the execution time of the victim. Then the attacker evicts all cache lines in a cache set. Lastly, the attacker measures another execution of the victim. If the execution takes longer, the attacker evicted a cache line that the victim application accessed. If the execution takes the same amount of time, then the monitored cache set was not accessed.

Evict+Time takes advantage of the time difference between cache hits and misses. When the attacker loads the members of the eviction set, the victim data present in the corresponding cache set is evicted. Depending on whether the victim has to reload the data into the cache set, a slight timing difference can be observed. This timing difference provides an attacker with information about the victim's execution. While the execution time difference of a few extra cache misses is small compared to the overall execution time of a typical victim application, as long as the attacker can repeat such a measurement often enough, it is detectable and exploitable.

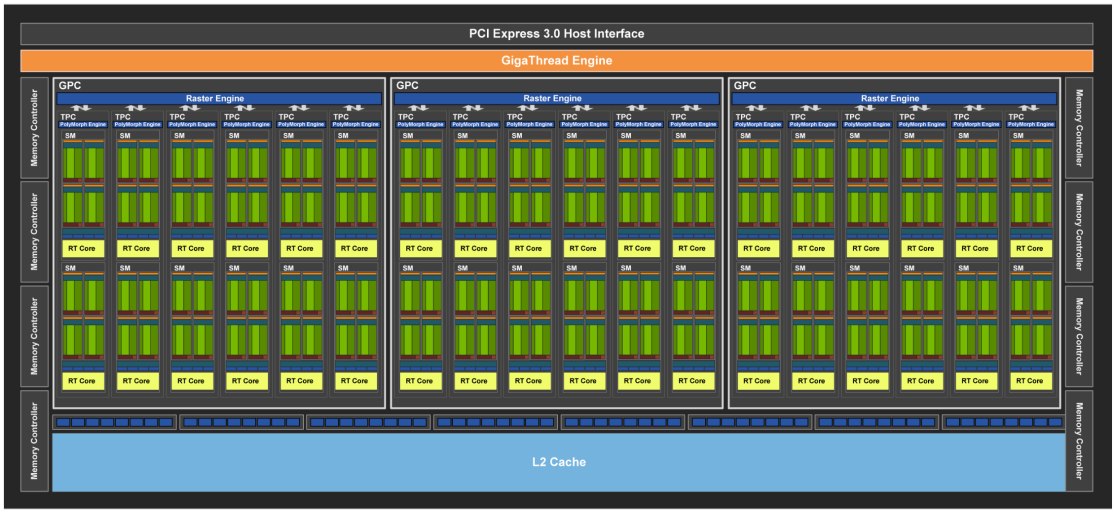


Figure 2.4: Chip layout of an RTX 2070 GPU taken from [38].

2.3 GPU

Over the last decades, Graphics Processing Units (GPUs) changed significantly in their roles and design. Initially designed to manage video output through the frame buffer, GPUs evolved from providing basic shaders and post-processing to a more general second processor, which offers high parallel throughput for graphics-related applications and general-purpose computing. Advances in modern-day graphics processing and the possibility for a very high amount of parallelism make General Purpose Computation on Graphics Processing Units (GPGPU) more and more relevant. Due to this, both GPU manufacturers and other vendors provide not only APIs for graphics processing such as Vulkan, OpenGL, and DirectX, but also APIs for GPGPU such as CUDA [35] for Nvidia GPUs, ROCm [1] for AMD GPUs, and OpenCL [10] as an open-source platform-independent API.

2.3.1 Nvidia GPU Architectures

The layout of an RTX 2070 can be seen in Figure 2.4. A typical Nvidia GPU consists of a global L2 cache and multiple Graphics Processing Clusters (GPCs). A GPC consists of multiple Texture Processing Clusters (TPCs) and, depending on the exact architecture, some extra hardware, such as a raster engine in the Ampere and Turing architectures. The number of TPCs, a given GPC consists of depends on the exact chip type and varies even within an architecture. TPCs consist of multiple Streaming Multiprocessors (SMs) and, depending on the architecture, extra hardware such as a polymorph engine in the Ampere and Turing architectures.

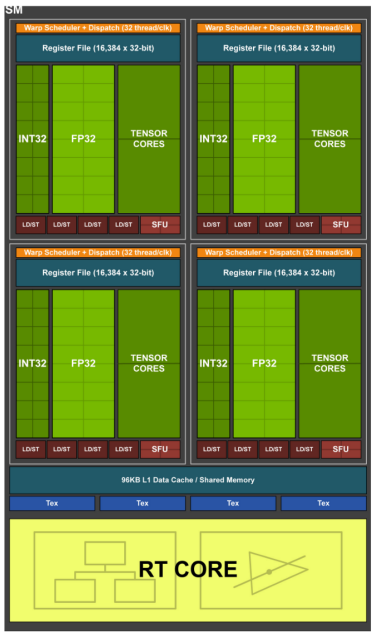


Figure 2.5: Nvidia Turing SM layout taken from [38].

Contrary to GPCs, TPCs are the same within a given architecture, and in the case of the Ampere, Turing, and Pascal architectures, they consist of two SMs [38].

Figure 2.5 shows the layout of an SM from the Ampere architecture. An SM is the part of an Nvidia GPU that comes closest to a traditional CPU core. It consists of an L1 cache, Shared Memory, which is fast on-chip memory that can be used by applications for frequently accessed data, a separate texture cache, multiple processing blocks, and in recent architectures, a raytracing core. Processing blocks are the parts of a GPU that execute instructions. Each processing block consists of one warp scheduler, one register file, one dispatch unit, and multiple sets of math and other units. Processing blocks are comparable to execution units on CPUs. On more recent architectures, processing blocks might also include an instruction L0 cache and tensor cores [37].

Nvidia GPUs have full support for virtual memory on the GPU. Modern Nvidia GPUs support virtual address sizes of up to 49 bit. The exact address size is dependent on both the GPU architecture and the architecture of the CPU. Virtual address spaces on the GPU are generally separate from the virtual address spaces on the CPU. The required paging structure is usually stored in the GPUs DRAM [35].

Modern GPUs are slowed down by DRAM access times and speed. To combat this, they employ caches similar to CPU caches. An example of a GPU cache hierarchy can be seen in Figure 2.6, which depicts the cache hierarchy of Nivida

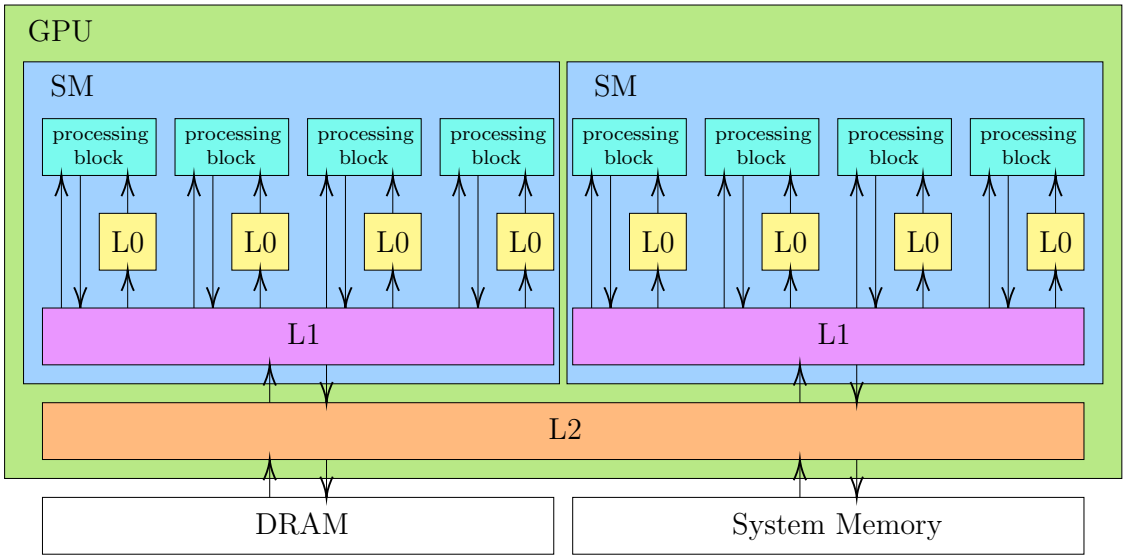


Figure 2.6: Cache hierarchy of Nvidia Turing GPUs.

Turing GPUs. While there have been a lot of different GPU architectures over the last few years, the overall cache hierarchy only received slight changes. The main change is the addition of an L0 instruction cache per processing block [37]. Nvidia GPUs have L0, L1, and L2 caches, with L2 being the last level cache (LLC). The L0 and L1 are VIVT, while the L2 is PIPT [20]. The L2 is a 16-way set-associative L2 cache with 128 B cache lines [20, 36]. All processing blocks within an SM share one L1 cache, and the whole GPU has one global L2 cache. The caches cannot be disabled, and all accesses from the GPU to the GPUs DRAM or the system memory must go through both the L1 and the L2. The GPUs DRAM can only be accessed through the L2, even when performing DMA from system memory.

2.3.2 CUDA

CUDA is a general-purpose compute platform for writing highly parallelized applications designed to run on Nvidia GPUs [35]. Compared to open-source solutions such as OpenCL, CUDA provides an experience tailored to Nvidia GPUs, including the support for inline assembly using Nvidias PTX ISA. Due to this, CUDA can provide significantly more functionality to the programmer that other GPU manufacturers might not support. Examples are a clock function to measure time, a more accurate consistency and memory model, more control over synchronization, support for prefetching, different memory access types, and instructions to control caching behavior. Each CUDA process running on the host has its own CUDA context. The CUDA context contains all relevant process-specific CUDA information, such

as the currently active device, GPU memory allocated, a virtual address space for the GPU, and errors that previously occurred.

A subroutine that runs on the GPU is called a kernel. Kernels are mostly normal C/C++ functions with a special specifier that defines whether they can only be called from the GPU or from the host application and the GPU. To execute a kernel, the host has to specify the number of blocks and threads per block. An arbitrary number of arguments can be passed to kernels. Kernel functions can, through defined variables, read out their block-ID and thread-ID, which can be used to determine the data a thread is supposed to process. Code running on the GPU does not have direct access to host memory. Therefore, data that is not passed by value has to be copied into a previously allocated GPU buffer before launching a kernel and back to the host afterward if data is supposed to be read. Since a single call to a kernel can launch thousands of threads simultaneously, a kernel can only return data by receiving a pointer as a function argument and writing to it in the kernel.

Threads running in a kernel do not necessarily all run simultaneously, and code should not depend on true parallelism. CUDA does not supply any locking mechanisms, and it is highly discouraged to implement custom locks. Workloads on the GPU should be designed to work in parallel on independent data most of the time. However, CUDA does provide atomic operations on both float and integer data types and functions to synchronize thread groups, similar to POSIX barriers.

Launching a kernel is a non-blocking operation that only schedules the kernel to be executed on the GPU. The actual execution of kernels is asynchronous. CUDA has functions that provide synchronization points that block until all previously scheduled work on a CUDA context is complete. Commonly used synchronization points are **cudaMemcpy** (copies data to and from the GPUs DRAM), **cudaCtxSynchronize** (waits until all previous work is complete), and **cudaDeviceReset** (destroys all allocations and resets the current context).

