Roland Czerny, BSc

Exploiting GPU Caches from the Browser with WebGPU

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur(in)

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Advisors

Daniel Gruss
Lukas Giner

Institute of Applied Information Processing and Communications

Graz, May 2024

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

I would like to express my gratitude to my advisors, Lukas Giner and Daniel Gruss, for their support and feedback throughout the development of this thesis. Special thanks to Fabian, whose master's thesis on GPU side channels was instrumental in helping me become acquainted with the topic.

I am deeply thankful to my friends and family for their constant support. A heartfelt thank you to my parents, my brother Flo, and my grandparents. I feel incredibly privileged to be surrounded by such a loving family.

Lastly, a special thank you goes to Lisa for her unwavering love and support. Her determination is truly inspiring, and her constant encouragement and patience have been essential in making this work possible.

Abstract

The use of GPUs for general-purpose computations has increased steadily in recent years. The massive parallelism of GPUs benefits many applications, including security-critical computations such as AES. Since computations on sensitive data have become more common on GPUs, the scrutiny must also increase. Meanwhile, WebGPU facilitates easy access to compute shaders from every web browser. Previous studies have shown that GPUs are susceptible to several cache-based side-channel attacks originally designed for CPUs. Prior work demonstrated Prime+Probe attacks in various scenarios from native code.

This work presents the first side-channel attack from within the browser using the WebGPU API. We construct several generic primitives to build a self-configuring attack that works across various different devices. We present a technique to distinguish L2 cache hits from cache misses in WebGPU, an essential building block for every cache attack. We use this building block to automatically detect parameters like the cache hit-miss threshold, the L2 cache size, and the number of cache sets. We demonstrate the effectiveness of these primitives on 12 desktop GPUs from 5 different generations and 2 vendors.

A crucial step to mount a Prime+Probe cache attack is to find eviction sets. We present the first parallel eviction-set-finding algorithm. The algorithm is tailored to GPUs and reliably finds more than 80 % of sets on all but one tested Nvidia GPU. We find the sets in 2 to 12 minutes, depending on the GPU.

We further evaluate a native-to-browser data-exfiltration scenario. For this, we construct the first Prime+Probe covert channel from a native CUDA application to a WebGPU application in the browser. Our covert channel reaches a true channel capacity of up to 10.9 kB/s. The self-configuring nature of our algorithms and the brief time frame of under 15 minutes enable drive-by attacks during internet browsing. Our attack requires no user interaction and works across a variety of Nvidia GPUs.

The content of this thesis was a major contribution to a conference paper at AsiaCCS'24 by Giner et al. [1]. The paper demonstrates more practical attacks using our generic primitives. The attacks presented in the paper include an inter-keystroke timing attack, which exploits timing variations between keystrokes, and an end-to-end attack that compromises the entire AES encryption process running natively in CUDA.

Our work demonstrates that access to GPUs from the browser can be a powerful tool for attackers. We suggest that browsers treat access to the GPU like access to other security- and privacy-related resources, like the microphone or camera.

Keywords: Security · GPU · WebGPU · Side-channel

Kurzfassung

Die Verwendung von GPUs für allgemeine Berechnungen hat in den letzten Jahren stetig zugenommen. Der ausgeprägte Parallelismus von GPUs bringt vielen Anwendungen, einschließlich sicherheitskritischer Berechnungen wie der AES-Verschlüsselung, erhebliche Vorteile. Mit der steigenden Nutzung von GPUs für die Verarbeitung sensibler Daten wird eine intensivere Überprüfung immer wichtiger. Zugleich ermöglicht WebGPU einen einfachen Zugriff auf Compute-Shader in jedem Webbrowser. Frühere Studien haben die Anfälligkeit von GPUs für verschiedene cache-basierte Seitenkanalangriffe aufgezeigt, die ursprünglich für CPUs entwickelt wurden. Dabei wurden Prime+Probe-Angriffe in nativen Code-Umgebungen demonstriert.

Diese Arbeit präsentiert den ersten Seitenkanalangriff innerhalb des Browsers unter Verwendung der WebGPU API. Wir konstruieren mehrere generische Primitive, um einen selbstkonfigurierenden Angriff zu bauen, der auf verschiedenen Geräten funktioniert. Wir präsentieren eine Technik zur Unterscheidung von L2-Cache-Treffern und Cache-Verfehlungen in WebGPU, einem grundlegenden Element für jeden Cache-Angriff. Diesen Baustein nutzen wir, um automatisch Parameter wie den Grenzwert für Cache-Treffer und -Verfehlungen, die Größe des L2-Caches und die Anzahl der Cache-Sets zu bestimmen. Wir demonstrieren die Wirksamkeit dieser Primitive auf 12 Desktop-GPUs aus 5 verschiedenen Generationen und 2 Herstellern.

Ein entscheidender Schritt, um einen Prime+Probe Cache-Angriff zu starten, ist das Finden von Eviction-Sets. Wir präsentieren den ersten parallelen Algorithmus zum Finden von Eviction-Sets. Der Algorithmus ist auf GPUs zugeschnitten und findet zuverlässig mehr als 80 % der Sets auf allen bis auf eine der getesteten Nvidia GPUs. Wir finden die Sets in 2 bis 12 Minuten, je nach GPU.

Wir analysieren weiterhin ein Szenario zur Datenexfiltration von einer nativen Anwendung zum Browser. Dazu haben wir den ersten Prime+Probe-basierten Cache-Covert-Kanal entwickelt, der Daten von einer nativen CUDA-Anwendung zu einer WebGPU-Anwendung im Browser überträgt. Unser Cache-Covert-Kanal erreicht eine Übertragungsgeschwindigkeit von bis zu 10.9 kB/s. Die selbstkonfigurierende Natur unserer Algorithmen und der kurze erforderliche Zeitrahmen von weniger als 15 Minuten ermöglichen Drive-by-Angriffe während des Surfens im Internet. Unser Angriff erfordert keine Benutzerinteraktion und funktioniert auf einer Vielzahl von Nvidia GPUs.

Der Inhalt dieser Thesis war ein wichtiger Beitrag zu einem Konferenzpapier bei AsiaCCS'24 von Giner et al. [1]. Das Papier demonstriert weitere praktische Angriffe unter Verwendung unserer generischen Primitive. Die in dem Papier beschriebenen Angriffe beinhalten einen Inter-Keystroke-Timing-Angriff, der Zeitvariationen zwischen Tastenanschlägen ausnutzt, sowie einen Ende-zu-Ende-Angriff, der den gesamten in CUDA nativ ausgeführten AES-Verschlüsselungsprozess kompromittiert.

Unsere Arbeit zeigt, dass der Zugang zu GPUs aus dem Browser ein mächtiges Werkzeug für Angreifer sein kann. Wir schlagen vor, dass Browser den Zugang zur GPU wie den Zugang zu anderen sicherheits- und datenschutzrelevanten Ressourcen wie dem Mikrofon oder der Kamera behandeln.

Schlagwörter: Sicherheit · GPU · WebGPU · Seitenkanal

Contents

Chapter 1

Introduction

Traditionally, GPUs are highly specialized devices designed for graphics rendering. Compared to CPUs, GPUs offer high levels of parallelism and throughput at the cost of reduced flexibility and higher latency. The inherent massive parallelism of GPUs also benefits various other applications, as the work from Thompson et al. [2] from 2002 suggests. During the early stages of general-purpose computing on GPUs, tedious work was required to translate problems into the graphics domain to be solved on GPUs [3].

To relieve developers of this burden, Nvidia released CUDA [4] in 2007. CUDA and OpenCL [5], an open standard introduced by the Khronos Group in 2009, are general-purpose computing frameworks for GPUs. With CUDA and OpenCL, developers can create their own compute shaders for execution on compatible GPUs, freeing them from the constraints of the fixed graphics pipeline. General-purpose computing on GPUs has become increasingly important in various fields due to the immense parallel processing power offered by modern GPUs. While the individual speed of GPUs is still slow compared to CPUs, parallelizable workloads profit from thousands of cores. GPUs play an important role in accelerating deep learning algorithms in artificial intelligence [6] to speeding up scientific simulations. GPUs enhance virtualization and cloud computing services by offloading graphics rendering and computational tasks [7]. Even in cryptography, GPUs contribute to accelerating encryption and decryption processes [8, 9]. Overall, general-purpose computing on GPUs has become a cornerstone technology, pushing the boundaries of computation across various domains. Additionally, GPUs have become ubiquitous across devices, from desktop computers to laptops, smartphones, and servers.

Subsequently, GPUs became an interesting target for attackers. Jiang et al. [10] were the first to demonstrate that GPUs are vulnerable to the same cache attacks as CPUs. Their attack code runs on the CPU and uses the shared cache to leak information from the GPU. Similarly, other works assume an attacker on the CPU to spy on GPU applications through leakage from shared components [11, 12, 13, 14]. Several works [15, 16, 6, 17] demonstrate side channels where attacker and victim are co-located on the same GPU. All attacks mentioned above rely on native code execution on the victim's machine. However, acquiring native code execution is often hard to achieve for any attacker.

In contrast to native code, code execution in a victim's browser is often easier to achieve, as users routinely run untrusted, third-party code on websites. WebGL is an API for GPU graphics rendering in the browser. Prior work demonstrates several attacks with WebGL [18, 19].

The introduction of WebGPU [20], a JavaScript GPU API and successor of WebGL, brings general-purpose computing on GPUs to browsers. All major browser vendors are actively contributing to the development of WebGPU [21], with the goal of making it the future standard for utilizing GPU capabilities within browsers. From the beginning, the designers of WebGPU incorporated protections against side-channel vulnerabilities, including restricted access to timers and mitigations similar to those against the malicious `SharedArrayBuffer` use [22]. Furthermore, WebGPU does not provide functions to query the underlying hardware's cache size or other architectural details. No works have demonstrated side-channel attacks in WebGPU so far. This leads us to the following research questions: *Can we construct all necessary building blocks for a GPU side-channel attack in WebGPU? Is it possible to construct a generic, self-configuring attack code that works on various GPU devices without user interaction?*

In this thesis, we present the first side-channel attack from the restricted browser environment using the WebGPU API. We construct a timer that reliably distinguishes cache hits from cache misses. For this distinction, we automatically determine a threshold value. Our self-configuring attack code detects the cache size and the number of cache sets of the underlying GPU hardware. We evaluate these primitives across 12 desktop GPUs from 5 different generations and 2 vendors.

We use this information and the building blocks to construct the first parallelized L2 cache eviction-set-finding algorithm. Based on prior work on eviction-set-finding algorithms on CPUs and GPUs, we develop an algorithm that works in the restricted browser environment. In contrast to native APIs, WebGPU does not provide fine-grained control over cache eviction. Furthermore, WebGPU does not support conventional concepts such as pointers and addresses, a limitation addressed by the group-elimination method proposed by Qureshi et al. [23]. Building on this method, we developed a parallel eviction-set-finding algorithm tailored for WebGPU. Our algorithm reliably finds more than 80 % of sets on all but one tested Nvidia GPU. Depending on the GPU, the set-finding takes between 2 to 12 minutes.

We use the eviction sets to construct the first Prime+Probe [24, 25] cache covert channel in WebGPU. We evaluate a data-exfiltration scenario from a native CUDA application to a WebGPU receiver in the browser. We use a variant of CJAG [26] to select the eviction sets used for transmission between sender and receiver. Using the parallelism of GPUs, we are able to use 1024 sets for message transmission. Our covert channel reaches a true channel capacity of up to 10.9 kB/s.

We evaluate our attack on Windows and Linux systems using Chromium versions 112 to 115 and Firefox 114. Our work demonstrates that access to general-purpose computing on GPUs in web browsers opens the door for various attacks. Our attack works in a time frame of less than 15 minutes and without user interaction, so a drive-by attack in a browser tab is feasible.

While GPUs improve the performance of a wide range of applications in the browser, we emphasize that access to GPUs should be treated like other security- and privacy-related resources, such as camera and microphone.

The content of this thesis was a major contribution to a conference paper at AsiaCCS'24 by Giner et al. [1]. The paper explores additional microarchitectural attacks based on the same techniques developed for this work. These attacks include an inter-keystroke timing attack and a full AES key recovery from the browser on a native AES CUDA application.

Outline. We provide background information about GPUs, GPU APIs, and cache attacks in Chapter 2. In Chapter 3, we present the primitives and set-finding algorithm to use Prime+Probe on the GPU in a restricted browser environment. In Chapter 4, we use the eviction sets to construct a cache covert channel between a native CUDA application and a web application. In Chapter 5, we explore the limitations of our attack, provide an overview of additional attacks featured in the related conference paper by Giner et al. [1], and discuss potential mitigations and other related work. We conclude in Chapter 6.

Chapter 2

Background

In this chapter, we provide background information for the following chapters. Furthermore, we introduce several terms used throughout the remainder of this thesis.

First, we discuss the architecture of GPUs and their evolution from their original role in graphics rendering to their use in general-purpose computations. Then, we compare the cache architecture of GPUs and CPUs and discuss cache addressing and cache replacement policies. Furthermore, we cover the most important cache attack techniques. Finally, we discuss native and web GPU APIs.

2.1 GPU

In this section, we explore the evolution of GPUs, tracing their development from specialized hardware for graphics rendering to their broader application in general-purpose computations. Furthermore, we cover the architecture of GPUs, focusing on an Nvidia Ampere GPU. While the terms are specific for Nvidia devices, other GPU vendors like AMD use similar concepts [27]. Since our work focuses on discrete GPUs, we do not discuss integrated GPUs in detail. While the architecture of integrated GPUs is similar to discrete GPUs, they differ significantly in some key aspects. For instance, integrated GPUs use a portion of the CPU's RAM instead of having dedicated memory like discrete GPUs.

2.1.1 General-Purpose Computations on GPUs

Initially, GPUs were designed for graphics rendering only [28, 29]. GPUs run shaders, specialized programs for different stages in graphics rendering, in a shader pipeline. All components, including units for vertex and pixel shaders, were highly specialized and limited to their specific purposes only.

The idea to use GPUs for more general computations emerged quite early [30, 31, 32, 33]. The inherent attributes of GPUs, namely massive parallelism and high throughput, find applications beyond graphics rendering in numerous other fields. Scientific computing and simulations are just two examples of fields that benefit from these unique characteristics.

Thompson et al. [2] found a way to use the graphics pipeline for matrix multiplication and 3-SAT in 2002. They demonstrated performance improvements compared to CPU implementations, emphasizing the need for general-purpose computations on GPUs.

Figure 2.1: Nvidia Ampere GA102 Architecture [37] with 7 graphics processing clusters. Each cluster is connected to the L2 cache and contains several streaming multiprocessors.

However, the GPU's fixed graphic pipeline necessitated mapping general problems to the computer graphics domain before solving them on GPUs [3, 32, 34].