Due to the asynchronous nature of kernels, errors can not be directly returned by them. Instead, following CUDA API functions that the host calls will return any pending errors from previous kernel executions. Once such an error occurs, all following CUDA API calls will return the same error, and kernel launches are silently ignored for the current CUDA context until the context is reset.

CUDA only allows one CUDA context to run code on a GPU at a time. Kernels should be designed to take advantage of as much of the GPU at a time as possible, and executing multiple kernels in parallel is highly discouraged. To allow multiple independent applications to access the GPU simultaneously, the GPU driver implements time slicing by regularly interrupting long-running code.

All Nvidia GPUs released over the last decade are compatible with CUDA, including consumer GPUs. Nvidia achieves this through their intermediate assembly language PTX [39] which is designed for GPU programming. Kernels are not directly compiled to the GPU architecture they will be running on. Instead, they are compiled to PTX assembly. The PTX assembly is then packed into the binary and compiled by the CUDA library to the target architecture at runtime.

While Nvidia's PTX ISA [39] is only a generic intermediate assembly language, it is very closely related to Nvidias low-level assembly language SASS which compiles directly to binary microcode that can be run on Nvidia GPUs. This extra abstraction layer allows PTX to provide functionality that would usually only be accessible through low-level assembly languages. Examples are specifiers for cache operations for data movement instructions such as **.cv** (do not cache), **.ca** (cache on all levels), **.cg** (cache only globally in the L2 cache), and **.lu** (last use, cache with an evict first policy). While most of these specifiers are only hints, they are usually not ignored by the compiler or architecture. In addition to extra flags, some instructions do not have a function equivalent in CUDA, such as **prefetch**, which requests a data load into a specified cache level, **discard**, which discards data from the cache without saving it, and instructions to influence cache eviction policies. Since PTX is not directly translated to machine code, it does not provide a fixed set of registers but uses generic register allocation. Most computational operations and register allocations have to provide a specific data type, such as float, binary or unsigned. PTX is not simply translated to the closer to the hardware assembly language SASS but is compiled to it, which includes very aggressive optimizations.

2.3.3 Vulkan

Vulkan is a modern cross-platform graphics and compute API developed by the Khronos Group [16]. Initially designed as an OpenGL replacement and released in 2016, Vulkan is not only highly optimized for graphics processing but also has the capability of performing GPGPU tasks with compute shaders. It provides an object-based design with execution states tied to command buffers, simplifying multi-threaded programs compared to a global execution state. Memory management and synchronization can be controlled by hand, and error checking can be enabled and disabled at compile time, enabling significantly faster release builds while providing high debuggability for production builds. Vulkans' support of all major GPU vendors and operating systems, and high performance, which is close to bare metal, makes it a viable alternative to existing graphics APIs such as OpenGL and DirectX.

Unlike DirectX and OpenGL, Vulkan does not provide its own human-readable shader language. Vulkan uses a bytecode format called SPIR-V for shaders to which higher-level shader languages such as DirectXs HLSL [32] and OpenGLs GLSL [12] can be compiled [11]. This extra level of abstraction provides a standardized and generic way for storing shaders that does not depend on one human-readable shader programming language. Vulkan supports many different shader types, including vertex shaders, mesh shaders, geometry shaders, fragment shaders, and compute shaders.

Compute shaders, in particular, allow for the processing of arbitrary information [16]. Code inside a compute shader is not required to do any rendering work. They do not have a well-defined set of input values, and the number of executions can be completely arbitrary, contrary to other shaders, such as vertex shaders which are executed once per vertex. This highly customizable shader allows Vulkan applications to take advantage of the GPU for generic programming tasks and tasks which might not work very well in any well-defined part of a typical rendering pipeline, such as particle systems or simulations. Due to this, compute shaders provide functionality for shader code that might not make sense for any other shader, including shared variables, fences for sequential consistency, and a basic set of atomic operations.

2.4 AES

AES-128 is a modern cryptographic algorithm that consists of 10 rounds, with each round performing the same operations except for the last round [7]. While many modern x86 CPUs provide hardware support for AES through an instruction-set extension, it is often still required to implement AES in software on processors that do not provide such a feature. A single round consists of substitute bytes, shift rows, mix columns, and add round key. Substitute bytes can be implemented as a table lookup into a 256 B lookup table. Shift rows is a byte-wise shift. Mix columns is a linear transformation over a finite field. Add round key is an XOR with the round key. Shift rows, add round key, and substitute bytes are trivial operations, while mix columns can be computationally expensive in software. To improve performance, T-table implementations of AES combine substitute bytes, shift rows, and mix columns into table lookups using precomputed tables and XOR calculations.

Figure 2.7 shows the code for a typical round of an AES T-table implementation. All tables hold the same values that are logically rotated by a different number of bytes depending on the T-table. Table 0 is rotated by 0 bytes, table 1 by 1 byte,

```
1 uint32_t tmp0 = table0[s0>>24] ^ table1[(s1>>16)&0xFF]
2                ^ table2[(s2>>8)&0xFF] ^ table3[s3&0xFF];
3 uint32_t tmp1 = table0[s1>>24] ^ table1[(s2>>16)&0xFF]
4                ^ table2[(s3>>8)&0xFF] ^ table3[s0&0xFF];
5 uint32_t tmp2 = table0[s2>>24] ^ table1[(s3>>16)&0xFF]
6                ^ table2[(s0>>8)&0xFF] ^ table3[s1&0xFF];
7 uint32_t tmp3 = table0[s3>>24] ^ table1[(s0>>16)&0xFF]
8                ^ table2[(s1>>8)&0xFF] ^ table3[s2&0xFF];
9
10 s0 = tmp0 ^ roundkey[0];
11 s1 = tmp1 ^ roundkey[1];
12 s2 = tmp2 ^ roundkey[2];
13 s3 = tmp3 ^ roundkey[3];
```

Figure 2.7: Example code of an AES-128 round using T-tables. Each input byte is used to index a T-table, and the resulting 4 B values are combined with XOR into four 4 B values. At the end, the round key is added to the four 4 B.

table 2 by 2 bytes, and table 3 by 3 bytes. This shift accounts for the shift rows operation. At the beginning of a round, the current 128 bit state is split into 4 32 bit parts. For each byte of a given 32 bit part, a table lookup is performed using one of the 4 T-tables each, resulting in four 32 bit values per 32 bit block which are then XORed. The last step of the round is to apply the current round key. The last AES round deviates from this slightly, only performing substitute bytes, shift rows, and add round key, leaving out the mix columns step. Since all T-tables contain the full AES S-Box, they can also be used for the last round by masking the unwanted bytes using a logical AND [7].

2.5 Existing GPU Side-Channel Attacks

This chapter provides information about GPU attacks and papers related to our findings discussed in this thesis.

GPU L2 cache attacks: Dutta et al. [8] propose an L2 cache covert channel and Prime+Probe attack on a multi-GPU system through the NVLink interface. They assume that the attacker and the victim are colocated on different GPUs that are connected through NVLink. This makes it possible for the attacker to

access the DRAM of the GPU the victim is currently running on. They leverage this to perform a Prime+Probe attack on the victim by measuring for L2 cache hits and misses when accessing data through NVLink on the victim's GPU. Our proposed attacks do not require more than one GPU and can even work on a single GPU consumer system.

Naghibijouybari et al. [33] propose an L2 cache attack on single GPU systems with Multi-Process Service (MPS) enabled. While similar to our work, MPS makes it possible to run multiple kernels on a GPU in parallel through context sharing. Enabling MPS can lead to a shared address space between kernels, and true parallelism is only allowed for kernels launched by the same user. MPS is also highly discouraged by Nvidia and disabled per default. Our attacks do not require MPS and work between different applications running under separate users.

Karimi et al. [23] demonstrate cache attacks on integrated GPUs used on mobile devices. They take advantage of the shared LLC cache between the CPU and the integrated GPU to attack an AES T-table implementation.

GPU Shared memory attacks: There exist multiple papers that exploit the GPUs shared memory by triggering bank conflicts. Jiang et el. [21] show an attack on an AES T-table implementation running on a GPU that takes advantage of the fast on-chip shared memory. They use conflicts that can occur when accessing shared memory to build a timing attack that checks for such conflicts when an encryption occurs by measuring the time an encryption takes. Through this timing side channel, they can recover all 16 key bytes of an AES-128 implementation using 10 million samples. Our attack requires only 10 thousand samples for a full key recovery. Jiang et al. [22] propose another timing attack that exploits bank conflicts in shared memory on Nvidia Kepler, Maxwell, Pascall, Volta, and Turing GPUs. Lin et al. [26] propose a scatter-gather-based mitigation to bank conflict-based timing side channels for table-based AES implementations.

Chapter 3

Cache Observations and Eviction Set Search

In this chapter, we take a closer look at L2 cache access times and their relation with cache sets, cache slices, and the physical layout of a GPU. We use the gathered information to improve cache attacks significantly and reveal previously undocumented parts of modern Nvidia GPUs.

First, we show how it is possible to measure L2 cache access times reliably using both CUDA and Vulkan.

Second, we take a closer look at L2 cache access times in general and their relation to the addresses accessed. Furthermore, we show how L2 cache timings on modern Nvidia GPUs relate to cache slices.

Third, we look at fence timings in relation to SMs. Moreover, we show how these metrics relate to the SM they are measured on and their relation to the physical layout of a GPU.

Finally, we show how our gathered knowledge can be leveraged to build eviction sets for the L2 cache on modern Nvidia GPUs. We propose an algorithm to build eviction sets fast and efficiently without the need for physical addresses.

3.1 Timing Measurement

CUDA and GLSL provide a clock function that returns the current value of a cycle counter. CUDA specifically does not only provide a clock function but also a special register in their intermediate assembly language PTX. This special register **clock64** is a 64 bit large cycle counter which gives a precise value for clock cycles since its last reset. The ease of access of this special register through basic move instructions makes it convenient to work with, even when using PTX. According

to our observations, CUDAs **clock** function and GLSLs **clockARB** function read out this special register on Nvidia GPUs.

The compilation step from PTX to SASS applies aggressive optimizations, including instruction reordering and removal of unnecessary instructions. These optimizations can be disabled in Nvidia's CUDA toolkit. According to our observations, the CUDA compiler only applies optimizations when compiling PTX to SASS, leaving the compilation from C++ to PTX unoptimized. Due to this, we decided to leave the optimizations enabled and tailor our attack kernel around them.

Algorithm 1 PTX L2 cache access time measurement code.

```
1 fence.sc.cta;
2 mov.u64 clk1, %%clock64;
3 fence.sc.cta;
4 ld.global.cg.f32 d1, [addr];
5 add.f32 tmp, d1, tmp;
6 fence.sc.cta;
7 mov.u64 clk2, %%clock64;
8 fence.sc.cta;
```

Our measurement PTX code can be seen in Algorithm 1. We measure instructions by reading twice from the **clock64** register and surrounding each read with fences to avoid instruction reordering. PTX has three different scopes for fences that each provide a different level of sequential consistency:

- **cta** for sequential consistency on SMs
- **gpu** for sequential consistency throughout the whole GPU
- **sys** for sequential consistency throughout all GPUs running the current program

Since our primary goal is to avoid reordering of instructions, **cta** fences suffice. For memory accesses, we use the load instruction **ld** with the cache global modifier **.cg**. The **.cg** modifier bypasses the L1 and only caches data in the L2. To avoid removal of the load instruction by the compiler, we add the loaded value to a temporary variable making it unlikely to be removed. This combination of carefully chosen instructions makes L2 cache timing measurements without the need to disable optimizations.

Algorithm 2 shows our GLSL measurement code. To measure time, we use the **clockARB** function from a GLSL extension which provides a monotonically incrementing 64 bit large counter similar to PTXs **clock64** register [16]. To avoid

Algorithm 2 GLSL L2 cache access time measurement code.

```
1 memoryBarrier();
2 uint64_t b = clockARB();
3 sum = atomicAdd(gpuBuffer.data[cur], sum);
4 atomicExchange(gpuBuffer.data[cur], sum);
5 uint64_t e = clockARB();
6 memoryBarrier();
```

reordering by the compiler, we surround the whole measurement block with memory barriers. Contrary to PTX, there is no standardized way to specifically access the L2 cache in GLSL. We use atomic operations provided by a GLSL extension to force an L2 access. While on modern CPUs, atomic operations do not always require an access to the LLC, according to observations using atomic operations on Nvidia GPUs leads to consistent L2 cache accesses. Due to this, we use atomic operations as a replacement for PTX's cache global modifier. Similar to our PTX measurement code, we use a temporary variable to which we add the value at the memory location that we access. This use of the data at the target address and an assert later in the code prevents the compiler from removing the L2 cache access.

While CUDA provides a lot of flexibility and even has the option to write inline assembly, a shader language such as GLSL has its advantages. Context switches are expensive, and CUDA applications focus on fast parallel computations that use the whole GPU. According to our observations, CUDA applications get interrupted less often if only CUDA applications are running, switching between running kernels only about every second. Graphics applications are often more time-sensitive, and there can be multiple applications at a time that require GPU time. Shaders are scheduled more often for shorter periods, even if CUDA kernels are running. An attacker application that uses Vulkan shaders is scheduled much more frequently on the GPU, simplifying the extraction of information and providing a significant advantage for most attacks.

To differentiate between L2 cache hits and cache misses, we need to know how many cycles each approximately take. We measured a 6 MB region in 128 B increments which is the exact cache-line size on Nvidia GPUs. To measure cache hits, we separate the memory region into smaller chunks of 128 kB and repeatedly measure the access timings when iterating over the memory chunks. 128 kB is significantly smaller than the L2 cache but too large for the L1 cache on our RTX 2070. Due to this, almost all memory accesses after the first iteration should result in L2 cache hits. For the cache misses, we iterate over the whole 6 MB at once multiple times

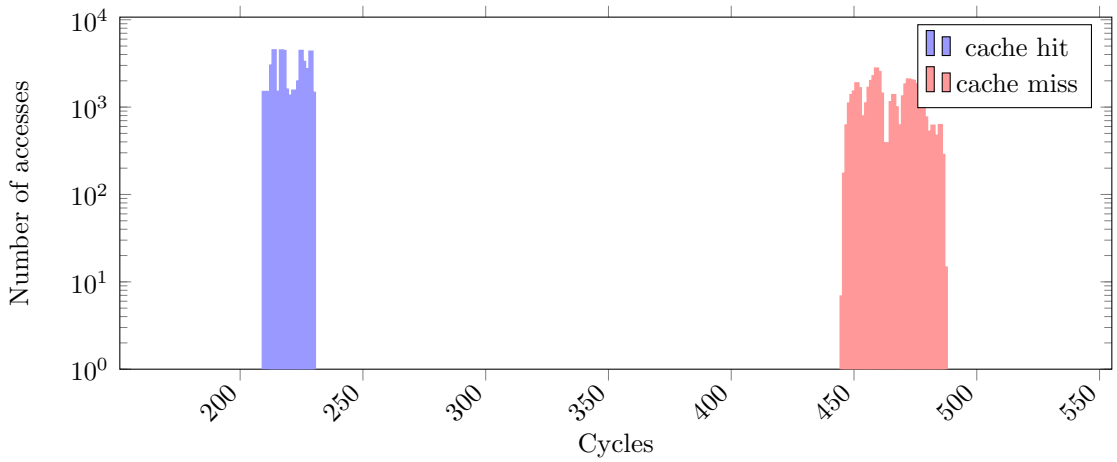


Figure 3.1: L2 cache hit and miss timings of an RTX 2070 GPU. The cache hits are at around 225 cycles, and the cache misses at around 475 cycles. There is a clear difference between the cycle counts of a cache miss and a cache hit of over 200 cycles.

and measure the access timings. Since 6 MB is significantly larger than the 4 MB available, the memory accesses should result in cache misses.

The results of these measurements are shown in Figure 3.1. We can see a clear distinction between L2 cache hits, which are located at around 200 cycles, and cache misses, which are located at around 450 cycles.

3.2 L2 Cache Timings

To measure L2 cache timings, we use our measurement approach described in Section 3.1. In Figure 3.2 we show the L2 cache access timings over a range of 128 kB on an RTX 2070 GPU starting at a 2 MB page boundary. While there is no one constant L2 cache access time, all measurements are in a 20 cycle range and depend on the address. All of our measured L2 cache timings stay the same within every 256 B block. The timings are highly consistent on all tested GPUs, even with other applications running in parallel to our measurement application. According to our observations, three measurements suffice for an accurate timing value. We assume this to be the case due to the restriction that modern Nvidia GPUs only run one application at a time on a given GPU.

Figure 3.3 contains a histogram of the L2 cache access timings from Figure 3.2. Multiple groups of 64 cache lines have the same access timing, and other groups

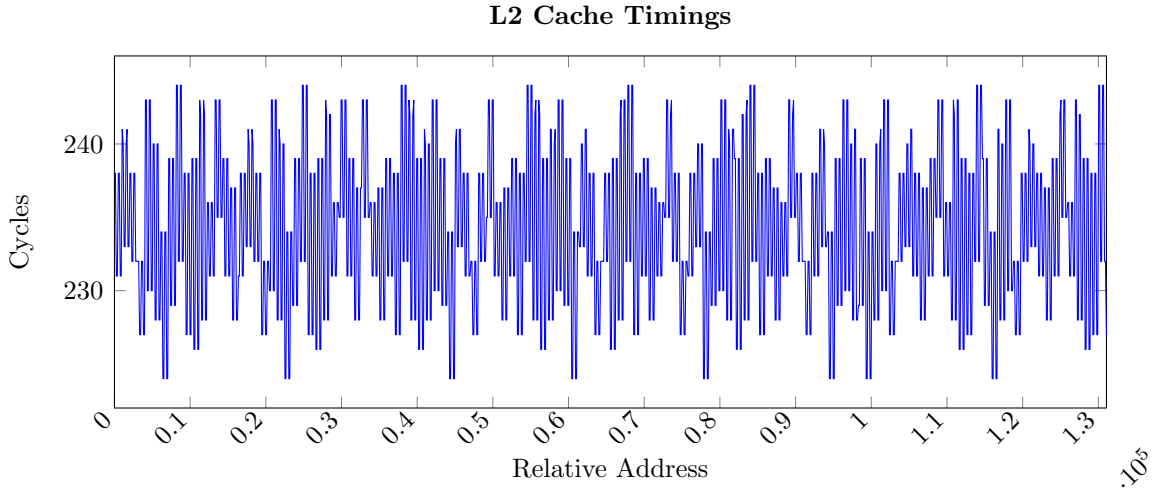


Figure 3.2: L2 cache access times over a 128 kB range on an RTX 2070. All timings are between 220 and 250 cycles with a clear address-dependent pattern. Memory blocks of consecutive 256 B or two cache lines have the exact same L2 cache access time.

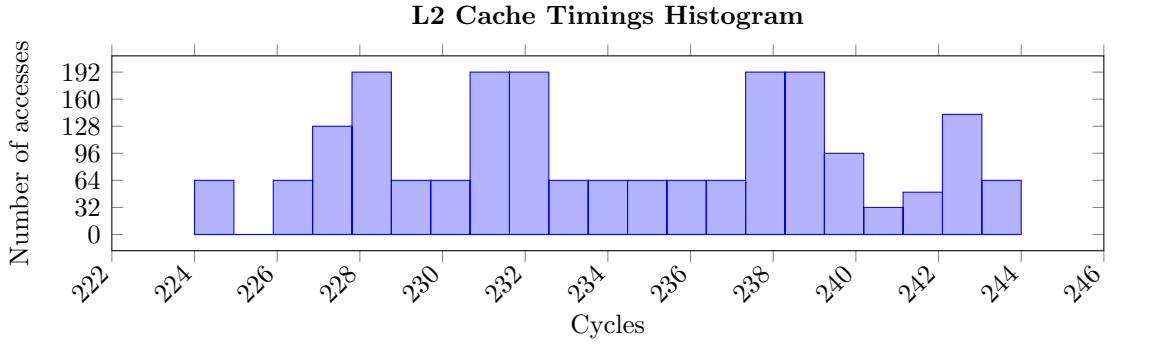


Figure 3.3: Histogram of the L2 cache access times over a 256 kB range in 128 B increments an RTX 2070. There is an almost uniform distribution of access times, with each bucket containing 64 measurements. This distribution indicates a hardware division of the cache lines into groups.

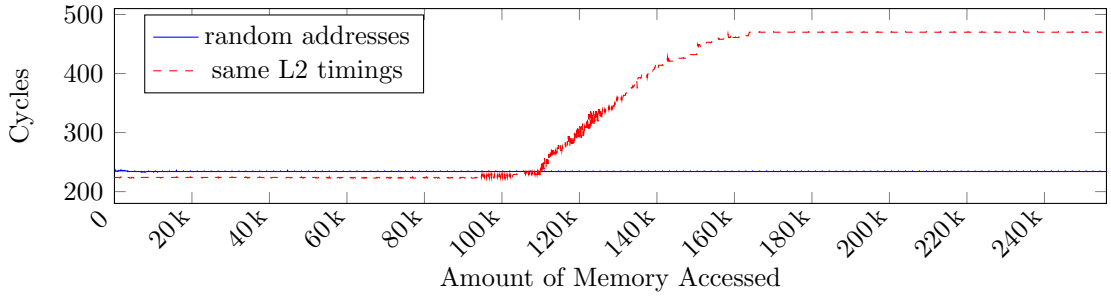


Figure 3.4: Average L2 cache access times for randomly chosen addresses and addresses from the same cache slice on an RTX 2070. For random addresses, the average access time stays constant throughout the 256 kB at around 240 cycles. For addresses with the same L2 cache timings, the average access times indicate primarily cache misses for measurements with more than 128 kB.

consist of a multiple of 64 cache lines. Due to this, we assume a division of the L2 cache into 32 equal parts, which affects the access times, similar to cache slices on modern CPUs. To confirm our assumption, we iterate over an increasing number of cache lines similarly to our previous measurements and average the access times for each iteration. We do this once for randomly chosen addresses and once for addresses with the same L2 cache access time.