The release of CUDA by Nvidia [4] in 2006 was a huge step forward for general-purpose computing on GPUs. CUDA enables native support for general-purpose computations on Nvidia GPUs. Other manufacturers, like AMD, followed a similar approach. While CUDA is a proprietary framework by Nvidia, the Khronos Group released the open standard OpenCL [5] in 2009. This paradigm shift meant a significant change in the hardware architecture of GPUs. GPUs adopted a unified shader model, allowing the same hardware to be used for different tasks [35, 36]. While early GPUs had dedicated hardware for specific tasks like vertex and pixel shading, most tasks of the render pipeline are executed on the same hardware. Still, exceptions, like raytracing, are executed on dedicated hardware units.

2.1.2 Architecture

This section explores the core concepts of modern GPU architecture, illustrating these ideas with a detailed examination of the Nvidia GA102 Ampere GPU [37]. As an example of current GPU technology, the GA102 incorporates concepts widely used in the industry by Nvidia and other major manufacturers like AMD [27]. We will explore

Figure 2.2: Nvidia Ampere GA102 streaming multiprocessor [37]. Threadblocks with up to 1024 threads are scheduled on a single streaming multiprocessor. Each streaming multiprocessor consists of 4 processing units. Warps with 32 threads are executed in parallel on each processing unit.

the design features that enable these GPUs to handle computationally intensive tasks effectively, including the use of multiple graphics processing clusters and the integration of sophisticated cache management techniques.

The GA102 is the highest-performing GPU in the Ampere architecture's GA10x lineup, powering devices like the GeForce RTX 3090, GeForce RTX 3080, NVIDIA RTX A6000, and the NVIDIA A40 data center GPU. The architecture is similar to prior Nvidia GPUs, with only minor differences and improvements. Figure 2.1 depicts the architecture of a GA102 GPU.

Graphics processing clusters represent the largest and most complex components within the architecture of Nvidia GPUs. A complete GA102 GPU consists of 7 graphics processing clusters. Each cluster is subdivided into several key functional units: up to 6 texture processing clusters, a raster engine, and two raster operator partitions, each containing 8 raster operator units.

Further breaking down the architecture, each texture processing cluster incorporates 2 SMs (streaming multiprocessors) and a PolyMorph engine. The SMs are particularly critical as they closely resemble CPU cores in their primary function of executing instructions. Figure 2.2 illustrates that an SM principally comprises four processing blocks, also called partitions. Each processing block is a separate SIMD (single-instruction multiple-data) execution unit. They share a common unified architecture for shared memory, L1 data cache, and texture caching with 128 kB capacity per SM. This versatile configuration enables the adaptive allocation of memory resources to meet specific workload demands. Processing blocks include a 64 kB register file, an L0 instruction cache, a warp scheduler, a dispatch unit, a tensor core, load and store units, and two datapaths for mathematical operations. One of these datapaths is capable of either 16 FP32 or 16 INT32 operations per clock, while the other only supports 16 FP32 operations per clock.

Other specialized units in an SM include a raytracing core, four texture units, and special function units dedicated to certain mathematical operations.

In CUDA, threads are organized in thread blocks. Threads within a block may use shared memory or synchronization primitives to communicate. A thread block can contain up to 1024 threads, which are executed on the same SM. These threads are further divided into warps containing 32 threads, all executing the same instructions. Each processing unit can execute one warp, meaning they always execute 32 threads in parallel.

To achieve a high throughput, GPUs execute warps in a SIMT (single-instruction multiple-thread) fashion [38]. Threads are executed in lockstep. All threads in a warp share a single program counter, meaning each thread executes the same instruction. In the case of divergent execution paths, e.g., branch instructions, all paths must be executed. The program counter is combined with an *active mask* to indicate which thread is active at each instruction. For example, a thread is only active for one branch of an `if-else` statement and is masked for the other branch. Divergent branches in a warp are serialized, and all statements in one branch are executed before any statements of the other side are executed [39]. Not only does this loss of concurrency incur a performance penalty, it also complicates fine-grained synchronization between threads within a warp.

Starting with the Volta architecture, Nvidia GPUs support independent thread scheduling [39]. They achieve independent scheduling by maintaining the execution state for each thread, which includes program counters and call stacks. Volta optimizes thread management using a schedule optimizer that groups active threads from the same warp into SIMT units, enhancing processing efficiency. Unlike previous architectures, where divergent paths in branch statements were executed serially, Volta interleaves the execution of such paths. This interleaving not only improves execution efficiency but also allows for fine-grained synchronization among threads within a warp.

In contrast to CPUs, GPUs typically have large register files. For example, Ampere GPUs have a 64 kB register file per processing block. The advantage of a large register file is that there is enough storage for all currently schedulable threads on an SM. Thus, there is no need to transfer registers upon a context switch.

<table>
<tr><td>PD3
[48:47]</td><td>PD2
[46:38]</td><td>PD1
[37:29]</td><td>PD0
[28:21]</td><td colspan="2">2MB PAGE
[0:20]</td></tr>
<tr><td colspan="4" rowspan="2"></td><td>PT (64K)
[20:16]</td><td>64k Page
[15:0]</td></tr>
<tr><td>PT (4K)
[20:12]</td><td>4k Page
[11:0]</td></tr>
</table>

Figure 2.3: 49-bit virtual address breakdown introduced in the Pascal architecture [40].

GPUs use a virtual memory system like CPUs. However, the GPU's virtual memory is completely separate from the CPU's virtual memory. Starting from the Pascal architecture, Nvidia GPUs employ a virtual memory system that uses up to 49 bits for virtual addresses [40]. The five-level page table format supports addressing up to 47 bits of system memory.

Figure 2.3 depicts a 49-bit virtual address breakdown of the Pascal architecture. Depending on the page size, the least significant 21 bits of the virtual address have a different meaning. While 21 bits serve as page offset for 2 MB pages, smaller pages require fewer bits. For 4 kB pages, 12 bits serve as page offset and 9 bits as offset into the page table. For 64 kB pages, 16 bits serve as page offset and 5 bits as offset into the page table. The remaining 28 bits of the virtual address are offsets into the PDs (page directories). 8 bits serve as offset into the PD0, 9 bits are used for PD1 and PD2, and 2 bits for PD3.

2.2 Caches

CPUs and GPUs encounter performance bottlenecks due to the high latency of accessing data from main memory. To address this issue, CPU and GPU architectures incorporate hierarchical cache systems to improve the speed of accessing frequently accessed data. Caches are small, fast memories that store copies of frequently accessed data, providing quicker access than main memory.

In this section, we look into the details of the cache architecture of GPUs. While GPU architectures differ in various aspects, most employ a similar cache arrangement. We discuss more general aspects of caches, like replacement policies, cache organization, and addressing. We conclude this section with architectural details of the caches on an Nvidia Ampere GPU [37].

2.2.1 GPU Caches

CPUs typically use a hierarchical system with multiple cache levels. The lower-level caches are close to the processor cores and are the smallest and fastest. Higher-level caches are further away from the processor core and provide slower access times. However, higher-level caches are typically larger than the lower-level caches.

In general, GPU caches are organized like CPU caches. There is a small, fast L0 instruction cache for each processing block. Each SM includes a 128 kB L1 data cache shared by the four processing blocks. Parts of the L1 cache can be used as a shared

memory region. The segmentation of the memory into L1 cache and shared memory can be configured based on the requirements of the current computation or graphics workload. The L2 cache is shared by all SMs. It has a maximum capacity of 6 MB on Ampere GPUs.

With many cores, cache coherency is very expensive to achieve on GPUs. This is why GPU caches are usually incoherent. To achieve coherency, the caches need to be flushed explicitly. However, the GPU's programming model involves less frequent data sharing among different shaders compared to interactions between processes on CPUs. As a result, using an incoherent caching approach does not present a significant practical challenge.

2.2.2 Cache Addressing

Caches can only store a fraction of the GPU's main memory. Data loaded from the main memory is stored in the cache for quicker subsequent access. Typically, it is not just the specific memory location that gets cached; instead, a larger portion known as a cache line, which is usually 128 bytes wide on GPUs [41, 42], is cached. The 128 bytes are memory locations that are spatially close to the requested memory location. This practice improves performance as programs frequently access neighboring memory locations. Furthermore, caching data byte-wise is too expensive. Another reason for the relatively wide cache lines on GPUs is coalescing. Concurrent memory accesses to contiguous memory locations by threads within a warp are coalesced into a single warp-wide access [43, 44]. Given that a warp consists of 32 threads, a 128-byte wide cache line efficiently stores 4 bytes per thread in one memory access.

The address is divided into three parts to determine the data's location within the cache. The least significant bits serve as an offset within the cache line. Given that cache lines on GPUs are usually 128 bytes wide, the corresponding offset is represented by 7 bits.

The second part is the cache index, which points to a cache set. Set-associative caches organize cache lines into sets, each comprising several lines linked by a common cache index. Each cache set contains several lines that can independently store data. In the case of a direct-mapped cache, the cache index points to a single cache line.

The remaining bits of the address serve as the tag. Since caches are smaller than the main memory, each cache set must handle more main memory addresses than it has the capacity for. Consequently, each cache line not only contains cached data and a valid bit but also includes a tag. This tag uniquely identifies the segment of main memory being cached in that specific cache line.

2.2.3 Replacement Policies

Newer entries replace older cached data. In systems where data can be stored in multiple cache locations, deciding which data to replace is crucial. Various strategies have been developed to manage this replacement process effectively, each designed to optimize cache

memory usage in different scenarios. These strategies determine which memory locations are to be replaced next.

With the optimal policy [45], the entry whose next memory reference is the farthest in the future would be replaced [46]. However, this policy requires knowledge of every future memory reference, making implementing the optimal replacement policy infeasible.

Several policies use program counter information [47, 48, 49] to optimize cache usage, yet processor manufacturers have been reluctant to implement these due to the significant hardware changes required [50]. Furthermore, there are combinations of different replacement policies. Adaptive cache replacement policies change their behavior and are used in various processors [51, 52]. In this section, we focus on the most prevalent and simple cache replacement policies rather than discussing sophisticated methods.

LRU (Least Recently Used). The LRU policy tracks the timestamps of each cache entry and replaces the entry that has been the longest in the cache. As LRU needs to keep track of timestamps, it is expensive to implement in hardware. LRU generally performs well compared to other replacement policies for many workloads [46]. LRU can lead to thrashing for memory-intensive workloads, where cache lines go from LRU to MRU without cache hits.

An approximation to LRU is NRU (Not Recently Used) [53]. In contrast to LRU, NRU does not keep timestamp information but information about which sets were used recently. According to Briongos et al. [54], Intel uses NRU for some of their processors.

Pseudo-LRU. Techniques that estimate LRU are referred to as pseudo-LRU. Research by Al-Zoubi et al. [46] indicates that pseudo-LRU can approximate and surpass LRU in performance but with considerably less complexity across various cache configurations. Unlike standard LRU, pseudo-LRU does not require keeping timestamp information, making it cheaper to implement in hardware [55]. This cost efficiency, combined with its minimal performance overhead, makes pseudo-LRU a preferred choice among processor vendors. For instance, Intel employs undocumented variants of pseudo-LRU in several of its processors [54, 52]. Additionally, prior work suggests that GPUs often utilize LRU or pseudo-LRU for their cache replacement strategies [56].

LFU (Least Frequently Used). LFU replaces the cache lines that were referenced the fewest number of times. The underlying assumption is that lines referenced more often are more likely to be referenced in the future [57]. However, LFU is susceptible to keeping obsolete lines that were accessed frequently.

Random. A random replacement policy can be used to reduce the implementation cost. Random replacement ignores temporal and spatial locality. While performance is worse than LRU for most workloads, there are cases when random performs similar or even better than FIFO [46].

FIFO (First-In First-Out). LRU can sometimes be too expensive to compute, so some architectures use FIFO [55]. FIFO, or round robin, replaces the oldest entry in the cache, regardless of when it was last accessed. The performance is not as good as LRU for most workloads and can even be worse than random in some cases [46].

2.2.4 Cache Organization

Several methods exist for mapping addresses from RAM to cache locations, each with advantages and disadvantages, as no universally optimal solution fits all cache sizes. The cache organization defines where a specific memory block can be placed within a cache.

Direct-Mapped Cache

In a direct-mapped cache, each location in RAM maps to exactly one cache line, simplifying the retrieval process as the specific cache location is immediately identifiable. However, this simplicity often results in poor cache utilization. Particularly, when multiple addresses that map to the same cache line are accessed alternatingly, they evict each other with each access, a phenomenon known as cache thrashing. This constant eviction cycle degrades performance severely.

Fully-Associative Cache

A fully-associative cache is the opposite of a direct mapped cache. Each memory address can be stored in any cache line, optimizing cache utilization to its theoretical maximum. The necessity to search the entire cache for a hit with each memory access renders this type of cache impractical for larger sizes due to its computational expense. Furthermore, when an eviction is necessary, any cached address can be a candidate for removal, depending on the used replacement policy, such as LRU or FIFO.

Set-Associative Cache

A set-associative cache combines the advantages of both direct-mapped and fully-associative caches. Unlike a fully-associative cache, where any address can map to any cache line, in a set-associative cache, addresses map to specific sets, and each set contains a predefined number of cache lines, known as *ways*. We refer to addresses mapping to the same cache set as *congruent*.

For instance, in a 16-way set-associative cache, each memory location in RAM can be stored in one of 16 different cache lines within a specific set. This configuration avoids thrashing by allowing up to 16 congruent addresses to be cached simultaneously without conflict. Moreover, compared to a fully-associative cache, which requires searching every cache line for a hit, a set-associative cache only searches within the specified set, drastically reducing the number of comparisons needed to find a hit or determine a line for eviction. This not only boosts performance but also scales efficiently with larger caches.

2.2.5 Indexing and Tagging

Caches can be indexed and tagged using virtual or physical addresses. Depending on the chosen method, several non-desirable scenarios can arise. Among these issues are homonyms, which describe situations where the same virtual address maps to multiple physical addresses, thus failing to uniquely identify cached data [58]. Solutions to homonyms include using physical address tags, flushing the cache during context switches, tagging the virtual address with an address space identifier, or using non-overlapping memory layouts for different virtual address spaces.

Also among the issues in caching are synonyms, where different virtual addresses map to the same physical address, resulting in multiple cache entries for the same data [59]. This setup complicates cache coherence as updates to data in one cache line are not automatically reflected in others, leading to inconsistencies. Flushing the cache during context switches does not resolve synonyms within a single address space.

Several strategies are employed to manage this problem effectively. Restricting write access to affected addresses can ensure data consistency, although this approach may not be feasible for all applications due to its restrictive nature. Furthermore, broadcast mechanisms and snoop protocols ensure cache coherency by synchronizing updates to cache lines, effectively mitigating synonyms. Bloom filters can help to decrease further the synonym lookup energy [60, 61].

Alternatively, a sophisticated combination of virtual index and physical tag can be used, which optimizes cache indexing and mitigates synonyms. The following combinations of virtual or physical indexing and tagging exist.