The results of the measurements are shown in Figure 3.4. The average access times for random addresses stay around 235 cycles throughout the 256 kB measurement region, indicating mostly L2 cache hits. The average access times for our specifically picked addresses with the same L2 cache access times stay at 224 cycles for the first 100 kB, which indicates primarily L2 cache hits. After this, they increase drastically to around 490 cycles, indicating primarily L2 cache misses. This sudden shift in access times after only 100 kB indicates that addresses with the same L2 cache access timings map to the same part of the L2, which is full after around 128 kB on our RTX 2070. Due to the similarities of these cache line groups to cache slices on CPUs, we will refer to them as cache slices. While the knowledge of this cache split on GPUs might not be important for everyday applications, it can be beneficial for an attacker trying to fill up cache sets.

All previous measurements come from the same SM, and multiple independent measurements yield the exact same number of cycles on average when accessing the L2 cache. Figure 3.5 shows the average L2 cache access time over a 256 kB memory region for each of the 36 SMs of an RTX 2070. The exact instructions measured are a fence, a store instruction to the memory location, and another

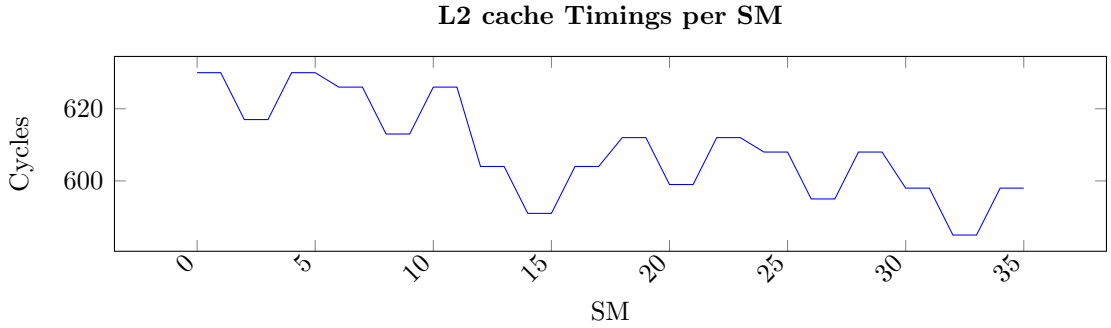


Figure 3.5: Average L2 cache access times for each SM on an RTX 2070.

fence to ensure that the store instructions are completed before the end of the measurement. According to our observations, the L2 cache access timings depend on the SM. While the access timing pattern for the addresses measured is the same on all SMs, meaning that addresses that are fast on one SM are fast on all SMs, similar to our previous measurements shown in Figure 3.2, the precise timings vary significantly between SMs.

We measured a 50 cycle difference in cache access times between the fastest and the slowest SM. To determine the exact reason for the timing differences, we measured the two fences independently without the store, as shown in Figure 3.6. Fences ensure that changes made to memory propagate through certain parts of the GPU and, for some fences, even to the host memory. In our case, we measured GPU fences, ensuring that memory changes are visible to all other SMs. As shown in Figure 3.6, the time a fence takes to propagate the changes through the whole GPU depends on the SM the fence is executed on. According to our measurements, there are groups of 6 SMs each, on which fences exhibit the exact same timing behavior.

Subtracting the fence timings from our measurements in Figure 3.5 results in a much more consistent pattern for the SMs, as shown in Figure 3.7. The average, minimum, and maximum access timings for each SM, as shown in Figure 3.7, show a repeating pattern with three different values for the minimum and maximum access timings. We measured significantly faster overall memory access timings for 12 of the 36 SMs present in our RTX 2070. The other 24 SMs have the same average L2 cache access speed but can be split into two groups of 12 when looking at their minimum and maximum access timings. Due to these highly consistent results, we assume that both the L2 cache access times and fence times correlate with the physical location of a given SM on the GPU, as shown in Figure 2.4. We assume that the groups of 12 SMs, when grouping by L2 cache timings, correlate with the GPC a given SM belongs to. We assume that the 12 fastest SMs are

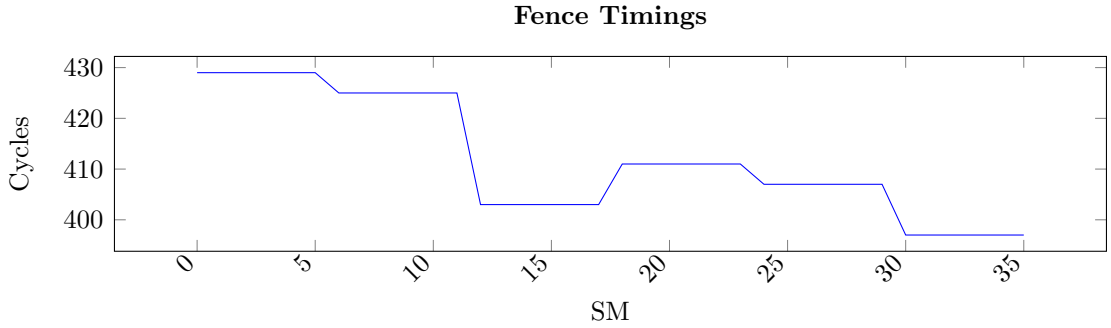


Figure 3.6: Average GPU fence timings for each SM on an RTX 2070. Groups of six SMs with consecutive IDs show the same timing behavior.

located in the middle GPC, which might have a faster connection to the L2, and the other two groups of 12 SMs are located in the two outer GPCs, due to them showing the same average access timings. Since there are always groups of two SMs with IDs next to each other that show the exact same timing behavior, we assume that they belong to the same TPC since each TPC holds exactly two SMs. The fence timings we assume correlate with the TPC an SM is located in. There are precisely 6 TPCs within a GPC, which correlates with our measurements seen in Figure 3.6 and also fits our assumption that the L2 cache timings correlate with the GPC. To confirm our assumptions, we repeated similar measurements on an RTX 3070 Ti, consisting of 6 GPCs with 4 TPCs each. Our measurements resulted in 6 groups of 8 SMs when grouping through only L2 cache access timings and 4 groups of 12 SMs when grouping by fence timings which perfectly fits the physical layout of an RTX 3070 Ti.

3.3 Eviction Set Search

A traditional Prime+Probe attack requires eviction sets that fill up the cache sets that the attacker wants to monitor. Since the GPUs L2 cache is physically indexed and physically tagged, it is not possible to directly compute the eviction set from virtual addresses alone. To combat this issue, we propose building eviction sets by carefully monitoring cache hits and misses that is similar to an existing algorithm proposed by Liu et al. [28]. Furthermore, we apply optimizations to the algorithm, which are possible due to the low noise level of cache timing measurements on GPUs. We start by allocating a big enough chunk of memory larger than the whole L2 cache. For our RTX 2070, with a 4 MB L2 cache, a memory size of 6 MB worked well for our measurements, which is exactly one memory page larger than the L2 cache. As a preprocessing step, we measure the L2 cache access speeds of all

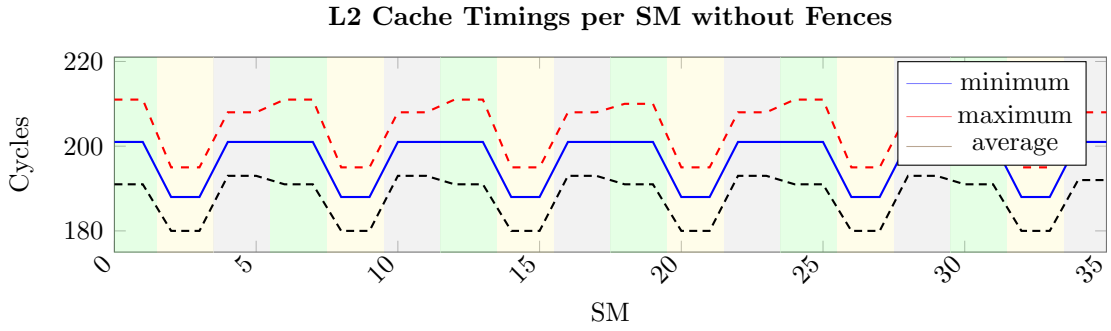


Figure 3.7: L2 cache access times minus the GPU fence timings for each SM on an RTX 2070. We can observe a clear distinction between three different equally sized groups of SMs, colored in green, yellow, and black, through the minimum and maximum timings. Pairs of two SMs with IDs next to each other show the same timing behavior.

Algorithm 3 Eviction Set Search

```
function EvictionSetSearch(slice)
    sets ← [ ]
    while size(slice) ≥ 17 do
        pool ← slice
        p ← pop random element from pool
        while size(pool) > 17 do
            el ← pop front from pool
            p_miss ← getMisses(pool, p)
            if not p_miss then
                pool ← pool + el
            end if
        end while
        if isSet(pool) then
            filterElements(slice, pool)
            sets ← sets + [pool]
        end if
    end while
return sets
end function
```

128 B chunks in our allocated memory region and sort the addresses into buckets according to them. The access time of each 128 B block correlates with its cache slice. We can decrease the amount of memory that needs to be checked at a time by grouping the memory using their L2 cache access timings and checking each group independently.

An overview of the basic eviction set search without optimizations is shown in Algorithm 3. We start by picking one of our pre-grouped memory buckets and group them into what we refer to as a search pool. All 128 B blocks within a bucket usually only belong to one cache slice. We iterate over the addresses in our chosen bucket multiple times and measure the access timings after the first iteration. Since our 6 MB memory region used for the eviction set search is significantly larger than the L2 cache, and this bucket should contain almost all memory addresses of a given slice, this should result in L2 cache misses for all addresses measured. Addresses that produce cache hits due to measurement errors in the previous steps can be filtered out and sorted back into their according buckets. After confirming that all addresses produce a cache miss when iterating over them multiple times, we choose a random pivot element. With a pivot element chosen, we remove addresses from our measurement pool one at a time. After removing an address, we search for L2 cache hits when iterating over the whole address pool multiple times. If any accesses to the current pivot element result in cache hits after removing an address, this indicates that the removed address most likely maps to the same cache set as the pivot element. In this case, we re-add the removed address back to the pool. We repeat this address removal until only 17 addresses are left in the pool, including our pivot element. These 17 addresses map to the same cache set, and 16 of them form an eviction set for the cache set. While on modern CPUs, eviction set sizes larger than the cache set size are common, the very strict LRU eviction policy on Nvidia GPUs allows for a 100% eviction rate with an eviction set that is exactly the size of a cache set. Next, we remove all addresses that are part of an eviction set from their address bucket. If an incorrect address is removed during the search for an eviction set due to a measurement error and the removal of addresses stalls, we re-add all addresses from the initial bucket, select a new pivot element and try again.