VIVT (virtually-indexed virtually-tagged). Using the virtual address for indexing and tagging is simple and fast since no address translation is required. The disadvantage of VIVT caches is that they suffer from homonyms and synonyms, complicating cache coherence significantly. For the lower-level caches of GPUs, this is a less significant concern compared to CPUs. The high level of parallelism in GPUs makes it impractical and expensive to ensure cache coherency in L0 and L1 caches. By flushing the cache during context switches, the problem of homonyms is also resolved. Since context switches on GPUs do not occur as frequently as on CPUs, this does not introduce a significant performance overhead.

VIPT (virtually-indexed physically-tagged). VIPT is used for various cache levels on many CPUs. The virtual index for the cache line can be used immediately to load the corresponding cache line. While loading the cache line, the address translation happens in parallel. Since the tag is only needed after loading the cache line, the performance overhead compared to VIVT is small, especially when the address translation is in the TLB (Translation-Lookaside Buffer).

In a virtual memory system, addresses are translated at the granularity of a page. The least significant bits of the virtual address serves as the byte offset within the physical page. Subsequently, the offset bits are identical in physical and virtual addresses. VIPT caches take advantage of this consistency; for instance, with commonly used 4 kB pages,

12 bits of the virtual address are used for the byte offset within these pages. Considering a cache line width of 128 B, 7 bits are used as the offset within the cache line. This configuration leaves 5 bits for indexing the cache set, allowing for a maximum of 32 (2^5) different sets. Given these parameters and a set associativity of 16, the maximum cache size in this setup is 64 kB (2^{12} bytes per cache line multiplied by 16 lines), mimicking a PIPT (physically-indexed, physically-tagged) cache in operation. As a result, this VIPT cache configuration avoids the complications associated with homonyms.

PIPT (physically-indexed physically-tagged). PIPT caches are relatively slow because the virtual address must be translated to its physical form before the cache index can be looked up. This makes PIPT caches bad candidates for the fast and small lower-level caches. Like VIPT caches, PIPT caches use a physical tag to avoid homonyms. Using the physical address as a cache index solves the synonym problem. Moreover, PIPT caches are not limited by the same size restrictions as VIPT caches, making them a good option for larger, higher-level caches.

PIVT (physically-indexed virtually-tagged). Although there is a documented real-world example of a PIVT cache [62], this architecture exhibits several drawbacks compared to the aforementioned indexing and tagging techniques. Primarily, the necessity to look up the physical address for indexing slows down the cache operation significantly. Using a virtual tag also leads to the same complications encountered with VIVT caches, namely homonyms and synonyms.

2.2.6 Ampere Cache

On Ampere GPUs, the L0 instruction and L1 data caches are VIVT, while the L2 cache is PIPT. Since cache coherence is complex to achieve due to the computing model of GPUs, the caches are non-inclusive. The cache lines are 128 B wide. Abdelhkalik et al. [63] report a latency of 33 cycles for an L1 cache hit and 200 cycles for an L2 hit. They note a latency of 290 cycles for global memory loads but do not specify if this is for a TLB hit or miss. In contrast, prior work by Jia et al. [64] on the Nvidia Turing architecture, the generation before Ampere, identified a 296-cycle latency for a TLB hit and 616 cycles for a TLB miss. Given the similarity of other values between the two studies, it is likely that Abdelhkalik's 290-cycle measurement corresponds to a TLB hit.

Access latencies on GPUs are relatively high compared to those on CPUs [65]. GPUs are optimized for high throughput, so increased latencies do not impact their performance as significantly as they do on CPUs [66, 67]. High latency values on GPUs often result from throughput optimizations. For example, cache lines on GPUs are generally larger than cache lines on CPUs. Large cache lines enable the GPU to coalesce parallel memory accesses of 32 threads in a warp and cache them in a single cache line [44]. This coalescing depends on the spatial locality of the memory accesses and improves throughput at the cost of increased latency.

2.3 Cache Attacks

Over the past decade, numerous studies have documented various microarchitectural attacks [68, 69]. These attacks target shared microarchitectural elements, including branch predictors [70, 71] and TLBs [72, 73]. Cache-based attacks enable a variety of malicious activities, ranging from leaking sensitive information through side channels [74, 75] to creating covert channels that bypass traditional security measures [25, 26, 76, 77, 78, 79, 80, 81, 82, 83], and even breaking cryptographic algorithms [24, 84, 85]. Notably, these vulnerabilities are not confined to native applications; they can also be executed within constrained environments like web browsers, demonstrating their broad potential for exploitation [86, 87, 81, 51].

The key component of timing-based cache side-channel attacks is the timing difference between a cache hit and a cache miss. The access latency to a memory location is compared to established baseline values. If the measured latency exceeds a certain threshold, the memory access was a cache miss. If the latency is below the threshold, the data was cached with a high probability. To measure the access latency, an attacker needs a timer that is accurate enough to distinguish a cache hit from a cache miss. In response to timing attacks, browser vendors removed access to fine-grained timers [88, 89, 90]. However, multiple studies present a wide range of sources of timing information accurate enough to mount attacks on systems with limited timer API access [91, 92, 87, 18, 93]. For example, Schwarz et al. [92] describe various new mechanisms for obtaining timestamps, some of which offer a resolution that is increased by 3 to 4 orders of magnitude compared to the timestamp provided by `timestamp.now`. For example, they construct a timer with a resolution close to the native timestamp counter by using the `SharedArrayBuffer` Javascript interface that is shared between web workers. Browser vendors responded again by restricting access to the timing sources [94, 95, 96, 97]. Advanced defenses try to mitigate timing side channels in a more principled manner. Kohlbrenner et al. [91] propose to add randomness in the JavaScript event loop to thwart exact timing measurements. On the contrary, Cao et al. [98] present their approach called *Deterministic Browser*, where an attacker will always obtain fixed timing information to prevent timing attacks.

There are various types of cache attacks with different requirements and efficacies. The efficacy of these methods can vary significantly, depending on factors such as the cache architecture and the isolation mechanisms in place. An example of a simple cache attack is the cache occupancy attack, where the attacker does not need knowledge about the cache architecture or addresses. The attacker simply fills the cache and lets the victim program execute. After that, the attacker checks which sections of the previously filled cache are still cached and which have been evicted. Shusterman et al. [99] demonstrated how a simple cache occupancy attack can be used for sophisticated purposes, like fingerprinting attacks.

In this thesis, we focus on more fine-grained cache attacks. Fine-grained attacks require a deeper understanding of the cache architecture and rely on a more sophisticated attacker model, making them more potent. While attacks like Flush+Reload [100] require shared

memory between an attacker and victim process, others like Prime+Probe [25, 24] have lower requirements. We discuss these attacks in the following subsections.

2.3.1 Prime+Probe

The Prime+Probe attack, first described by Percival [25] and Osvik et al. [24], allows an attacker to determine whether a victim has accessed a specific cache set. Like other cache attacks, the victim can be a cooperating program and act as a sender in a cache covert channel.

Prime+Probe exploits shared cache sets between attacker and victim and is coarser grained than Flush+Reload, which targets individual cache lines to deliver more detailed insights into memory access patterns. Prime+Probe offers a significant advantage despite its broader scope: it does not require shared memory between the attacker and the victim, circumventing a key limitation of Flush+Reload. Additionally, unlike Flush+Reload, Prime+Probe does not require the ability to manipulate the cache directly using a flush instruction.

Its weak assumptions make Prime+Probe well-suited for conducting attacks in constrained environments, such as from within web browsers. Prior work shows how Prime+Probe can be used for cross-VM attacks [101, 80], covert channels [102], and attacks from within sandboxed JavaScript [81, 51]. The Prime+Probe attack technique extends beyond cache attacks, targeting various types of buffers, including DRAM row buffers to monitor keystrokes [103] and combining with Rowhammer for cryptographic attacks [104]. It has also been used on branch predictors to compromise RSA [70] and establish covert channels [71], as well as to attack KASLR [105].

Eviction Sets

Before we can mount a Prime+Probe attack, we need to find *eviction sets*. An eviction set is a minimal set of congruent addresses that evicts a cache set. Applications of eviction sets go beyond the scope of Prime+Probe attacks. Gruss et al. [51] use eviction sets to enforce high frequency DRAM accesses and subsequently mount a fault attack. Kocher et al. [86] use eviction sets to delay the decision on which direction a branch instruction will take, thereby increasing the number of speculatively executed instructions.

The ease or difficulty of finding eviction sets depends on the system architecture and the capabilities of the attacking program. While an attacker only requires a timer that is accurate enough to distinguish cache hits and cache misses to find eviction sets, knowledge about certain aspects of the underlying cache architecture or access to addresses can help to improve the eviction set search significantly. In general, there are two approaches to finding eviction sets. Static approaches to finding eviction sets use reverse-engineered mapping functions and virtual or physical addresses. In contrast, dynamic approaches rely on timing measurements to identify colliding addresses, eliminating the need for knowledge about mapping functions or often even access to virtual or physical addresses. Often, hybrid methods are implemented in practice, such as the one detailed by Maurice et al. [26], which combines elements of both static and dynamic approaches.

Finding eviction sets in a virtually indexed cache is trivial, as the attacker already has control over the set index. Finding eviction sets in physically mapped caches is more complicated, even with knowledge about the physical address. Modern processors use per-core cache *slices* for their LLCs [80]. Every slice is a separate cache, but every core has access to the entire LLC. Intel uses an undocumented hash function to map addresses to cache slices. Prior work reverse-engineered the slice mapping functions for various processors [73, 106, 107, 108]. Liu et al. [80] present an eviction-set-finding algorithm that finds eviction sets for all cache slices independent of the cache slice hash function.

In response to several attacks, access to the virtual-to-physical address mapping in the Linux kernel is only available with *root* privileges [109]. Various attacks on CPU caches use huge pages with 2 MB [80, 101] to circumvent this restriction. As the last 21 bits of the virtual and physical addresses are the same for 2 MB pages, the LLC becomes effectively virtually indexed.

The situation is even more complicated if the attacker runs in a restricted browser environment without access to pointers and virtual addresses. Oren et al. [81] first demonstrated that microarchitectural side-channel attacks in the browser are feasible. They encounter similar limitations to those we experience in our eviction set search, such as the absence of pointers and addresses in the browser and the unavailability of huge pages. In contrast to Liu et al [80], they do not assume 2 MB huge pages, but more common 4 kB pages. Their algorithm accesses a buffer larger than the cache size in strides of 64 B. Each memory access refers to a different cache line as they target 64 byte wide cache lines. However, as there are no pointers in JavaScript, they use offsets in the buffer for the memory accesses.

They start with a large buffer and a target address to find an eviction set for the target address. Like Liu et al. [80], they iteratively remove addresses from the buffer and verify that the target address is still evicted. They stop when only as many addresses remain as there are cache ways. As virtual memory is aligned to 4 kB pages, they can exploit that the lower 12 bits of virtual and physical addresses are equal and that bits 12 to 6 are used as part of the cache index to reduce the search space for each address [73].

In contrast to Liu et al. [80], they cannot identify the CPU's cache sets corresponding to the eviction sets they found. Furthermore, they find different mappings each time the algorithm runs. As they cannot identify the CPU's cache sets corresponding to the found eviction sets, their algorithm is prone to generating duplicate sets. These duplicates can result in self-evictions, creating misleading outcomes during the probing phase. Specifically, the probing might erroneously indicate a victim's activity when, in fact, the duplicate set has evicted the probed set.

Prior work reverse-engineered mapping functions on various GPUs [110, 111, 112, 64]. However, our tests suggest GPUs use different mapping functions across different devices. As many GPUs have non-power-of-two cache and VRAM sizes, they employ non-linear mapping functions or mapping functions that use a different address range. Furthermore, GPUs from different manufacturers and mobile GPUs use different mapping functions. Qureshi et al. [23] build upon methods from prior work [81, 80] and present the group-elimination method. Instead of removing one address at a time, they remove

multiple addresses to improve the algorithm's performance. Vila et al. [113] follow a similar approach to the group-elimination method. In Chapter 3, we discuss how we extend prior work to identify eviction sets by leveraging the WebGPU API.

Prime

The first step of Prime+Probe is the *prime* phase, where the attacker causes contention in a specific cache set. For this, the attacker fills a specific cache set with an eviction set, effectively evicting all previously cached data in that specific set. Following the prime phase, the attacker either triggers or waits for the execution of the victim program. During this waiting period, the victim may use the targetted cache set and evict elements from the attacker's cache set. The attacker can use multiple cache sets to observe the victim's activity. However, the probing frequency decreases with a higher number of sets.

Probe

In the *probe* phase, the attacker loads all addresses of the eviction set and measures the access time for each address. If the victim program accesses data cached in the targetted set, at least one of the attacker's cache lines is evicted. The attacker can detect this because the evicted line results in a noticeably slower access from the main memory. Consequently, the attacker learns that the victim program accessed a memory location that maps to the same cache set as the eviction set. The attacker accesses all addresses of the eviction set in the *probe* phase, thus implicitly *priming* the set for the next round of Prime+Probe.

Replacement Policies

Cache replacement policies play an important role in a Prime+Probe attack. Prime+Probe attacks work best for architectures with LRU, pseudo-LRU, or FIFO replacement. With these replacement strategies, every address of the eviction set is accessed only once to evict the entire cache set. Gruss et al. [51] developed a method to efficiently evict cache sets under complex cache replacement policies. They create eviction strategies for various systems by combining static and dynamic techniques. These eviction strategies evict cache sets quickly enough to trigger the Rowhammer bug.

Variations

One challenge of Prime+Probe is cache trashing, where probing an address evicts a previously primed address. Prior work discusses several solutions to cache trashing. Tromer et al. [84] use a doubly-linked list and traverse it forward for priming and backward for probing to minimize cache trashing. Their solution works best for LRU caches. Lipp et al. [79] reduce the chance of set trashing in a cache with a random replacement strategy by using smaller sets for priming and probing. While many Prime+Probe attacks assume inclusive caches, Yan et al. [114] target cache directories with Prime+Probe to attack non-inclusive caches.

Prime+Count. Cho et al. [115] discuss Prime+Count, a slight alteration of Prime+Probe. Instead of probing cache lines, they use performance counters to count how many cache lines have been evicted. They show how their coarser-grained technique can reduce noise introduced by pseudo-random replacement policies.

Evict+Time. Osvik et al. [24] also proposed Evict+Time, a more coarse-grained variant of Prime+Probe. First, they measure the duration of a victim program as a baseline value. Then, they evict a cache set and measure the execution time of the victim's program again. If the victim uses the evicted cache set, the execution time is higher than the baseline value.

2.3.2 Flush+Reload

Gullasch et al. [116] exploit that shared memory between two different processes is cached in the same cache line in the L1 cache. Yarom and Falkner [100] built on the attack from Gullasch and proposed the Flush+Reload attack, targeting the L3 cache. Both attacks allow the attacker to monitor the memory activity of another program at cache line granularity.

For Flush+Reload, an attacker constantly flushes a cache line in shared memory and waits for the victim to be scheduled. After that, the attacker accesses the location again and measures the access latency. A low access latency indicates a memory access of the victim program, as the recently flushed memory location is cached after the victim has been scheduled.

A drawback of Flush+Reload is that it requires shared memory for the location to be monitored between the attacker and victim program, which is harder to obtain than co-location. Furthermore, the memory shared between the attacker and the victim must be cached in the same cache location. Another limitation of Flush+Reload is that it requires a dedicated cache line invalidation instruction, like `clflush`. Such instructions are unavailable on various processors and in certain sandboxed environments like the browser or are only available to privileged users.

Flush+Reload was initially proposed as an attack targeting shared libraries, which are typically accessible to the attacker. However, this approach requires knowledge of the specific libraries the victim uses. Beyond shared libraries, attackers can also target files used by the victim and accessible to the attacker. In addition to monitoring access to shared libraries or files the victim uses, an attacker can exploit the operating system's active page deduplication, where identical pages are merged into a single copy-on-write page. This approach, which requires the attacker to create pages identical to those used by the victim, is more sophisticated than simply mapping shared libraries or files.

Flush+Reload has become a fundamental component in various microarchitectural attacks. Prior work demonstrates how Flush+Reload can be used to attack cryptographic implementations [117, 100, 118, 119, 120], monitor user interaction [74, 79, 121, 122], or as a covert channel [86, 79, 123, 124].

Variations

There are several approaches to overcome the limitations of Flush+Reload:

Evict+Reload. As the Flush+Reload attack requires a dedicated flush instruction, it is not applicable in environments where such an instruction either does not exist or is only available to privileged users. Gruss et al. [74] introduced Evict+Reload. Evict+Reload fills the cache set containing the target cache line similar to *priming* in Prime+Probe. The result is similar to flushing, but the entire cache set is evicted instead of only flushing a single cache line. The second step, reloading the target address, remains the same as in the traditional Flush+Reload attack. Evict+Reload enables attacks in JavaScript [87, 124], remote attacks [125], and attacks on ARM devices, where flush instructions might not exist [79]. Gruss et al. [51] and Aweke et al. [126] demonstrate flush-free Rowhammer attacks by evicting cache sets in their concurrent works.

Flush+Flush. In a Flush+Reload attack, we are not interested in the loaded value of the *reload* step. Instead, we only measure the access time to infer the victim's behavior. Furthermore, the cache line is flushed again after reloading to prepare for the next measurement. Gruss et al. [127] build on this observation and present Flush+Flush. Flush+Flush requires no memory access. Instead, they exploit the timing differences in the `clflush` instruction. As the LLC on CPUs is commonly inclusive, flushed memory is also removed from the lower cache levels. Flushing data from several cache levels takes longer than flushing a cache line where no data is currently cached. Flush+Flush does not make a potentially slow access from memory for each measurement, so it achieves a higher time resolution than Flush+Reload. Gruss et al. [127] show that Flush+Flush can be used to attack cryptographic algorithms, monitor user input, and construct channels for covert communication.

Cache Template Attacks. One challenge in mounting a Flush+Reload attack is finding interesting shared memory regions between the attacker and the victim. Shared memory regions can be shared libraries and files used by the victim that are accessible to the attacker or the victim's executable binary. Gruss et al. [74] introduced *cache template attacks*. Cache template attacks enable automatic attacks on cryptographic implementations and keystroke logging, among other applications. A cache template attack consists of a profiling phase, where memory regions that leak information about a victim are identified automatically. In the exploit phase, the attacker monitors these memory regions to infer the victim's behavior. An advantage of cache template attacks is that attackers need no knowledge about specific software versions or other specific system information.

Since sharing memory between applications on GPUs is uncommon and not possible on many devices, to our knowledge, there have been no known demonstrations of Flush+Reload-style attacks on GPUs so far.

2.4 GPU APIs

GPU APIs are interfaces that enable software developers to harness the computational power of GPUs for rendering and general-purpose computations. These APIs act as a bridge between the application or software and the GPU hardware, abstracting the complexity of direct hardware manipulation while offering various degrees of control over the GPU's capabilities. While some of the GPU APIs are portable and compatible with various GPU manufacturers [128, 129, 130], others like CUDA [4] are only available for Nvidia GPUs. We classify GPU APIs into two main categories: Native APIs provide developers with low-level access to the GPU, and high-level APIs, like those accessible from web browsers, provide a higher abstraction level to developers. In this section, we first look into native APIs and discuss CUDA in more detail. After that, we cover Web APIs and discuss the characteristics of WebGL [131] and its successor, WebGPU [20].

2.4.1 Native APIs

Native GPU APIs are low-level APIs that provide detailed control over the GPU. They offer a low abstraction level and fine-tuned control over the hardware. This level of control enables precise optimization, leading to superior performance compared to higher-level APIs. Native GPUs provide means to use the GPU for graphics rendering or general-purpose computations. In 1992, the Khronos Group introduced OpenGL [132], a cross-language, cross-platform API for graphics rendering on Linux and Apple platforms. Shaders for OpenGL are written in GLSL (OpenGL Shading Language). Windows uses Direct3D [130], part of Microsoft's DirectX API collection, for graphics rendering.

In 2007, Nvidia released CUDA [4], allowing for general-purpose computations on Nvidia GPUs. With the high market share for discrete GPUs of Nvidia and CUDA's ease of use, CUDA is currently the most widely used framework for general-purpose computations on GPUs [133, 134, 135]. Recognizing the need for a universal solution, the Khronos Group introduced OpenCL [5] for general-purpose computations across GPUs from various manufacturers. Apple chose a different approach and transitioned from OpenGL to Metal [129], a modern alternative that merges graphics rendering and general-purpose computations into a single API.

In 2016, the Khronos Group released Vulkan [128], the successor and modern alternative to OpenGL. Vulkan provides developers with a lower-level API and supports graphics rendering and general-purpose computations. The Vulkan API is supported on desktop and mobile devices. In contrast to OpenGL, Vulkan shaders are compiled to the intermediate binary format SPIR-V. This pre-compilation step speeds up application initialization. Unlike OpenGL, where each OpenGL driver implements its compiler, Vulkan drivers can focus on hardware-specific optimizations and code generation.

While these APIs have different characteristics, their operations typically include texture mappings, rasterization, and memory management.

CUDA

In 2007, Nvidia introduced CUDA, a proprietary API for general-purpose computing on Nvidia GPUs. Compared to alternative APIs, CUDA offers more precise control over Nvidia GPUs. CUDA is available on all Nvidia GPUs, starting with the Tesla architecture. GPUs are categorized by their *compute capabilities*, each represented by a version number that specifies the features and technical specifications the hardware supports, including specific instructions and the number of registers per multiprocessor [136]. CUDA applications can query these compute capabilities at runtime.

A CUDA program is essentially a C/C++ program, making it easily accessible even for developers with minimal prior knowledge of graphics programming. In a CUDA program, functions are marked with annotations specifying whether they can run on the host or the graphics hardware. A subroutine on the GPU is called kernel, and one CUDA kernel may run at a time. To launch a subroutine, the host specifies the number of blocks and threads responsible for executing the routine. Passing arguments by value is possible in CUDA, but since the GPU cannot directly access the host's memory, data intended for computation on the GPU must be transferred beforehand through buffer copies. Launching a CUDA kernel is a non-blocking operation. CUDA provides synchronization mechanisms that facilitate waiting for scheduled programs. All subsequent API calls are ignored upon an error, and information about the error is returned.

An advantage of CUDA over other GPU APIs lies in the power of Nvidia's PTX ISA. The PTX assembly language is represented as ASCII text, enabling inline assembly in CUDA. With inline assembly, developers can achieve fine-grained hardware control, including cache control for prefetching and even altering the replacement strategy. Extensive optimizations are applied when compiling CUDA code to PTX, resulting in high-performance levels. The PTX assembly code is compiled to SASS, a low-level assembly language specific to Nvidia GPUs. SASS is further optimized when translated into machine code, which is executed directly by the GPU hardware.

CUDA does not provide any locking mechanisms. Generally, GPU workloads should operate in parallel on independent data. Even though there are atomic operations for float and integer values, building custom locking mechanisms is highly discouraged.

2.4.2 Web APIs

Web APIs provide a more accessible, higher-level abstraction of GPU hardware compared to native GPU APIs, significantly simplifying the interface for developers. These APIs, accessible through JavaScript running directly in the browser, require minimal setup and integrate seamlessly, facilitating rapid and straightforward development. However, this ease of use often comes at the expense of performance. In this section, we discuss WebGL, the predominant browser GPU API, and its successor, WebGPU, which enables general-purpose computations on GPUs directly within the browser.

WebGL

WebGL [131] is a JavaScript graphics rendering API. WebGL is designed for rendering interactive 2D and 3D graphics in compatible web browsers and does not provide direct support for general-purpose computations on GPUs. However, libraries like GPU.js [137] enable general-purpose computing in the browser by transpiling JavaScript code to WebGL shader language. WebGL builds upon OpenGL ES (OpenGL for Embedded Systems) [138], a subset of the OpenGL API. WebGL shader code is written in GLSL ES, which is similar to C and C++. The code is passed to the WebGL API as text strings and compiled into GPU code.

Like OpenGL, WebGL operates as a state machine where commands modify the current state and execute subsequent commands within that context. Modifications to the state by one part of an application can inadvertently affect other parts that rely on a different state. This creates hidden dependencies within the code, potentially leading to subtle bugs that are hard to diagnose and fix.

Although technically feasible [139], conducting general-purpose computations on WebGL involves cumbersome workarounds. These workarounds lack the efficiency and flexibility of modern general-purpose GPU APIs like CUDA or WebGPU. The *WebGL 2.0 Compute* initiative aimed to introduce support for compute shaders on the web through the WebGL rendering context [131]. However, due to the inherent limitations of WebGL and the emergence of new native GPU APIs, the group decided to discontinue efforts to further develop the WebGL API. Instead, they announced that *WebGPU will be the path forward for compute shaders on the web* [131].

WebGPU

WebGPU [20] represents a significant evolution in web-based graphics and computation, aiming to provide a more efficient, powerful, and secure means of accessing GPU resources directly from web browsers. It is a flexible, modern compute API that enables developers to use the GPU in web browsers not only for graphics rendering but also for general-purpose computations. WebGPU is standardized by the W3C [20], with participation from all major browser vendors. Despite being in active development, the engagement of these vendors indicates a strong potential for widespread deployment in the coming years [21]. As GPUs become increasingly ubiquitous across devices, including mobile phones, the utilization of GPU access from web browsers is predicted to become even more prevalent [140, 141]. The underlying implementation of WebGPU in Chromium, Dawn [142], and Firefox, wgpu [143], are available as open-source and pave the way for the integration of WebGPU into applications beyond the scope of web browsers.

Compared to WebGL, the WebGPU API is more streamlined and does not depend on a single state object. In contrast to WebGL, which is a wrapper around OpenGL ES, WebGPU supports native APIs, like Vulkan, Metal, and DirectX, through a JavaScript API. While implementations provide access to the GPU through these APIs, they may restrict access to resources, like memory and runtime, for security reasons. Instead of mirroring a single native API, WebGPU uses its abstractions. The language for WebGPU

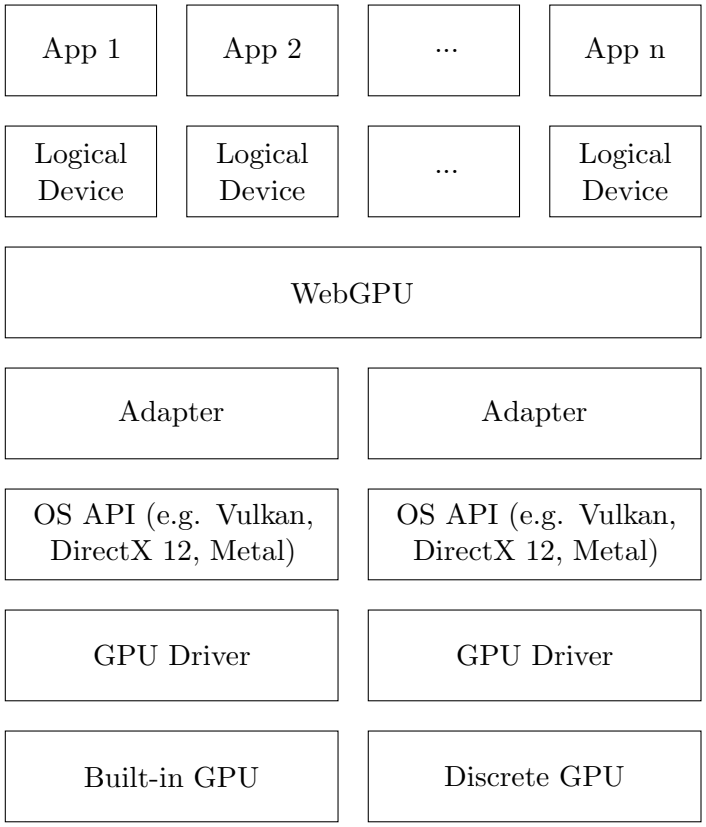


Figure 2.4: Hierarchical abstraction layers spanning from applications utilizing GPUs through the WebGPU interface down to GPU hardware [144].

is WGSL (WebGPU Shading Language). It is compiled into a format the underlying system understands, like SPIR-V for Vulkan or HLSL for DirectX 12.

Abstractions. WebGPU abstracts away the details of the underlying hardware architecture so that the same application runs on different kinds of GPUs. This is achieved by multiple layers of abstractions, depicted in Figure 2.4.

From the bottom-up, the GPU driver provides an interface to the operating system. Drivers and the operating system facilitate multiplexing, abstracting away shared resources and enabling applications to use the GPU as if they were the sole users. The operating system then provides an interface to applications. These interfaces include Vulkan, Microsoft's DirectX 12, and Apple's Metal. Adapters are a crucial part of WebGPU. They translate the underlying interface to a common denominator. Each physical GPU corresponds to one adapter. WebGPU facilitates another level of multiplexing to provide multiple logical GPU devices to web applications. To request an adapter in JavaScript, we use `navigator.gpu.requestAdapter()`. There are optional arguments to request a high-performance or a low-energy device. Some implementations even provide software

fallback adapters. After retrieving an adapter, we can request a logical device using `adapter.requestDevice()`. However, the logical device may not match the underlying physical device's capabilities. Instead, logical devices adhere to a common denominator with specific limits for properties like the maximum amount of simultaneously running threads. If an app runs within these limitations, it works across many different GPUs, thus achieving high portability. It is possible to request higher limits at the cost of reduced compatibility with a narrower range of GPUs.

Computations. Historically, GPUs ran different shaders in a pipeline to render graphics. This is also reflected in the design of WebGPU. First, data is copied to the GPU. Then, different shaders operate on this data in a pipeline, and finally, the data is rendered on screen or copied back to the host memory. In WebGPU, we program shaders as stages for a pipeline and set the stage's entry points.