Once an eviction set is found, we use it to filter out addresses in all remaining address buckets that map to the same cache set. This filtering can be achieved by accessing the 16 addresses from the eviction set and, afterward, the address that is supposed to be checked. If, after multiple repetitions, all accesses result in cache misses, we can be certain that the address maps to the same cache set as the evictions set and remove it from its bucket. We repeat the selection of a pivot and search for an eviction set containing it until the current bucket is either empty or

smaller than a complete eviction set. Once this is the case, we move the remaining addresses to the closest bucket, on which we repeat the same process. We search the buckets in order of their L2 cache access times. This way, addresses that were sorted into an incorrect bucket during the initial sorting phase can be moved to the next best bucket.

The search algorithm can be improved further by monitoring not only the access timing of the pivot element after each iteration but the access timings of all addresses in the current search pool. We propose to check for cache hits in the remaining addresses in the pool after an address was removed and remove them as well. Since the access of these addresses resulted in a cache hit, even though the access to the pivot resulted in a cache miss, and the L2 cache follows an LRU eviction policy, it is highly unlikely that they map to the same cache set as the current pivot. Furthermore, we propose to check for the number of addresses that are removed from the search pool through this optimization. If the number of removed addresses combined with the initially removed address is 17, then they most likely map to the same cache set and can be used as an eviction set. To check if this is the case, we can iterate over these 17 addresses multiple times and measure the access time. If the accesses after the first iteration result in cache misses for all addresses, then 16 of them can be used as an eviction set. If such a set is found, we can again search for other addresses mapping to the same cache set in all the buckets and remove them from the search space.

Once all the buckets are empty, there should be one eviction set for each cache set in the L2 of the GPU. For an RTX 2070, this results in 2048 eviction sets. While DRAM accesses are extremely slow, and our search algorithm relies on achieving as many of them as possible, due to our algorithm searching through a few slices at a time and the further improvements we introduced when searching for eviction sets within a slice, recovering an eviction set for all cache sets on an RTX 2070 can be achieved within roughly 10 minutes.

Chapter 4

Covert Channel

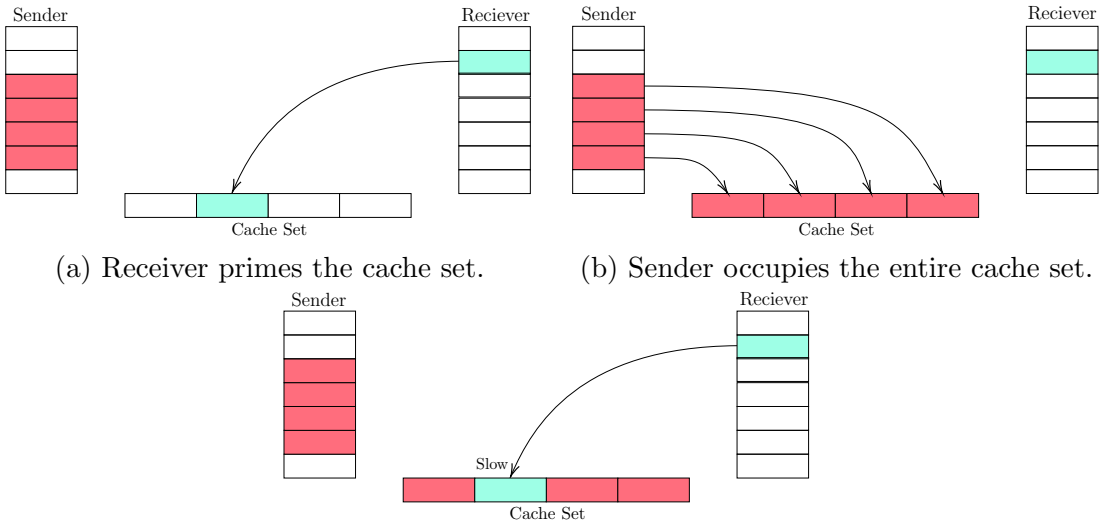
In this chapter, we propose a Prime+Probe-based cache covert channel running on modern Nvidia GPUs that works with CUDA, OpenGL, and Vulkan to transmit data between two completely independent users as long as their applications run on the same GPU. We take advantage of the L2 cache in modern Nvidia GPUs to transfer data between two GPU applications without any shared memory. We are able to achieve a transfer rate of 2.5 kB/s even though the sender and receiver cannot be scheduled on the GPU in parallel.

4.1 Transmission Protocol

The transmission of a single bit with the value 1 over a cache set is shown in Figure 4.1. The receiver loads a cache line into a cache set and regularly checks the access time. To send a 1, the sender evicts all currently present cache lines in the cache set. The receiver detects that a 1 was sent if an access to the initially loaded cache line results in a cache miss.

To transmit data, we take advantage of the eviction sets from Section 3.3. The main challenge for our covert channel is the initial connection establishment and agreement on channel parameters, namely the precise cache sets used for the connection. Since the receiver and the sender are independent applications with independent GPU virtual address spaces finding an eviction set in both applications that maps to the same cache set without any direct data exchange can be challenging. We propose a simple protocol for the transmission channel establishment.

Before a transmission can be established, both the receiver and the sender build eviction sets, with our algorithm described in Section 3.3. The receiver builds an eviction set for all cache sets on the GPU, while the sender needs an eviction set for each cache set directly used for the transmission channel. Since the sender always evicts all cache lines in a set, the receiver needs only one address per eviction set.



(a) Receiver primes the cache set.

(b) Sender occupies the entire cache set.

(c) Receiver checks the initially loaded data to receive the bit.

Figure 4.1: Transmission process of a single bit with a value of 1. First, the receiver loads a memory location into the cache. Second, the sender evicts all cache lines in the cache set. Last, the receiver checks if its memory is still in the cache.

Once the eviction sets are complete, the receiver awaits incoming connections by monitoring all cache sets for full cache-set evictions.

Sender: To start a connection, the sender picks an eviction set for sending control signals. This chosen eviction set is used for initializing the connection and confirming finished packages. The sender sends a 1 over the control set 10 times with pauses in between to allow the receiver to receive the information. Following this, the sender evicts the cache sets for the data transmission with pauses to allow the receiver to detect one complete cache eviction at a time. The delay is necessary to convey the order of the cache sets to the receiver. After this, the initial channel setup is complete, and the sender can start the data transmission. We decided to use 32 cache sets for data transmission to allow for the transmission of 4 B at a time. We use only a tiny part of the overall number of cache sets, which can be increased significantly to increase the throughput and to establish multiple connections simultaneously. To send data, the sender splits the data into 4 B chunks and transmits them one by one. For each of the 4 B the sender evicts the cache sets for which the data chunk bits are 1 and signals the control channel to let the receiver know that the package is complete. Once a chunk is sent, the

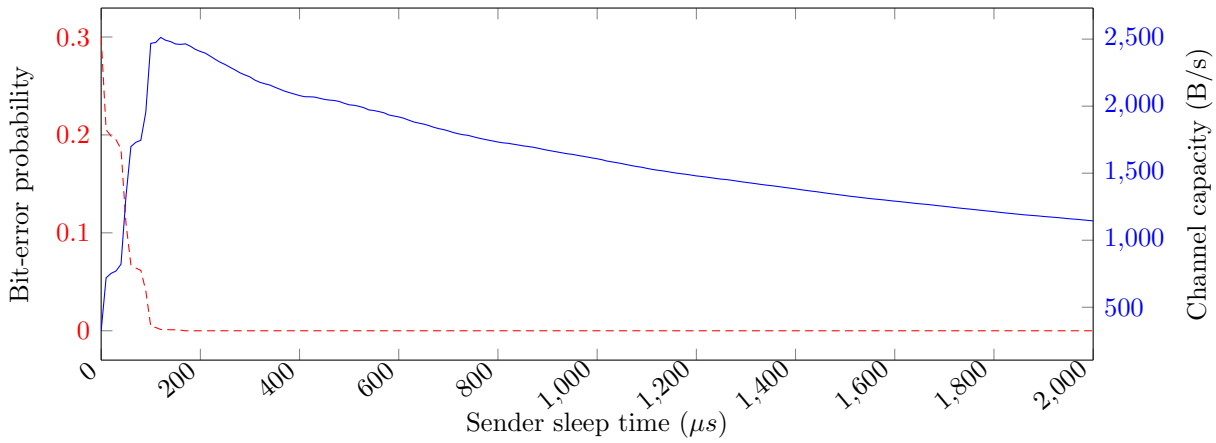


Figure 4.2: Channel capacity and bit-error rate in relation to the time the sender waits after each package transmission on an RTX 2070.

sender waits a short period, usually only a few hundred microseconds, to allow the receiver to read the data chunk. The first 8 B in a transmission consist of the number of data bytes that are going to be transmitted through this new channel. Following this is the data. Once the transmission of the last data chunk is complete, the sender stops without transmitting any termination sequence.

Reciever: The receiver awaits a connection by monitoring all L2 cache sets on the GPU. Once the receiver detects 10 complete cache-set eviction on the exact same cache set in succession without any other complete set evictions, it saves the corresponding eviction set as the control set. Following the control set, the receiver expects 32 complete cache-set evictions from distinct cache sets precisely one at a time. The evicted cache sets are saved as the data transmission sets in order of their detection. If, at any time during this initialization step, more than one complete cache set is evicted at a time, the receiver transitions into the initial state and waits again for incoming connections. With the setup complete, the receiver starts receiving data packages. To receive data, the reciever checks for full cache-set evictions of the data sets and the control set. Initially, a package is assumed to be 0. If a data set is evicted, the corresponding bit is changed to a 1. When a complete eviction of the control set is detected, the package is complete, the data is saved, and the reciever starts their measurements for the next data package. The first 8 B are interpreted as the overall transmission size and determine how much data the receiver expects. Following is the actual data. Once the initially agreed-upon number of bytes is received, the receiver transitions back to listening for incoming connections.

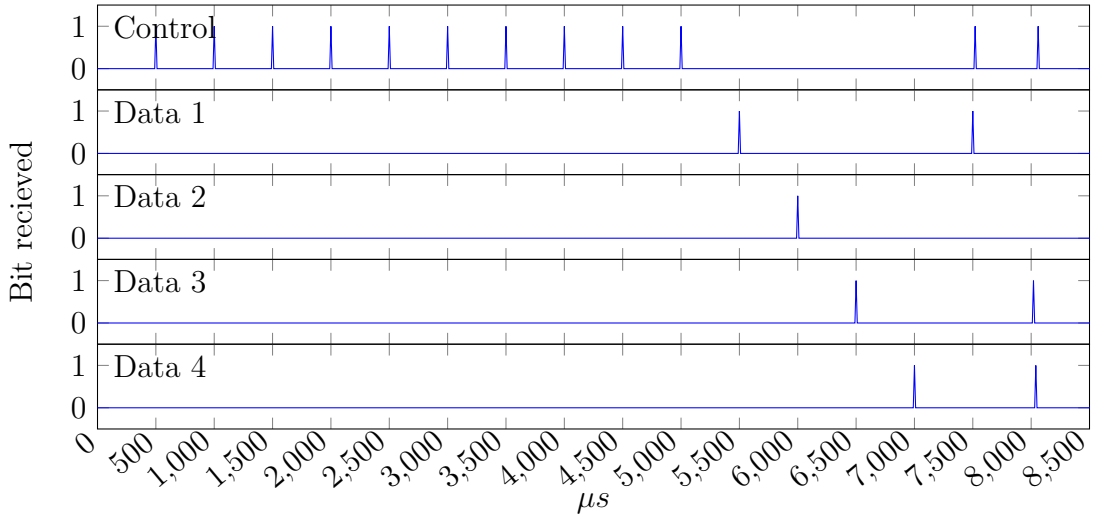


Figure 4.3: Covert channel example transmission. The sender pulses a 1 ten times on the control set to initialize the connection, followed by a 1 on the four data transmission sets. Afterward, two 4 bit packages are sent by pulsing the channels corresponding to a bit value of 1 and the control set.