WebGPU currently supports render pipelines and compute pipelines. Other pipelines, like a raytracing pipeline, may be supported in the future. A render pipeline typically consists of multiple stages, e.g., a vertex and a pixel shader. The output of the render pipeline is typically a 2D image. A compute pipeline usually only consists of one stage. It operates on data and returns a buffer containing the result.

WebGPU offers developers the concept of workgroups to use the parallelism of GPUs. The CUDA equivalent to a workgroup is a thread block. A workgroup represents a collection of threads that execute a shader in parallel. Within a workgroup, threads are executed on a single core and may use a small shared memory region. WebGPU provides a means of synchronization for these threads. Workgroups are organized three-dimensionally to increase locality. Ideally, this improves performance since neighboring workgroups are expected to access similar areas in memory, leading to better utilization of caches.

To run a pipeline, we need to place the pipeline in a command buffer. The command buffer is a queue containing encoded commands that can be executed on the underlying GPU. This enables efficient batching of commands and parallel command generation. The command encoder schedules pipelines and copies data between GPU buffers or between the GPU and the host. To encode a pipeline, we use the `PassEncoder`. In the `PassEncoder`, we set the pipeline and the number of workgroups to be dispatched. The `PassEncoder` contains the state and data required to execute the shader. This contrasts with OpenGL, where state information is stored in a global state object.

WebGPU uses *Bind Group Layouts* to define properties of resources, like textures and buffers, used in a pipeline. They act as an interface between the pipeline and the resources. A *Bind Group Layout* defines the type of buffers used in a pipeline. The type defines the purpose of the buffer and other properties, like whether the buffer is writeable. An actual *Bind Group* needs to adhere to the *Bind Group Layout* and contains the actual GPU resources.

GPU buffers need to achieve high levels of throughput. That is why GPU buffers are generally not exposed to host memory. Staging buffers are used to copy data to and from the GPU. Staging buffers are GPU buffers accessible for both the host and the GPU. In

WebGPU, we can copy data from an internal buffer to the staging buffer and map it into the host memory. We can also do the same in the opposite direction to copy data to the GPU. To copy data from internal buffers to the staging buffer, we request a buffer with usage set to `GPUBufferUsage.STORAGE | GPUBufferUsage.COPY_SRC`. To copy data to the staging buffer from the host, we request a buffer that can be mapped to host memory using `GPUBufferUsage.MAP_READ | GPUBufferUsage.COPY_DST`.

Enqueuing commands to the command buffer is a synchronous operation. The command is enqueued and executed at some point in the near future. To use the result of a command, we map the memory from the staging buffer to the host's memory using `mapAsync`. `mapAsync` waits for the completion of the commands using the requested buffer. To wait for all work submitted to the GPU to complete, we can use `onSubmittedWorkDone()`.

Chapter 3

Eviction Set Search in WebGPU

Recent works demonstrate that Prime+Probe attacks work on GPUs [13, 145]. A crucial part of Prime+Probe attacks is the finding of eviction sets. There is no eviction-set-finding algorithm for WebGPU, although algorithms for native applications, such as CUDA, have demonstrated effectiveness.

In this chapter, we explore the challenges of finding eviction sets in a restricted environment like the browser. We develop primitives to overcome these challenges, such as a counting thread and self-configuring components to automatically identify eviction sets across discrete GPUs from various generations, covering both Nvidia and AMD platforms. Our set-finding algorithm builds upon set-finding algorithms for CPUs and uses the parallelism of GPUs to improve its performance.

3.1 Primitives

We require several components to build a set-finding algorithm. First, we need a reliable timer. The timer must be precise enough to distinguish a cache hit from a cache miss. We then use the timer to detect cache activity. We determine the cache size by incrementally enlarging a buffer and observing the changes in its cache hitrate. An accurate timer and knowledge about the cache size allow us to build Prime+Probe eviction sets.

While such primitives have been extensively studied for CPUs, adapting them for GPU architectures poses notable challenges. Additional challenges emerge due to the assumption of a weak generic attacker from the browser.

3.1.1 Cache Timing Measurement

The WebGPU API uses several measures to prevent cache timing attacks. There is an optional `timestamp-query` feature. At the time of writing, major browsers either do not support this feature or only permit its use for experimental purposes. For example, Google Chrome supports the `timestamp-query` feature through the flag `--disable-dawn-features=disallow_unsafe_apis`. The WebGPU standard furthermore specifies a reduced precision of timestamp queries [20]. The `timestamp-query` is not available inside a WGSL shader. Instead, timestamps can only be recorded between GPU commands. Hence, at the time of writing, WGSL does not provide any timer functionality.

Furthermore, WebGPU restricts access to shared memory similarly to how browsers manage access to `SharedArrayBuffer`. This includes adhering to cross-origin policies, thus ensuring that shared memory is only accessible within the same origin.

JavaScript uses similar defenses against timing attacks, such as reducing timer precision. Previous research addressed this by using counting threads [92, 87, 13]. Generally, a counting thread continuously writes an incrementing value to a shared memory region. Another thread that requires timing information reads this value continuously and computes the relative difference between values to compare the duration of various operations. While the incremental value does not directly represent a real-time value, its consistent incrementing at regular intervals proves accurate enough for many applications, such as distinguishing between a cache hit and a cache miss. When implementing a counting thread in WebGPU, we face several challenges:

C1. Thread Serialization. With WebGPU, only one compute shader may run at a time. Consequently, the attack shader also needs to increment the counting variable simultaneously. While this task is straightforward on CPUs, a challenge arises on GPUs due to execution in *lockstep* within a workgroup [146]. This implies that while the shader can handle different tasks, such as executing the attack code and a counting thread, actual execution does not occur in parallel. Instead, diverging instructions result in sequential execution. While this may lead to poor performance for many practical applications, it also renders the counting thread useless, as the attack code depends on the counting thread incrementing the shared timer variable in parallel.

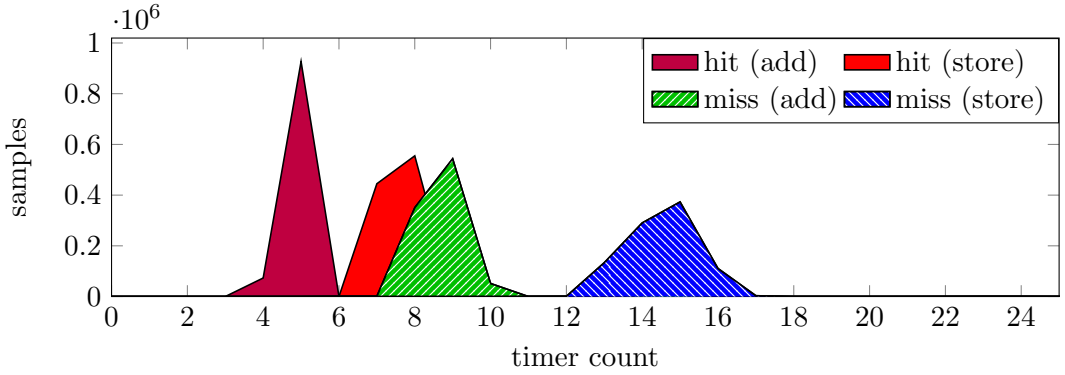
C2. Memory Coherency. There is no automatic memory coherency guarantee on GPUs in L0 and L1 caches. It is the responsibility of developers to ensure coherency through synchronization. Due to each SM operating with its dedicated memory subsystem, copies of the same data may exist in different L1 caches. Consequently, a counting thread might increment the timer variable in its own L1 cache, unobservable for other SMs.

C3. Compiler Optimizations. Before WGSL shaders are executed, the compiler applies aggressive optimizations. Instructions may be reordered or eliminated. The compiler replaces loops with their final results and substitutes memory accesses with registers when possible.

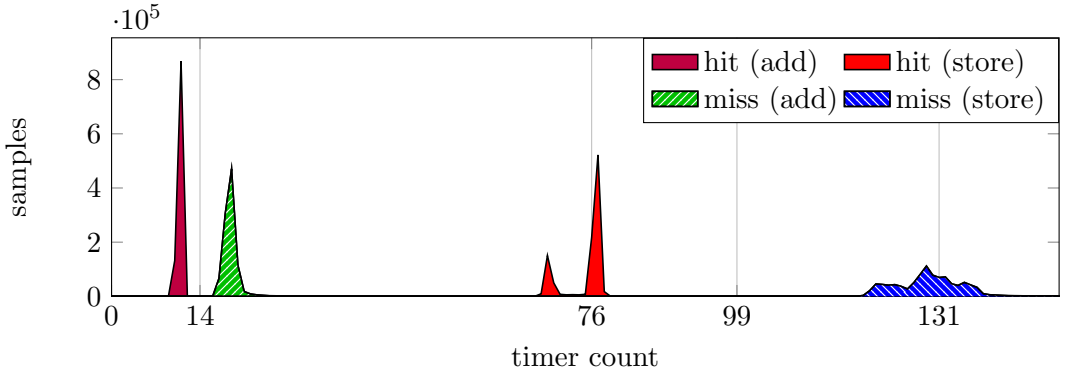
```
@binding(0) @group(0) var<storage, read_write> timer     : atomic<u32>;
@binding(1) @group(0) var<storage, read_write> stop      : atomic<u32>;
@binding(2) @group(0) var<storage, read_write> buffer    : atomic<u32>;
@binding(3) @group(0) var<storage, read_write> threshold : atomic<u32>;

@compute @workgroup_size(1)
fn main(@builtin(global_invocation_id) global_id : vec3<u32>)
{
    var threshold : u32 = atomicLoad(&threshold);
    storageBarrier();

    if global_id.x == 0 {
        var time : u32 = 0;
        atomicStore(&timer, 0);
        while (atomicLoad(&stop) != stop_value)
        {
          for(var a: u32 = 0; a < 100000; a++)
          {
            time++;
            atomicStore(&timer, time);
          }
        }
        return;
    }
    else {
        var time : u32 = atomicLoad(&timer); // before
        var c : u32 = atomicLoad(&buffer);
        if c != 0 { // prevent compiler optimization
          return;
        }
        time = atomicLoad(&timer) - time; // after
        if (time < threshold) {
            // cache hit
        }
        else {
            // cache miss
        }
        atomicStore(&stop, stop_value);
    }
}
```

Listing 3.1: Implementation of a counting thread in WGSL. The thread with global ID 0 incrementally updates a shared timer variable, facilitating timing measurement for other threads.

Solutions. Listing 3.1 shows our solution to the abovementioned challenges. Dutta et al. [13] execute the counting thread multiple times to fill a warp and guarantee that it is the only code executing on one processing unit. They execute the attack code on another processing unit within the same SM. This solution solves the first challenge since the threads on a processing unit execute the same code without diverging instructions.



(a) AMD RX 6800 XT



(b) NVIDIA RTX 2070 Super

Figure 3.1: Cache hit and cache miss histograms in WebGPU for different GPUs for 2 million samples. Storing data to a register offers better temporal precision than adding to a memory location. Higher counts show higher timer resolution [1].

We aim for a more general solution that does not rely on a specific number of threads to fill a warp. Therefore, we set the workgroup size to 1, as shown on line 6 in Listing 3.1. Consequently, dispatching the shader multiple times results in executions on different processing units. We dispatch the shader twice for the simple case, where we need one attacker thread and a counting thread. In the shader, we use the built-in variable `global_id` to distinguish between the counting and attacking threads.

We solve the second challenge by using atomic operations. While atomic operations guarantee that the compiler does not turn memory accesses into registers, they do not solve the third challenge entirely. We use a branch instruction depending on the value fetched at runtime to mitigate compiler optimizations, which would otherwise cause the removal of instructions with unused results. This can be seen on line 28 in Listing 3.1. Since the compiler cannot predict the outcome of this branch instruction, the load instruction is not removed in the optimization step.

To gracefully halt the shader, we use a shared `stop` variable. Given the expensive nature of loading the value atomically, we perform this operation only once every 1 000 000

Table 3.1: Comparison of cache hit and cache miss timing counter values at the 98th and 5th percentiles for the `add` and `store` methods across a variety of GPUs. A separable distribution yields a good threshold value. Total of $n = 1\,000\,000$ hits and misses recorded per GPU [1].

		Add		Store	
	GPU	$\text{hit}_{>98\%}$	$\text{miss}_{<5\%}$	$\text{hit}_{>98\%}$	$\text{miss}_{<5\%}$
AMD	RX 6800 XT	6	7	9	11
	RX 6900 XT	5	7	9	11
NVIDIA	GTX 1070	5	8	62	95
	GTX 1650	7	13	75	94
	GTX 1660 Ti	7	11	74	94
	GTX 1660 Ti Lin	4	7	10	18
	RTX 2070 SUPER	6	8	80	106
	RTX 2070 SUPER Lin	4	8	11	18
	RTX 3060 Mobile Lin	5	8	11	18
	RTX 3060 Ti	8	13	90	124
	RTX 3060 Ti Lin	5	7	11	20
	RTX 3080	8	12	95	119
	RTX 4090	7	10	99	145
	Quadro P620	5	7	61	88

increments. When the attacker thread is done, it writes a predetermined value to the `stop` variable to signal the counting thread to stop execution. When we require more attacker threads, each thread increments the `stop` value by one once it has finished its task. In such instances, the timing thread waits for the value of the `stop` variable to match the number of attacking threads.

3.1.2 Cache Hit-Miss Threshold

To distinguish a cache hit from a cache miss, we build upon our timing primitive to automatically find a threshold value. Therefore, we measure the access latency from memory locations that are highly likely cache misses and from cached memory locations. As can be seen on line 20 in Listing 3.1, we use `atomicStore` to store a register value to the shared memory location. We found that using `atomicStore` brings a higher timer resolution compared to `atomicAdd`, which can be seen in Figure 3.1. This occurs because `atomicAdd` requires loading the current value from memory before applying the increment. In contrast, `atomicStore` simply stores the current register value to the shared memory location. This optimization was first presented by Schwarz et al. [147] and exploits the fact that only the counting thread writes to the counter variable. We found that this optimization works best with Nvidia cards running on Windows. Table 3.1 compares different values for different architectures on Windows and Linux. Even in cases where

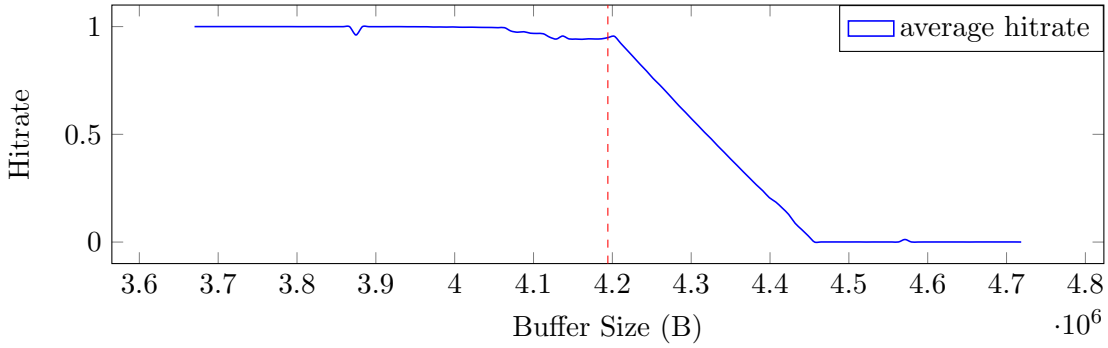


Figure 3.2: Average L2 hitrate for varying buffer sizes on a NVIDIA RTX 2070 Super. The red dotted line indicates the L2 cache size of 4 MiB.

Table 3.2: Our WebGPU cache-size detection algorithm on various GPUs ($n = 10$). The algorithm consistently identifies the correct cache size across all cards with one exception [1].

		Size			Runtime	
	GPU	Actual MB	Detected MB	Correct μ %	$\bar{x}$ ms	σ ms
AMD	RX 6800 XT	4.0	4.0	100	179.3	19.21
	RX 6900 XT	4.0	4.0	100	185.6	26.20
NVIDIA	GTX 1070	2.0	2.0	100	192.6	26.15
	GTX 1650	1.0	1.0	100	422.2	31.56
	GTX 1660 Ti	1.5	1.5	100	283.6	11.02
	RTX 2070 SUPER	4.0	4.0	100	189.9	6.15
	RTX 3060 Mobile	3.0	2.975	90	285.3	15.03
	RTX 3060 Ti	4.0	2.975	0	276.8	9.50
	RTX 3080	5.0	5.0	100	257.4	9.81
	RTX 4090	72.0	72.0	100	1729.6	60.23
	Quadro P620	1.0	1.0	100	251.7	23.25

we do not observe an improvement in resolution, the difference between a cache hit and a cache miss is significant enough to distinguish.

3.1.3 Cache Size Detection

A central goal of our set-finding algorithm is to make it as generic as possible. We aim to create WebGPU code with a minimal number of hardcoded values that works across various GPUs. We need to know how many sets we are looking for to find eviction sets, which we can infer from the cache size.

As the WebGPU does not provide functions to query the cache size, we use the primitives described above to detect the cache size of the underlying hardware automatically. The central idea is to fill a buffer and measure access latency in a second iteration. The access

latency increases when the buffer is larger than the cache size, as depicted in Figure 3.2. This simple method assumes an LRU replacement policy [111].

With this primitive, we conduct a binary search to determine the cache size. The result of the binary search is compared to known L2 cache sizes for common GPUs. We found this method effective for various GPUs, as shown in Table 3.2. For many of the tested GPUs, the cache size was detected correctly within approximately 300ms. The Nvidia RTX 3060 Ti is an outlier, as the detected cache size is 3 MB, despite the actual cache size being 4 MB. The straightforward explanation is that the card has a 3 MB L2 cache, contrary to the official specification. Alternatively, a more complex mapping function could be influencing cache behavior.

3.1.4 Noise

We observed periodic flushes of the full L2 cache during our experiments. The GPU flushes almost the entire cache when something is drawn on the screen. While we did not investigate the cause extensively, Giner et al. [1] show that it is possible to construct an inter-keystroke timing attack based on this observation. Because full evictions negatively impact our attack primitives, we increased the sample size to reduce the effects of this noise source.

3.2 Algorithm

We build upon the primitives described above to find eviction sets. An *eviction set* is a set of addresses that map to the same cache set and consequently fill the cache set. To replace all entries in the cache set, the size of an eviction set must be at least equal to the cache's associativity, denoted as $\mathcal{W}$.

Knowledge about the mapping function of virtual or physical addresses to cache sets simplifies the finding of eviction sets massively. Research on CPUs reverse-engineered how virtual and physical addresses map to cache sets on various CPUs. Jain et al. [111] reverse-engineered the cache mapping functions for an Nvidia GTX 1080 GPU and found that the cache addressing is much more complex than for traditional CPU caches. Furthermore, our tests confirmed that the mapping functions differ between generations of Nvidia GPUs. Given the variations in cache mapping functions among Nvidia GPUs, and even more so across manufacturers and their mobile variants, our generic algorithm does not rely on mapping functions.

Because GPU memory architecture is proprietary, and our goal is a universal solution, we cannot depend on contiguous memory, huge pages, or fixed page sizes. In WebGPU, we do not even have virtual or physical addresses, and the concept of pointers does not exist. Furthermore, we cannot use fine-grained cache control to assist the set-finding algorithm. Instead, our generic set-finding algorithm is based on prior work for CPUs [23, 148]. Here, only a timer accurate enough to distinguish cache hits from cache misses is required.

```
1  S ← {1.5x cacheSize, 128B steps}
2  P ← S[0], S ← S/P
3  while S > W
4    circularShift(S, 1/W+1|S|)
5    G ← S[0:1/W+1|S|]
6    access(P)
7    access(S\G)
8    if isNotCached(P)
9      S ← S\G // G does not change eviction of P
10 return S
```

Listing 3.2: Pseudo-code of the group-elimination method. We iteratively partition the set $\mathcal{S}$ into $\mathcal{W}+1$ groups, ensuring that at least one group does not affect the eviction of $\mathcal{P}$. The process terminates when $\mathcal{S}$ contains only $\mathcal{W}$ elements, representing an eviction set of $\mathcal{P}$.

3.2.1 Group-Elimination Method

The basis of our set-finding algorithm is the group-elimination method proposed by Qureshi et al. [23]. The code for the group-elimination method is illustrated in Listing 3.2. The group-elimination method uses the pseudo-LRU cache replacement policy to find congruent addresses for a target pivot address $\mathcal{P}$. We start by selecting a large set of addresses $\mathcal{S}$, significantly larger than $\mathcal{W}$, ensuring that $\mathcal{S}$ evicts the target address. Next, we partition $\mathcal{S}$ into $\mathcal{W}+1$ groups. Each of these groups contains approximately the same number of addresses.

Since $\mathcal{S}$ contains at least $\mathcal{W}$ addresses that evict the target address, we know that at least one out of the $\mathcal{W}+1$ groups does not influence the eviction. The group-elimination method removes one group at a time and checks whether the target address is still evicted. The remaining groups form the new search set $\mathcal{S}$. We repeat the steps until only $\mathcal{W}$ addresses remain in $\mathcal{S}$. These $\mathcal{W}$ addresses form an eviction set for the target address.

While this algorithm works well to find an eviction set for one target address, our goal is somewhat different. We want to find all eviction sets in the L2 cache, so we parallelize the group-elimination method to improve its performance. We reach an increased performance not only by utilizing the parallelism of GPUs but also by using the predictable behavior of LRU, similar to Prime+Prune+Probe [148].

Some constants presented in the following section were found empirically and are not optimal values. However, these values work well for various GPUs. When referring to addresses in the following sections, we refer to offsets within a WGSL buffer.

3.2.2 Parallel Eviction-Set Finding

The basis for the group-elimination method is pseudo-LRU cache replacement behavior. The naive approach to parallelizing the group-elimination method is distributing the computation among multiple threads. However, when we access addresses from the same

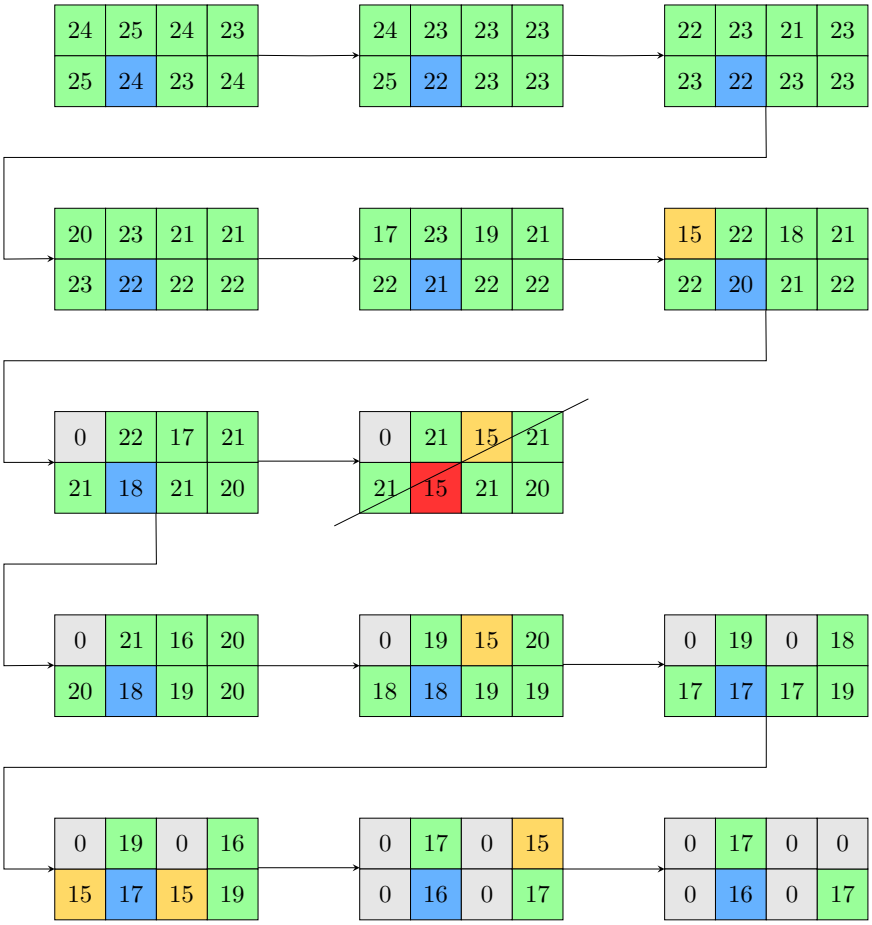


Figure 3.3: Constructing the first bucket containing 3 full eviction sets. First, a pivot $\mathcal{P}$ is chosen. The set containing $\mathcal{P}$ is blue; all other eviction sets are green. In each step, $1/32$ of all addresses are removed. We remove addresses that are cache hits, as they are not part of a full eviction set anymore (yellow). If the pivot $\mathcal{P}$ is still part of a full eviction set, we continue with the next step. If $\mathcal{P}$ is not evicted (red set after 7 iterations), the step is reverted. The step is repeated until only a certain number of addresses remain. The pivot $\mathcal{P}$ guarantees that at least one eviction set remains; however, with a high probability, more eviction sets remain in the bucket.

set from different threads, we can no longer rely on the access order and, therefore, lose the LRU behavior.

Therefore, we introduce a pre-processing step to split the addresses into buckets with non-overlapping sets. Afterward, we can independently search for eviction sets in the buckets using multiple threads.

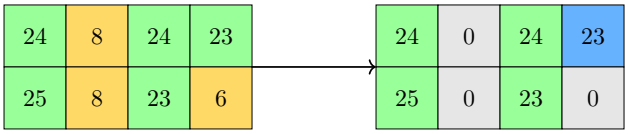


Figure 3.4: Pre-processing step for new bucket. Addresses that are cache hits are not part of a full eviction set and can be removed from the search space. With an initial search space of 1.5x cache size, it is highly unlikely that the remaining addresses, which map to an eviction set found in one bucket, will form a full eviction set in another bucket. Therefore, the buckets contain non-overlapping eviction sets.

Step 1: Buckets

The goal of the first step is to separate eviction sets into different buckets. Each bucket contains eviction sets, with the assurance that all addresses within a set exclusively belong to that particular bucket. We begin with a higher-level description of the algorithm and discuss the details afterward.

High-level Description. Figure 3.3 shows the separation of a cache with 8 sets with 16 ways into buckets containing at most 50 addresses. For our initial set of addresses $\mathcal{S}$, we chose addresses ranging over 1.5x the cache size in 128 B steps. This results in about 24 addresses per cache set. We chose one pivot address to guarantee that at least one eviction set is part of the bucket. However, the probability that more eviction sets are within each bucket is high. The set containing the pivot address is depicted as the blue square, and other sets of addresses forming a complete eviction set are depicted as green squares. The number within each square represents the number of addresses within that set. In each step, we remove $1/32$ of the addresses. In the case of this example, this results in the removal of 6 addresses per step. After every removal step, we check for hits in the remaining addresses. We remove the hits since they are no longer part of a full eviction set. We can see this, for example, after 5 steps in Figure 3.3, where the first set only contains 15 addresses and does not form a full eviction set anymore. We remove these partial sets except for the set containing the pivot address. After step 7, we see that the set containing the pivot address is no longer a full eviction set. Since we want at least to keep the pivot set in our bucket, we revert the step and remove other addresses. After several rounds of removing addresses, we end up with addresses that form our first bucket, containing 50 addresses that form 3 eviction sets.

For the search for the next bucket, we repeat the process with all addresses not contained in the first bucket. Note that at the beginning, these addresses still contain addresses of sets from the first bucket. However, Figure 3.4 shows that these addresses cannot form a full eviction set with a high probability when we chose 1.5x the cache size as our initial search space. Before searching for the next bucket, we remove addresses that are cache hits and continue like before by selecting a new pivot.

```
1  S ← {1.5x cacheSize, 128B steps}
2  Buckets ← {{}}
3  while S != {}
4    B ← S, P ← B[0]
5    B ← B\P
6    while |B| > targetSize
7      G ← {}
8      hits ← {}
9      do
10       shuffle(B)
11       B ← B ∪ G
12       G ← B[0:1/2w|B|]
13       B ← B\G
14       if |B| > 1/6|S| (1)
15         access(P), parallelAccess(B)
16         if isNotCached(P) and hybridCondition() (1b)
17           hits ← accessAndMeasure({P ∪ B})
18       else (2)
19         hits ← accessAndMeasure({P ∪ B})
20     while isCached(P)
21     B ← B\hits (3)
22   B ← B ∪ P
23   Buckets ← Buckets ∪ {B}
24   S ← S\B (4)
25   hits ← accessAndMeasure(S)
26   S ← S\hits
27 return Buckets
```

Listing 3.3: Pseudo-code dividing set of addresses into buckets with non-overlapping eviction sets. We iteratively remove addresses from $\mathcal{B}$ and check if $\mathcal{P}$ is still part of a full eviction set. If $\mathcal{P}$ is not evicted, we add $\mathcal{B}$ again, ensuring that at least an eviction set for $\mathcal{P}$ remains in the bucket. To check whether $\mathcal{P}$ is evicted, we access other addresses either in parallel (❶) or serially (❶b, ❷). Serial access gives us information about the eviction of each address, so we can remove addresses that are not part of an eviction set anymore (❷). Once $\mathcal{B}$ reaches a target size, we continue with the construction of the next bucket, continuing with the remaining addresses.

It is not necessary to know the number of addresses within each set. The information that some addresses are still in the cache after accessing all other addresses is enough to infer that these addresses do not form a full eviction set anymore and can safely be removed. This is important because the performance of this algorithm stems from the fact that, at this point, we do not care about actual eviction sets. The only goal is to separate a set of addresses into buckets with non-overlapping sets, which we can later use to find the actual eviction sets in parallel.