The time the sender waits after transmitting a package significantly impacts the transmission rate and the package loss. The bit-error rate of our covert channel in relation to the sender's wait time on an RTX 2070 is shown in Figure 4.2. The error rate decreases from 0.3 at a wait time of 0 µs to an error rate of 0 at 170 µs. All wait times higher than 170 µs result in no bit-errors. The channel capacity C can be calculated with

$$C = T \cdot (p \log_2 p + (1-p) \log_2 (1-p) + 1)$$

where T is the transmission rate, and p is the bit-error probability [6]. The channel capacity of our covert channel compared to the bit-error rate with different sender wait times is shown in Figure 4.2. We keep the wait times for the initial channel setup constant at 200 ms since the receiver has to scan all cache sets in the setup phase, and this wait time worked consistently throughout our measurements. The channel capacity peaks at 120 µs with 2.5 kB/s with a bit-error rate of 0.134%. Afterward, the channel capacity decreases due to the higher wait time of the sender. We use a wait time of 120 µs since it results in the highest channel capacity.

An example of a data transmission on a simplified version of our covert channel with 4 cache sets for data transmission is shown in Figure 4.3. The start of the initialization sequence can be seen in the first cache set, where a 1 is sent 10 times

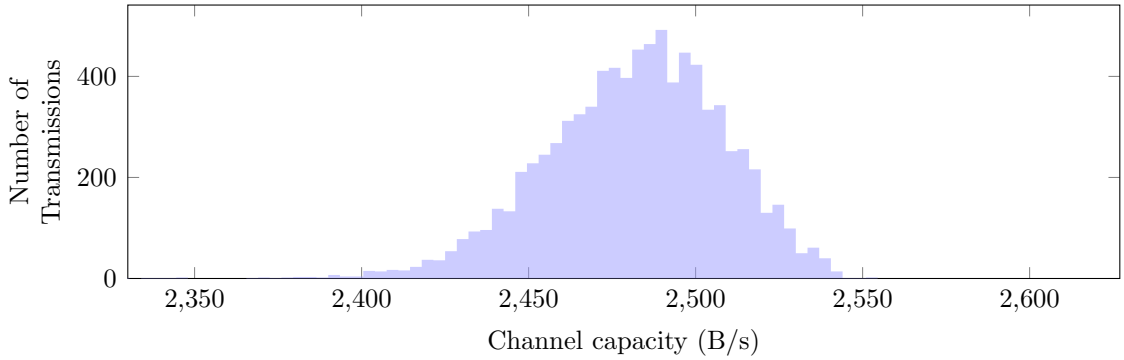


Figure 4.4: Histogram of the cache capacity of our covert channel over 8000 transmissions. Our covert channel has a capacity of 2480.7 ± 52.3 B/s.

in a regular interval to indicate the beginning of a transmission. 10 complete evictions of only one particular cache set in a short period is a pattern that is extremely unlikely to happen in a normal program, making it a perfect indicator for a starting connection. Right afterward, the other four cache sets used for data transmission are evicted one at a time, with a pause in between to give the receiver time to check the GPUs cache. Next, two packages are sent. The first package with a value of 1 (0b0001), where only the first data set is evicted, and later the control set, which finishes the package transmission. The second package with a value of 12 (0b1100), where first the third data set is evicted, then the fourth data set, and at the end, the control set to finish the package transmission.

4.2 Evaluation

All our measurements were taken in an environment where no other applications were running on the GPU at the time of transmission. Due to the sender waiting after every transmitted package and the receiver constantly probing the cache, there is no extra need for extra synchronization besides the control channel to indicate a finished package. We achieve data transmission rates of 2480.7 ± 52.3 B/s after measuring 8000 independent data transmissions, as shown in Figure 4.4. We use 33 out of 2048 cache sets available on our RTX 2070 and RTX 3070 Ti. The overall transmission speed could be significantly increased by employing more cache sets to allow for parallel data transmission and switching to an acknowledge-based transmission protocol. While significantly higher transmission rates are possible, our covert channel proves that data transmission at a high rate through cache evictions on modern Nvidia GPUs is possible.

Chapter 5

AES Key Recovery

In this chapter, we propose an attack on an AES T-table implementation written in CUDA. Due to the high parallel throughput achievable on GPUs, GPGPU is becoming more and more viable for encrypting enormous amounts of data and can even outperform hardware-accelerated AES on modern x86 CPUs given enough data [29].

First, we show how it is possible to build eviction sets for the T-tables of an AES implementation on a GPU without access to physical addresses or the address space of the target application. For this, we propose Prime+Count, a more powerful version of Prime+Probe that takes advantage of a predictable LLC replacement policy to recover the exact number of cache lines evicted by the victim.

Second, we propose a simple first-round attack that recovers 48 bits of the full 128 bit AES-128 key. We achieve this with a known-plaintext attack by monitoring memory accesses to the T-tables and inferring the key bits through them.

Finally, we present a full AES-128 key-recovery attack using Prime+Count on a GPU with the target and attacker code running in two completely different processes under two separate users. We compare our Prime+Count implementation with Prime+Probe and discuss our new attack's advantages and disadvantages.

5.1 Eviction Set Search

To perform an attack on T-tables, we first need to find eviction sets that cover the cache sets of the target application's T-tables. Since direct access to the target application's physical addresses is impossible, we propose an online approach to search for the required eviction sets. We assume that we can request encryptions from the victim with an attacker-provided key and plaintext. We leverage this encryption oracle to create known accesses to certain parts of the T-table.

We start by building eviction sets using our algorithm described in Section 3.3. A typical T-table implementation uses four tables consisting 256 distinct 32 bit values each. Since the cache-line size on recent Nvidia GPUs is 128 B, this means that eight cache lines cover one table assuming the tables are aligned to 128 B. Usually, memory addresses next to each other map to different cache sets, meaning we need eight eviction sets per table.

To simplfy our search, we use a zero key for the encryptions. We target the T-table accesses for the first round of AES. Since we use a zero key, the first round key is also zero. With a zero first-round key, each T-table access in the first round directly uses the plaintext bytes as indices.

We construct the plaintext for an encryption by setting a byte that is used with the target T-table to a value from 0 to 255 and randomizing the rest. For the 0th T-table, this would be the 0th plaintext byte. Before each encryption, we load all members of the eviction sets into the L2 cache by accessing them. Next, we request an encryption for our constructed plaintext. Once the encryption is complete, we access each eviction set in reverse order and measure the L2 access timings. We can determine through the access timings how many cache lines were evicted. This is possible due to the strict LRU L2 cache eviction policy. By accessing the eviction set members in reverse order, we avoid evicting further elements of the eviction set while being able to determine the exact number of evicted cache lines. We call this technique Prime+Count. Contrary to traditional Prime+Probe, where only the access to a cache set is detected, Prime+Count can recover the exact number of cache lines loaded by a victim. Prime+Count requires less fine-tuning of the eviction set than Prime+Probe while providing more information about the victim's execution. The number of evicted eviction set members is saved together with the byte chosen for the plaintext. We repeat this a few hundred times for each chosen byte value.

If an eviction set does **not** map to a cache set of the target T-table, the average number of evictions stays constant independent of the chosen plaintext byte. If an eviction set does map to the same cache set as a part of the target T-table, the average number of evictions is higher for some chosen plaintext byte values and lower for others. For the correct eviction sets, there should always be a continuous range of 32 plaintext byte values for which the number of average evictions is slightly higher than for other values. Examples of this are shown in Figure 5.1. This figure shows the average miss rate of eight eviction sets after a few thousand encryptions for each plaintext byte. The first eviction set has an average miss rate that is 0.005 higher for the plaintext bytes 0 to 31, indicating that it maps to the same cache set as the first 128 B of the target T-table. The second eviction set has an average miss rate that is 0.005 higher for the plaintext bytes 32 to 61, indicating

that it maps to the same cache set as the second 128 B of the target T-table. The probability that the target part of a specific T-table is not accessed each time it is used is $\frac{7}{8}$, and assuming that we only control one access throughout the whole AES encryption and that all others are random, this gives us $\left(\frac{7}{8}\right)^{4*9+3} = 0.00547$ as the probability that the monitored T-table part has not been accessed by any other part of the encryption. This probability fits our measured miss rate difference of 0.005. On a typical modern x86 CPU with a cache line size of 64 B the probability is $\left(\frac{15}{16}\right)^{4*9+3} = 0.0807$.

To find these patterns for each of the 32 plaintext value chunks, we compute an average over the average miss rate of the target 32 value area and an average over all other plaintext miss rates. We use the difference between these two averages for each of our 2048 eviction sets as a metric to determine the correct eviction sets. A good eviction set candidate for the target T-table part is the eviction set, where this difference is the largest. We compute the currently best eviction set candidates for the target T-table every 100 encryptions for each plaintext byte. Once the candidates converge to eight distinct eviction sets, we assume to have found the correct eviction sets. We repeat this process for each of the four T-tables using a different part of the plaintext for the chosen plaintext byte values. This results in eight eviction sets for each of the four T-tables.

5.2 First-Round Attack

In this section, we demonstrate a first-round attack on an AES-128 T-table implementation running on a GPU using Prime+Count. We demonstrate that it is possible to recover 48 bits of the 128 bit AES key from a completely different process only using known plaintext. To achieve this, we leverage our eviction sets found in Section 5.1 and measure multiple AES-128 encryptions with known plaintext but unknown key. By measuring the number of evictions in each set and combining it with the known plaintext for each of the encryptions, we are able to infer 3 bits of each key byte.