Implementation Details. Now that we understand the high-level abstraction of separating addresses into buckets, we dive into the details. Listing 3.3 shows the pseudo-code for this step. We start with a large set of addresses $\mathcal{S}$ (1.5x cache size, 128 B steps) that evicts the whole cache. Next, we select a pivot address $\mathcal{P}$ and fill the current bucket $\mathcal{B}$ with all addresses in $\mathcal{S}$. If the current bucket is larger than the target size, we remove addresses iteratively. In contrast to the group-elimination method, we remove $1/2W|\mathcal{B}|$ instead of $1/W + 1|\mathcal{B}|$, which prevents the removal of too many addresses in a later step (3). Depending on the current size of the bucket, we differentiate between three cases:

Parallel Access for $|\mathcal{B}| > 1/6|\mathcal{S}|$ (1). If the current size of the bucket is larger than $1/6|\mathcal{S}|$, we access $\mathcal{P}$ and then all other addresses in parallel. We then check if the remaining addresses evicted $\mathcal{P}$. In that case, we know that removing $\mathcal{G}$ did not affect the eviction, and we can continue removing other addresses. This is very similar to the group-elimination method. This step is fast since we access all addresses but $\mathcal{P}$ in parallel.

Serial Access for $|\mathcal{B}| \leq 1/6|\mathcal{S}|$ (2). If less than $1/6|\mathcal{S}|$ addresses are in the current bucket, we access all addresses serially. If $\mathcal{P}$ is not cached, we can safely remove $\mathcal{G}$ again. While this is slow compared to parallel accesses, we gain an essential advantage. Having the timing information for all addresses, we can remove all addresses that are cache hits from the current bucket since they are not part of full eviction sets anymore (3).

Hybrid Transmission from Parallel to Serial Access (1b). We found that transitioning from parallel to serial access mode in a hybrid manner enhances the efficiency of this algorithm. If $\mathcal{P}$ is still evicted after removing $\mathcal{G}$ and accessing the remaining addresses in parallel, we check for the `hybridCondition()`. The `hybridCondition()` is true on every fourth iteration if $|\mathcal{B}| < 3/4|\mathcal{S}|$. In that case, we additionally access and measure all addresses serially like in 2 to gain timing information and remove all hits in a later step 3.

When $|\mathcal{B}|$ is below the target size, we add the pivot $\mathcal{P}$ to the bucket. Then, the newly found bucket is added to the set of buckets. We remove the addresses contained in this bucket from the set of remaining addresses (4). Additionally, we check if there are cache hits in the set of remaining addresses. Since these addresses are not part of any eviction set, we remove them from the set of remaining addresses. We continue searching for more buckets as long as we have addresses in $\mathcal{S}$.

Empirical tests have shown that a bucket size of 3500 addresses works well across various devices. This is equivalent to around 146-206 eviction sets per bucket.

Step 2: Sets

The algorithm from Step 1 yields a separation of addresses into buckets with non-overlapping eviction sets. With non-overlapping sets, we can search for eviction sets in

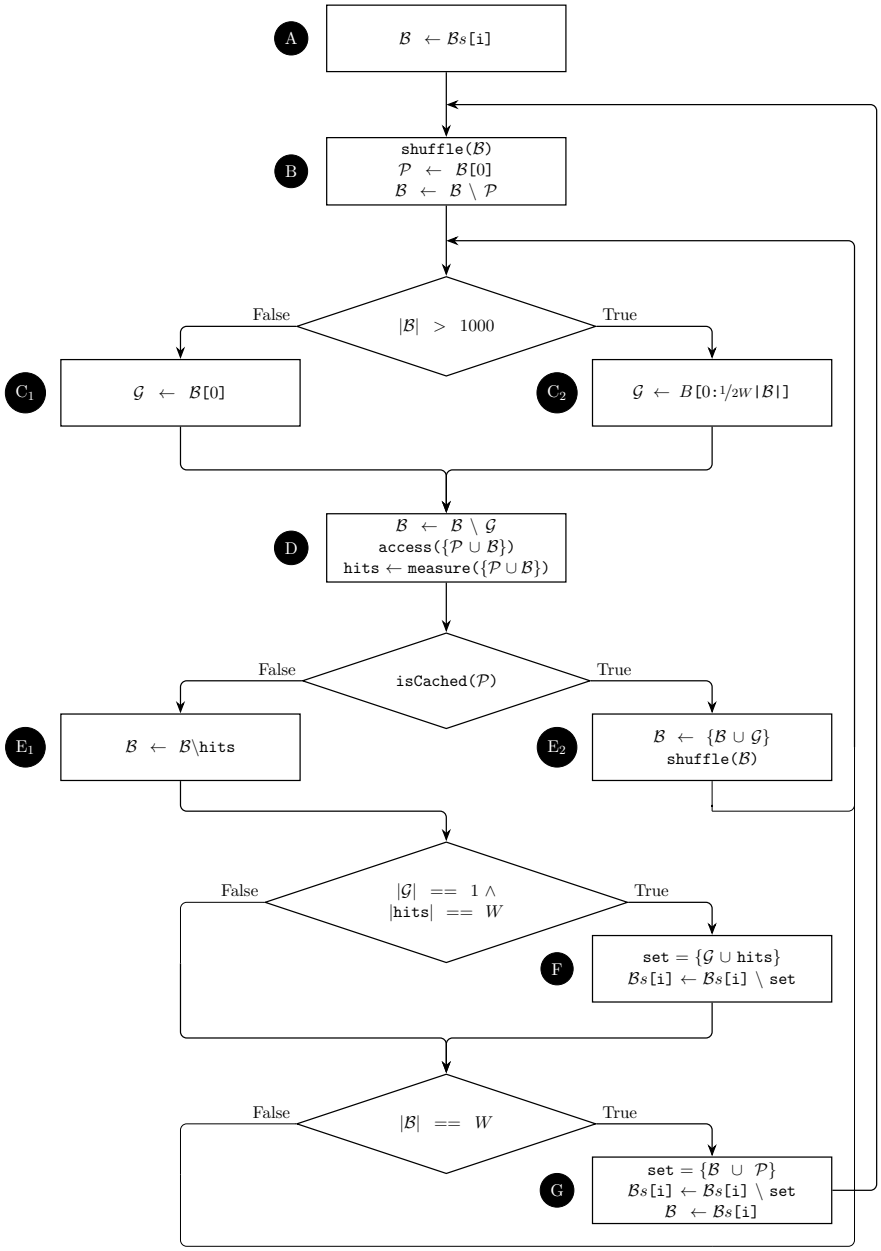


Figure 3.5: Flow chart of set search within one bucket.

each bucket independently, as accesses to one bucket do not evict addresses from another bucket. Thus, we can rely on LRU cache eviction. Only the `access` and `measure` parts of the algorithm are executed on the GPU.

Figure 3.5 depicts a flowchart of the parallel set-finding algorithm for one bucket. Initially, we fill $\mathcal{B}$ with one bucket from the algorithm described in Step 1 (A). One bucket contains around 3500 addresses, equivalent to 145-206 sets. On an Nvidia RTX

2070 Super, we start with 13-14 buckets, which are searched in parallel. The exact number of buckets depends on the results from Step 1.

Similar to Step 1, we select one pivot address $\mathcal{P}$ (B). Subsequently, we look for a set that evicts $\mathcal{P}$. Depending on the size of the bucket $\mathcal{B}$, we remove either one address (C_1) or a fraction ($1/2\mathcal{W}$ of $|\mathcal{B}|$, C_2) from $\mathcal{B}$, storing the removed addresses in $\mathcal{G}$. The threshold value for this decision is 1000 addresses. While this is not an optimal value, we found that it works well across various GPUs. We remove only one address when a few remain in the bucket because it allows us to find many sets for free at a later step (F).

After removing $\mathcal{G}$, we run the WebGPU shader in parallel with the other instantiations of this algorithm (D). The shader invocation is the computationally expensive part of this algorithm and a synchronization point. As a result, we get information about cache hits and misses for all addresses within the bucket. Note that this is only possible because the other instantiations of this algorithm work on different buckets containing addresses that do not influence cache sets of other buckets. So, even though we access addresses in parallel, we can rely on LRU eviction within each bucket.

Using the result of the shader, we check whether $\mathcal{P}$ is cached. If $\mathcal{P}$ is indeed cached, removing $\mathcal{G}$ from $\mathcal{B}$ yielded an unwanted outcome: $\mathcal{B}$ does not contain an eviction set for $\mathcal{P}$ anymore. We add $\mathcal{G}$ to $\mathcal{B}$ again and try again removing different addresses (E_2).

This is repeated until we remove a set $\mathcal{G}$ from $\mathcal{B}$, after which $\mathcal{B}$ still contains an eviction set for $\mathcal{P}$. We can then remove all addresses that are cache hits from $\mathcal{B}$ since they are not part of a full eviction set anymore (E_1).

Then, a vital optimization step comes into play. When we remove one address and, consequently, $\mathcal{W}$ hits occur, we find an eviction set for free. This set consists of the removed address in $\mathcal{G}$ and the addresses that are cache hits after removing $\mathcal{G}$. After checking if $\{\mathcal{G} \cup \texttt{hits}\}$ is indeed an eviction set, we store the newly found set and remove it from the initial bucket $\mathcal{B}s$`[i]` (F).

Similar to the group-elimination method, we check if we have $\mathcal{W}$ addresses remaining in $\mathcal{B}$. In that case, we have found an eviction set that evicts $\mathcal{P}$. We verify once more if the newly discovered set $\{\mathcal{B} \cup \mathcal{P}\}$ indeed is an eviction set for $\mathcal{P}$, and store the set upon confirmation (G).

Finally, we choose a new $\mathcal{P}$ and start again. Note that Figure 3.5 does not include an exit condition for the algorithm for simplification. However, we stop the search once the bucket is empty or no more sets can be found.

3.3 Implementation

As mentioned, large parts of the algorithm are not executed on the GPU. The goal of the implementation is to minimize noise sources. Since every memory access within the shader causes cache activity, we only use a GPU shader for measurements. All other computations are done in JavaScript.

We observed significant noise levels when implementing our cache attack, which posed a substantial challenge. For example, we observe periodic evictions of almost the entire L2 cache, which we attribute to screen updates rendered by the GPU. We access all

Table 3.3: Evaluation of set-finding algorithm on various GPUs ($n = 10$). More than 80% of sets found on all GPUs but one [1].

		Sets			Runtime	
	GPU	Overall	Found $\bar{x}$ %	Found σ %	$\bar{x}$ min	σ min
NVIDIA	GTX 1070	1024	96.0	2.1	11.8	4.8
	GTX 1650	512	82.9	2.2	4.2	3.9
	GTX 1660 Ti	768	96.4	2.1	12.1	3.8
	RTX 2070 SUPER	2048	98.7	1.0	7.0	2.1
	RTX 3060 Mobile	1536	99.9	0.1	2.3	0.4
	RTX 3060 Ti	1536	94.5	5.3	2.6	1.5
	RTX 3080	2560	99.3	1.9	2.8	1.2
	Quadro P620	512	50.8	24.5	13.7	9.0

addresses in a pointer-chase style to eliminate the unwanted effect of cache accesses from address lookups. As suggested by prior work on CPUs and GPUs [149, 150, 112], we use pointer-chasing to reduce the effect of address lookups on the cache. In pointer-chase, we traverse an array whose elements are initialized as the indices for the next memory access [112]. Since we use atomic memory accesses and only know the following address after loading the current one, we do not need additional memory barriers. Naturally, we cannot get rid of all other cache activity. For example, reading shader code evicts some cache lines. To reduce the effect of noise, we take several samples per measurement.

3.4 Evaluation

In this section, we evaluate the performance of the eviction-set-finding algorithm on various Nvidia GPUs. Furthermore, we discuss empirically determined values used for the separation of addresses into buckets with non-overlapping eviction sets as discussed in Section 3.3.

3.4.1 Set Search

Our set-finding algorithm finds eviction sets on most of the tested GPUs in less than 5 minutes, as shown in Table 3.3. However, the set-finding was not successful on AMD cards. We suspect that on AMD GPUs, the measured difference between a cache hit and a cache miss value from our timing primitive is too small, as detailed in Table 3.1. Nevertheless, we expect that finding eviction sets with our algorithm on AMD cards is also possible. For example, choosing a higher sample size could be sufficient to find sets on AMD GPUs as well. However, because access to AMD GPUs was restricted, we did not pursue further investigation into this matter.

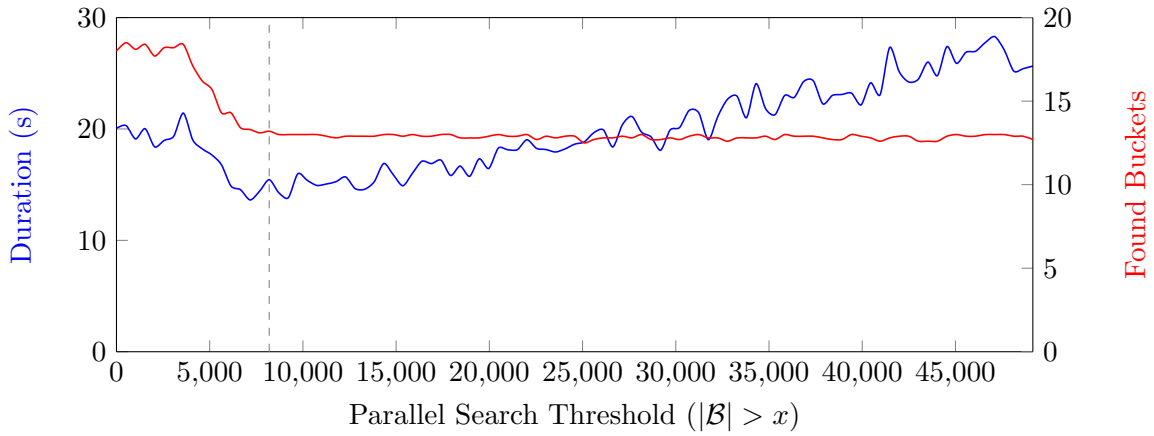


Figure 3.6: Comparative analysis of accuracy and duration of different threshold values for parallel bucket search. If more than x values are in the current bucket, the algorithm accesses addresses in parallel, which introduces noise but improves performance. The goal is separation of $\mathcal{S}$ into 13-14 buckets; different values come from high noise levels. Threshold of $1/6|\mathcal{S}|$ (gray dotted line) gives consistent results with good performance on various devices. Evaluation on Nvidia RTX 2070 Super ($n = 10$).

The optimization step allows us to find sets for "free" and improves the algorithm's efficiency. For instance, on the Nvidia RTX 2070 Super, which has 2048 L2 cache sets, we divided all addresses into 13-14 buckets, each containing approximately 3500 addresses. We find about 1950 sets for "free", using only about 100 pivot addresses. This means that around 95 % of all sets were found indirectly as a side-product of the actual search.