To start our attack, we build eviction sets for all four AES T-tables using our method described in Section 5.1. After the setup is complete, we start measuring encryptions. Similar to our eviction set search, we first load all eviction set members into the L2 cache of the target GPU. Contrary to the eviction set search in Section 5.1, there are only eight eviction sets for each T-table left, speeding up measurements significantly. Once all eviction set members are loaded, we wait for an encryption. After an encryption, we use Prime+Count and save the number of evictions for each set together with the plaintext used for the encryption. We repeat

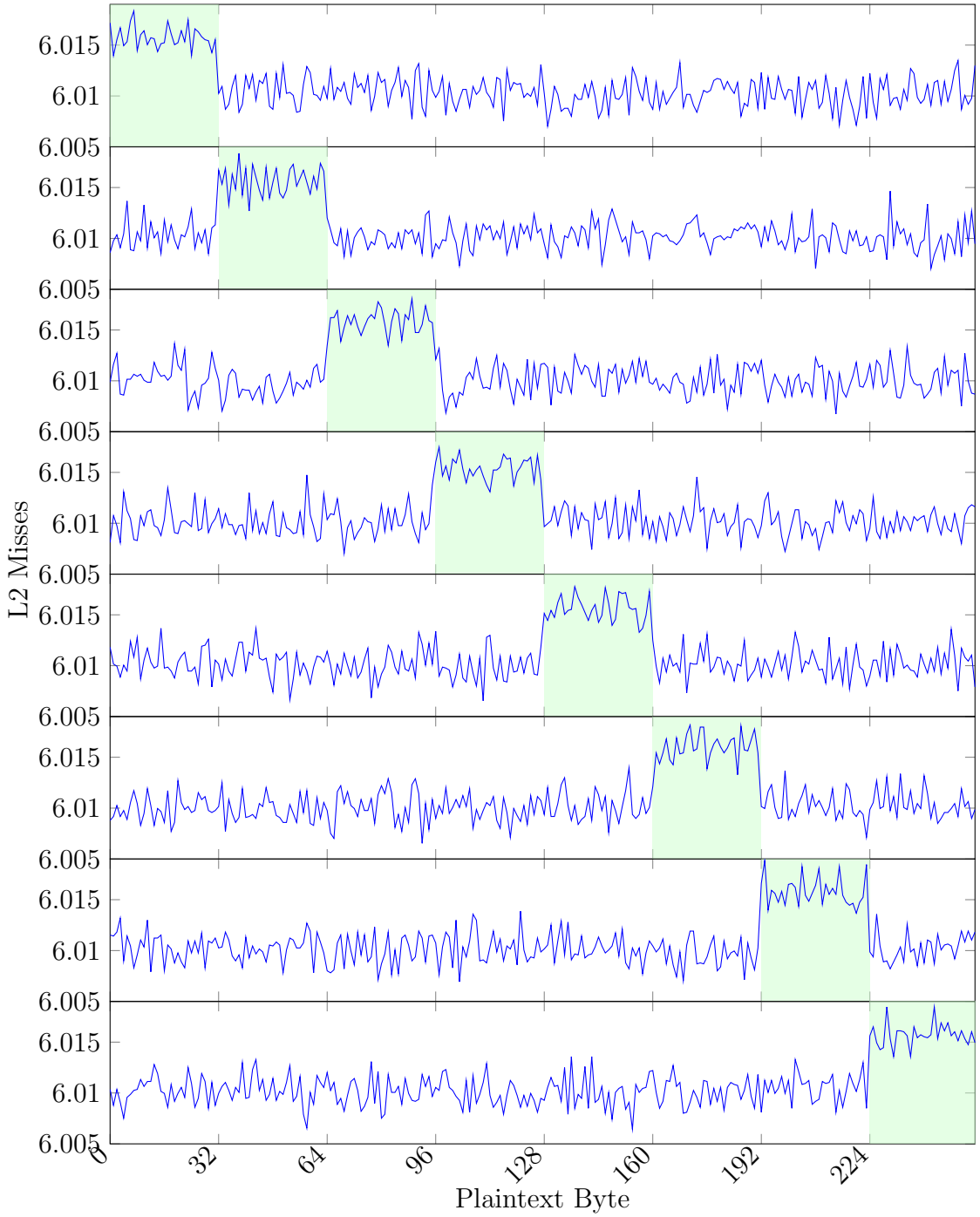


Figure 5.1: The average number of evicted cache lines of eight eviction sets that cover one whole AES T-table in relation to the chosen plaintext byte. The evictions for each set are 0.005 for the part of the T-table that the eviction set covers.

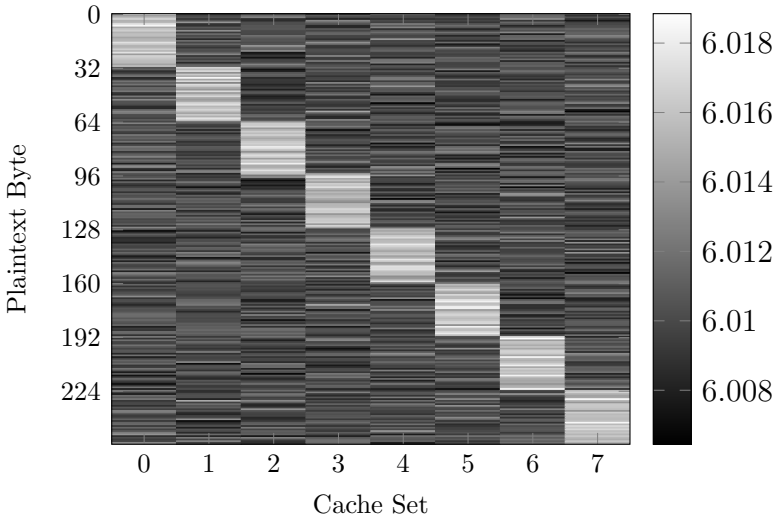


Figure 5.2: Eviction set cache miss heatmap for a key where the upper 3 bits of a key byte are zero. The average miss rate is 0.005 higher when the 3 most significant bits of the plaintext byte are the same as the cache set ID.

these measurements and collect plaintext and cache miss values of encryptions using the unknown target key.

Each T-table is indexed by four plaintext bytes XORed with the corresponding key byte independently in the first round. Since the cache lines on modern Nvidia GPUs have a size of 128 B, it is only possible to detect which part of the target T-table a given encryption accessed. This means that it is only possible to recover 3 bit of each key byte, and the recovery can only be done on a key byte granularity.

To recover a part of the key, we use the plaintext byte that is XORed with the target key byte and its corresponding T-table eviction sets. We start by sorting the encryptions into 256 buckets using the plaintext bytes value. Next, we compute the average cache misses of each possible plaintext byte for each eviction set. For each plaintext value block of 32 values, we compute the average cache misses for the block and subtract it from the average number of cache misses for all other plaintext byte values. We repeat this for each of the eight eviction sets. Afterward, we search for the eviction sets that maximize this difference for each block. Since there are precisely eight eviction sets per T-table and eight blocks for which we want to find the maximum difference, each eviction set should be the maximum for precisely one of these differences.

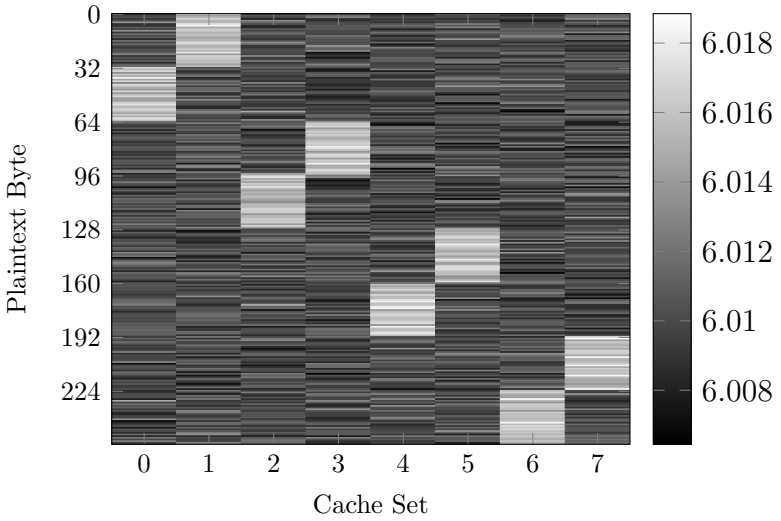


Figure 5.3: Eviction set cache miss heatmap for a key where only bit 5 (0b0010000) of the key byte is set.

The key bits can be extracted from the correlation between the eviction sets and the plaintext bytes. Figure 5.2 shows the average evictions of our eight eviction sets for each key byte for a zero key. The x-axis shows which 8th of the T-table corresponds to which eviction set, and the y-axis shows the plaintext byte. With a zero key, the plaintext byte is directly used as an index into the T-table. Therefore, resulting in a clear correlation between the plaintext chunk that maps to an eviction set and the part of the T-table the eviction set covers.

Figure 5.3 shows a key where in the upper 3 bits of the targeted key byte, only the least significant bit is set (0b00010000). We can still observe plaintext byte chunks of 32 values that map to a distinct eviction set, but the mapping is not the same as the mapping to the T-table part covered by the eviction set, as shown in Figure 5.2. The key byte XOR the plaintext byte is used to index into the T-table, and the cache miss rates reflect this correlation. When looking at the first part of the T-table, it is accessed for plaintext bytes 32-63 with the non-zero key instead of the zero-key range of plaintext bytes 0-31. The key-byte bits can be recovered by XORing the plaintext byte range, a T-table part is accessed in, with the part it covers of the T-table.

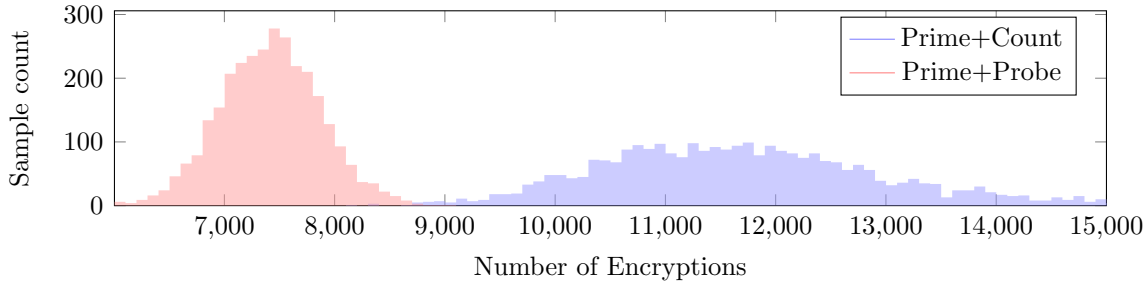


Figure 5.4: Distribution of the number of encryptions measured until the correct last round key is recovered over three thousand attacks. The attack requires 11773 ± 1443 encryptions to recover the full key for Prime+Count and 7466 ± 451 for Prime+Probe.

5.3 Last-Round Attack

In this section, we propose a last-round attack on an AES-128 T-table implementation running on a GPU using Prime+Count. We show that it is possible to recover the full 128 bit AES key from a different process only using known ciphertext.

Contrary to the first-round attack from Section 5.2, we propose a last-round attack that is able to recover the complete last-round key. Our attack is based on Briongos et al. [4]. While the last-round key of AES-128 is not the key itself, it can be trivially reversed to the key as long as the whole last-round key is known [7].

Similar to the first-round attack, we start the attack by building eviction sets for all four AES T-tables using our method, as described in Section 5.1. Once the setup is complete, we measure encryptions similar to Section 5.1 and store the number of evictions for each cache set together with the ciphertexts until a sufficient amount of encryptions were measured. If the recovered encryption key is incorrect, the result can be refined by measuring more encryptions. Our attack requires 11773 ± 1443 encryptions to recover the full key. The exact distribution of the number of encryptions required is shown in Figure 5.4.

In the first-round attack, we go forward through the AES encryption to the beginning of the first round directly after the first-round key was added and, therefore, to the first T-table access. In the last-round attack, we go back from the end of the AES encryption one round to the beginning of the last round and the corresponding T-table accesses. We do this one byte at a time to recover the whole last-round key.

To recover a key byte, we first guess the byte. We then go through all measurements and reverse the last round for the ciphertext byte that corresponds to the position

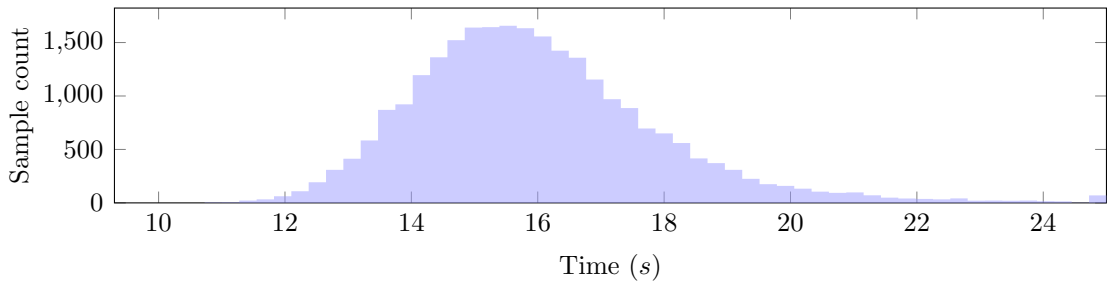


Figure 5.5: Distribution of the time until the correct last round key is recovered with Prime+Count over 25 thousand attacks. The attack requires $15.988 \pm 1.960s$ to recover the full key.

of key byte guessed. This can be done by calculating the XOR of the key byte guess and the ciphertext byte and applying the inverse AES S-Box to the result. The resulting value is our guess for the byte used to index a T-table. Next, we sort the measured cache misses from the eviction sets corresponding to the targetted T-table according to the computed byte. Similar to the first-round attack discussed in Section 5.2, we average the cache misses for each eviction set. A correct key byte guess looks similar to our measurements from the initial eviction set filtering shown in Figure 5.1, with the byte used to index the T-table corresponding to the plaintext byte. If the key byte is correct, on average, there should be more cache misses in the value range covered by the eviction set. We can leverage this fact as a metric to determine which key byte is the most likely to be correct. We take the average miss rate for the value range an eviction set covers and the rest and compute the difference. We sum up the resulting eight differences and use them as a metric for how likely the guessed key byte is correct. We repeat this process for the remaining possible key byte values. The key byte guess with the highest accumulated difference is the correct key.

We repeat the key byte recovery for all last-round key bytes to obtain the complete last-round key. To obtain the AES-128 key, we take the recovered last-round key and reverse the key schedule. The recovered key can be checked for correctness if a plaintext-ciphertext-pair is known. If the key is incorrect, the key guess can be refined by measuring more encryptions and repeating the offline recovery phase. The initial setup of searching for the correct eviction sets, described in Section 5.1, can take one to two hours. The key recovery takes $15.988 \pm 1.960s$ due to the small number of eviction sets that need to be monitored during the attack and the small amount of ciphertexts required to recover the key. The exact runtime distribution is shown in Figure 5.5. In Figure 5.4 we compare our last-round attack using Prime+Count with Prime+Probe. We simulate Prime+Probe with Prime+Count and a threshold of 5 evictions. If more than 5 cache lines are evicted, we count it

as an access when using Prime+Probe. This results in Prime+Probe requiering 7466 ± 451 encryptions and Prime+Count requiring 11773 ± 1443 encryptions. Prime+Count requires slightly more encryptions than Prime+Probe in our test. While Prime+Count is an inherently stronger attack than Prime+Probes, since it can provide significantly more information, our use of averages makes the last-round attack more susceptible to outliers. The advantage of Prime+Count in this attack is that it requires no fine-tuning of the eviction sets compared to Prime+Probe, simplifying the initial setup.

Chapter 6

Conclusion

In this thesis, we showed that cache attacks on modern Nvidia GPUs, even in a single GPU environment, are both possible and practical. We show that there is a clear correlation between the cache layout and cache access timings on Nvidia GPUs. By taking advantage of this correlation and the low noise of cache access timings, we were able to create an algorithm for building eviction sets that can find an eviction set for all L2 cache sets within a matter of minutes. Furthermore, we took a closer look at fence timings and cache access timings and found a correlation with the physical layout of Nvidia GPUs. We built a Prime+Probe-based cache covert channel that can transfer data at a speed of 2.5 kB/s. Our covert channel works on a single GPU between two completely independent applications running under different users. Finally, we proposed a Prime+Count-based first-round attack that can recover 48 bits of the AES-128 key and a last-round attack that can recover the full 128 bit AES key of an AES-128 implementation running on Nvidia GPUs. The last-round attack requires 11773 ± 1443 encryptions and $15.988 \pm 1.960s$ to recover the full key.

Bibliography

[1] AMD. *ROCm Documentation*. 2022. URL: https://rocmdocs.amd.com.

[2] Andrea Arcangeli, Izik Eidus, and Chris Wright. "Increasing memory density by using KSM." In: *Proceedings of the Linux Symposium*. 2009.

[3] Dennis Bazow, Ulrich Heinz, and Michael Strickland. "Massively parallel simulations of relativistic fluid dynamics on graphics processing units with CUDA." In: *Computer Physics Communications* 225 (2018), pp. 92–113.

[4] Samira Briongos, Pedro Malagón, Juan-Mariano de Goyeneche, and Jose M Moya. "Cache misses and the recovery of the full AES 256 key." In: *Applied Sciences* (2019).

[5] KVM contributors. *Kernel-based Virtual Machine*. 2019. URL: https://www.linux-kvm.org.

[6] T. M. Cover and J. A. Thomas. *Elements of information theory*. John Wiley & Sons, 2012, p. 46.

[7] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. 2013.

[8] Sankha Baran Dutta, Hoda Naghibijouybari, Arjun Gupta, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. "Spy in the GPU-box: Covert and Side Channel Attacks on Multi-GPU Systems." In: *arXiv:2203.15981* (2022).

[9] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks." In: *USENIX Security Symposium*. 2018.

[10] Khronos Group. *OpenCL*. 2022. URL: https://www.khronos.org/opencl/.

[11] Khronos Group. *SPIR-V Specification*. 2022. URL: https://registry.khronos.org/SPIR-V/specs/unified1/SPIRV.html.

[12] Khronos Group. *The OpenGL Shading Language*. 2022. URL: https://registry.khronos.org/OpenGL/specs/gl/GLSLangSpec.3.30.pdf.

[13] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. "Flush+Flush: A Fast and Stealthy Cache Attack." In: *DIMVA*. 2016.

[14] Omid Hajihassani, Saleh Khalaj Monfared, Seyed Hossein Khasteh, and Saeid Gorgin. "Fast AES implementation: A high-throughput bitsliced approach." In: *IEEE Transactions on Parallel and Distributed Systems* 30.10 (2019), pp. 2211–2222.

[15] Ralf Hund, Carsten Willems, and Thorsten Holz. "Practical Timing Side Channel Attacks against Kernel Space ASLR." In: *S&P*. 2013.

[16] Khronos Group Inc. *Vulkan Documentation and Extensions*. 2022. URL: `https://registry.khronos.org/vulkan/specs/1.3/styleguide.html`.

[17] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. 2019.

[18] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "Systematic reverse engineering of cache slice selection in Intel processors." In: *Euromicro DSD*. 2015.

[19] Keisuke Iwai, Naoki Nishikawa, and Takakazu Kurokawa. "Acceleration of AES encryption on CUDA GPU." In: *International Journal of Networking and Computing* 2.1 (2012), pp. 131–145.

[20] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. "Dissecting the NVidia Turing T4 GPU via microbenchmarking." In: *arXiv:1903.07486* (2019).

[21] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. "A novel side-channel timing attack on GPUs." In: *VLSI*. 2017.

[22] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. "Exploiting bank conflict-based side-channel timing leakage of gpus." In: *ACM TACO* (2019).

[23] Elmira Karimi, Zhen Hang Jiang, Yunsi Fei, and David Kaeli. "A timing side-channel attack on a mobile gpu." In: *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE. 2018, pp. 67–74.

[24] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution." In: *S&P*. 2019.

[25] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. "Spectre Returns! Speculation Attacks using the Return Stack Buffer." In: *WOOT*. 2018.

[26] Zhen Lin, Utkarsh Mathur, and Huiyang Zhou. "Scatter-and-gather revisited: High-performance side-channel-resistant AES on GPUs." In: *Workshop on General Purpose Processing Using GPUs*. 2019.

[27] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown: Reading Kernel Memory from User Space." In: *USENIX Security Symposium*. 2018.

[28] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. "Last-Level Cache Side-Channel Attacks are Practical." In: *S&P*. 2015.

[29] Jianwei Ma, Xiaojun Chen, Rui Xu, and Jinqiao Shi. "Implementation and evaluation of different parallel designs of AES using CUDA." In: *International Conference on Data Science in Cyberspace (DSC)*. 2017.

[30] Svetlin A Manavski. "CUDA compatible GPU as an efficient hardware accelerator for AES cryptography." In: *International Conference on Signal Processing and Communications*. 2007.

[31] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. "Reverse Engineering Intel Complex Addressing Using Performance Counters." In: *RAID*. 2015.

[32] Microsoft. *High-level shader language (HLSL)*. 2022. URL: https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl.

[33] Hoda Naghibijouybari, Khaled N Khasawneh, and Nael Abu-Ghazaleh. "Constructing and characterizing covert channels on gpgpus." In: *MICRO*. 2017.

[34] Naoki Nishikawa, Hideharu Amano, and Keisuke Iwai. "Implementation of bitsliced AES encryption on CUDA-enabled GPU." In: *International Conference on Network and System Security*. 2017.

[35] NVIDIA. *CUDA C++ Programming Guide*. 2022.

[36] NVIDIA. *Kernel Profiling Guide*. 2022. URL: https://docs.nvidia.com/nsight-compute/pdf/ProfilingGuide.pdf.

[37] NVIDIA. *NVIDIA AMPERE GA102 GPU ARCHITECTURE*. 2018. URL: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf.

[38] NVIDIA. *NVIDIA TURING GPU ARCHITECTURE*. 2018. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[39] NVIDIA. *Parallel Thread Execution ISA*. 2022. URL: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[40] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: the Case of AES." In: *CT-RSA*. 2006.

[41] M Mazhar Rathore, Hojae Son, Awais Ahmad, and Anand Paul. "Real-time video processing for traffic control in smart city using Hadoop ecosystem with GPUs." In: *Soft Computing* 22.5 (2018), pp. 1533–1544.

[42] 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*. 2019.

[43] 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.

[44] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. "NetSpectre: Read Arbitrary Memory over Network." In: *ESORICS*. 2019.

[45] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection." In: *S&P*. 2020.

[46] Carl A Waldspurger. "Memory Resource Management in VMware ESX Server." In: *ACM SIGOPS Operating Systems Review* 36.SI (2002), pp. 181–194.

[47] Yuval Yarom and Katrina Falkner. "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack." In: *USENIX Security Symposium*. 2014.