3.4.2 Parameter Evaluation Bucket Separation

We use several empirically determined values that work for various GPUs to separate addresses into buckets. As these values are not tailored for specific devices, they are not optimal.

Parallel Access Threshold. As described in Section 3.2.2, we differentiate between three cases to reduce the size of the current bucket. Accessing the addresses in the bucket in parallel improves performance at the cost of lower accuracy of the results. Figure 3.6 compares different threshold values for parallel access. All memory accesses are serial if the number of remaining addresses in a bucket is below the threshold value. For the values above, we access all addresses in parallel. However, we add a serial access on every 4th iteration for the hybrid mode. This helps to mitigate the noise introduced with parallel accesses. We found that switching to serial-only access when less than $1/6|\mathcal{S}|$ addresses remain in the bucket yields good performance and consistent results on various devices.

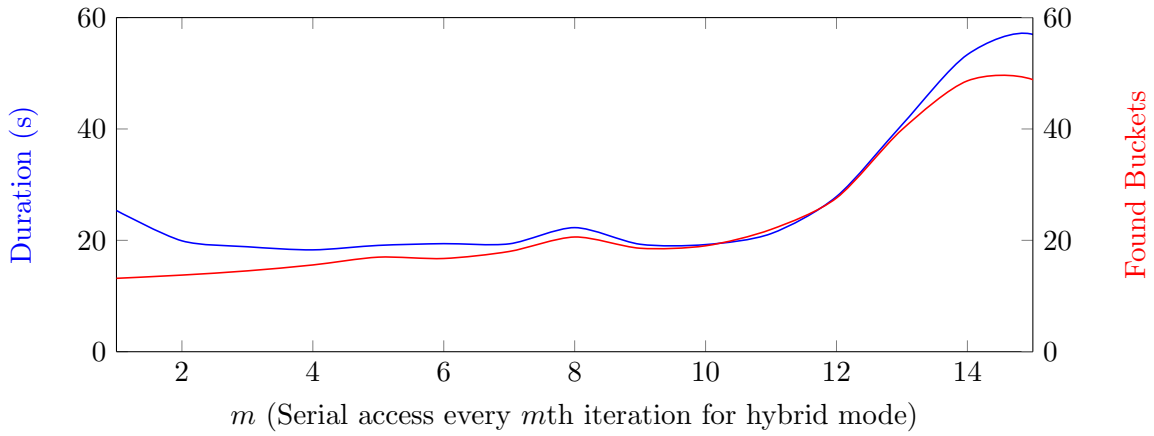


Figure 3.7: With hybrid mode, we add a serial access step while parallel probing for every mth iteration. Comparison of accuracy and duration for different values for m. Goal is separation into 13-14 buckets, different values from high noise levels. Accessing all addresses serially every 4th iteration works consistently on various devices. Evaluation on Nvidia RTX 2070 Super ($n = 50$).

Hybrid Mode. Accessing all addresses in parallel improves performance but yields inaccurate results. While this does not affect the separation of buckets if the current bucket size is larger than $3/4|\mathcal{S}|$, we introduce a serial access for every mth iteration of the parallel mode. This hybrid mode is described in detail in Section 3.2.2. Figure 3.7 compares different values of m. Adding a serial access step for every 4th parallel access works well across various devices.

Chapter 4

Covert Channel

A *covert channel* refers to a method of communication that involves exploiting shared system resources, using them in unintended ways to transmit data. Attackers often use these channels to transmit data between domains that are either isolated or under strict surveillance, thereby violating the system's security policy. In such channels, the sender and receiver typically collaborate.

One example of using a covert channel is to exfiltrate sensitive data. An attacker may control an application with access to sensitive data but lacks internet access. The attacker may now lure the victim to a website with the receiver program in JavaScript. Detecting covert channels can be challenging, especially when the communication occurs through a channel that is not architecturally visible. The collaboration between sender and receiver in a covert channel also makes it an effective tool for measuring the bandwidth of any side channel.

In a Prime+Probe covert channel, the sender and receiver operate on one or more eviction sets to transmit data. Before transmitting data, both parties need to agree on the set of eviction sets used for transmission. After that, the sender *primes* (evicts) an eviction set to transmit a binary 1 or leaves the set as it is to transmit a binary 0. The receiver then *probes* the eviction set. When the receiver detects an eviction, it infers that the sender transmitted a binary 1. Conversely, the receiver registers a binary 0 if the probed set is still cached.

4.1 Construction

To construct a cache covert channel, we first address the challenge of synchronizing both parties, a process detailed in Section 4.1.1. Next, the sender $\mathcal{S}$ and receiver $\mathcal{R}$ establish a shared set of eviction sets for communication, as outlined in Section 4.1.2. We then use these eviction sets to transmit data, with our method presented in Section 4.1.3. Finally, the implementation details are discussed in Section 4.1.4.

4.1.1 Synchronization

We synchronize the sender and receiver using the system's wall clock. While the accuracy of the wall clock is limited to 100µs, this is not a significant limitation for our covert channel. The invocation of a shader takes approximately 3ms, overshadowing the

inaccuracy of the wall clock. For this reason, the accuracy of the wall clock is sufficient for our covert channel.

Furthermore, using a common clock signal simplifies synchronization. For the first stage of the channel, where we transmit the eviction sets used for message transmission, we synchronize using the start of even-numbered seconds. The second stage, where the message is transmitted, runs immediately after the first stage. We use one more synchronization point at the start of a full second and compute each message package's start and end times using a predetermined package window length.

4.1.2 Set Transmission

Sender $\mathcal{S}$ and receiver $\mathcal{R}$ agree upon some constant parameters beforehand. These include the number of sets used for communication (1024) and the duration of a single package transmission, the package window length.

To establish a communication channel, sender $\mathcal{S}$ and receiver $\mathcal{R}$ need to agree upon a set of eviction sets used in the communication. Neither $\mathcal{S}$ nor $\mathcal{R}$ know absolute labels for their respective eviction sets. Hence, we implement CJAG (Cache-based Jamming Agreement) as described by Maurice et al. [26], with modifications tailored to optimize performance on GPUs. In CJAG, $\mathcal{S}$ continuously *jam*s (evicts) and probes a set. At the same time, $\mathcal{R}$ probes all sets continuously. Once $\mathcal{R}$ detects the eviction of a set, it starts to *jam* the *detected* set. $\mathcal{S}$ then recognizes the detection in its probe phase and continues with *jamming* the next set to be transmitted to $\mathcal{R}$.

CJAG on CPUs works in a way where $\mathcal{S}$ and $\mathcal{R}$ are executed simultaneously and continuously. However, on GPUs, only one compute shader may run simultaneously. Since compute shaders may run for longer periods of time, there is no assurance that both the sender and receiver shaders are scheduled in a manner conducive to the synchronized jam and detect cycle. Therefore, we invoke the shaders for jamming and detection in each iteration rather than executing them continuously. Unfortunately, this introduces additional overhead. We found that invoking a shader in WebGPU and waiting for its termination takes around 3ms on an Nvidia RTX 2070 Super. We observe similar values for other GPUs. We also observe an increased overhead when we copy buffers larger than 200 kB to and from GPU buffers. We investigate this further in Section 4.2. The buffer used to evict the cache on the Nvidia RTX 2070 Super is 6 MB, which consists of 49 152 offsets with a stride of 128 B. Since we are only interested in whether the offsets are cache hits or cache misses in each iteration, we copy a buffer with around 200 kB after each shader invocation. As mentioned before, a buffer copy of this size only marginally increases the duration of each invocation.

Since we invoke the shader for each *prime* and *probe* step, we face a hard limit of 3ms per package transmission. However, we mitigate this limitation using multiple threads with the parallel *priming* and *probing* of 1024 sets.

For our modified CJAG protocol, we use the eviction sets obtained through the algorithm described in Chapter 3. We additionally obtain all eviction sets in CUDA using the algorithm described by Liu et al. [80].

```
1 S ← Sets
2 T ← {} // sets used for message transmission
3 while |T| < 1024
4   C ← S[0:1023]
5   sync()
6   jamParallel(C)
7   T ← detectParallel()
8   L = C \ T // sets lost in transmission
9   S ← S \ L
```

Listing 4.1: Sender Pseudo-code of parallel CJAG

```
1 T ← {}
2 while |T| < 1024
3   sync()
4   T ← detectParallel()
5   jamParallel(T)
```

Listing 4.2: Receiver Pseudo-code of parallel CJAG

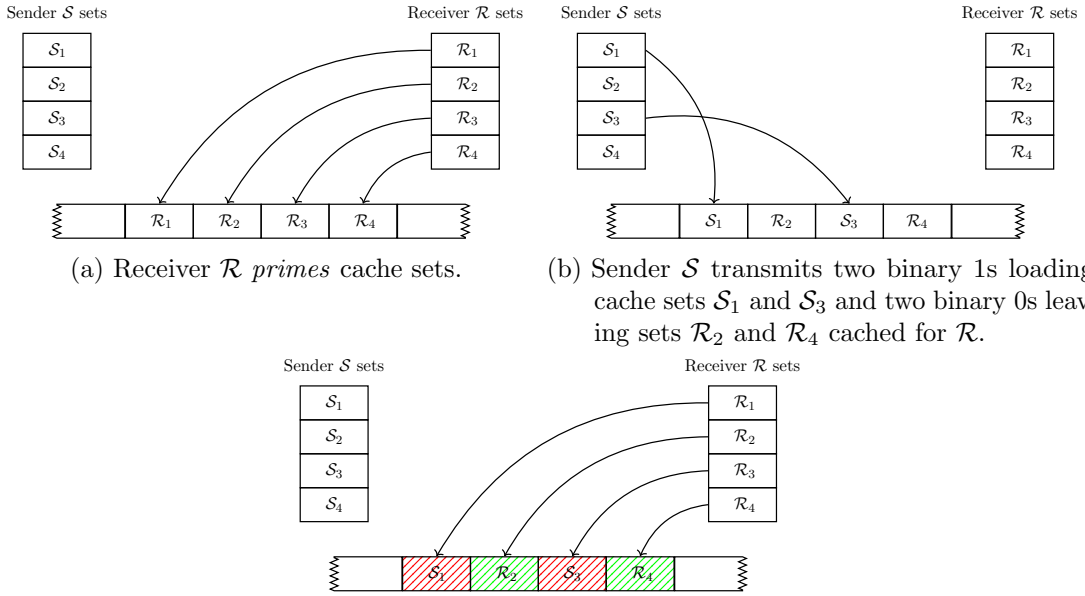
$\mathcal{S}$ chooses several sets they want to use for communication with $\mathcal{R}$. As described before, $\mathcal{S}$ waits for the start of a full even second to begin with the *jamming*. We use many sets, so transmitting them serially, like in the traditional CJAG, is not feasible. Therefore, we *jam* and *detect* all sets simultaneously. This is feasible because we leverage the parallelism of GPUs for both $\mathcal{S}$ and $\mathcal{R}$. We opted to use 64 sets per shader invocation. This approach allows us to simultaneously *jam* and *detect* 1024 sets using 16 threads in parallel.

Once $\mathcal{R}$ detects sets, it starts *jamming* these sets. This serves as an acknowledgment for $\mathcal{S}$. We observe that $\mathcal{R}$ does not always detect all sets. Therefore, the transmission of sets is an iterative process, where $\mathcal{S}$ swaps out the sets not detected by $\mathcal{R}$. This ensures that only sets found by both parties are used for communication. Listings 4.1 and 4.2 depict the pseudo-code for $\mathcal{S}$ and $\mathcal{R}$ respectively.

With our solution of parallel transmission of sets, another challenge arises. In contrast to CJAG, where sets are transmitted serially, we no longer receive the sets in a specific order. Of course, the order of sets is essential for transmitting messages. Therefore, after

```
1 i ← 0
2 T ← transmissionSets
3 while i < log2(1024)
4   T_i ← {T[n] | n ∈ {0, 1, ..., |T| − 1} ∧ (n & (1 ≪ i)) = 1}
5   sync()
6   jamParallel(T_i)
7   i ← i + 1
```

Listing 4.3: Sender Pseudo-code set-order transmission.



(a) Receiver $\mathcal{R}$ *primes* cache sets.

(b) Sender $\mathcal{S}$ transmits two binary 1s loading cache sets $\mathcal{S}_1$ and $\mathcal{S}_3$ and two binary 0s leaving sets $\mathcal{R}_2$ and $\mathcal{R}_4$ cached for $\mathcal{R}$.

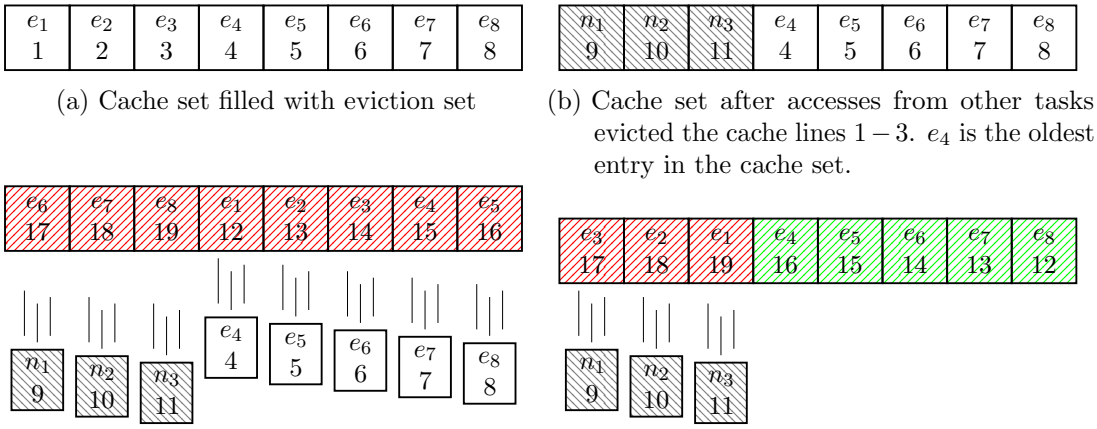
(c) Receiver probes cache sets. Sets $\mathcal{R}_1$ and $\mathcal{R}_3$ are cache misses, and thus binary 1s. $\mathcal{R}_2$ and $\mathcal{R}_4$ are cache hits, and are recorded as binary 0s.

Figure 4.1: Transmission of data using a Prime+Probe covert channel. Since $\mathcal{S}$ and $\mathcal{R}$ agreed upon the order of sets, they can communicate a 4-bit value in this simplified example.

successfully transmitting all sets, $\mathcal{S}$ starts to transmit the order of the sets. This works as follows: $\mathcal{S}$ *jam*s only half of the sets. The selection of sets to be *jammed* depends on the current bit in the index number of each cache set. For example, the set with the index number $816 = b1100110000$ is jammed in the set order transmission's 1st, 2nd, 5th, and 6th iteration. This enables a transmission of the set order from $\mathcal{S}$ to $\mathcal{R}$ in 10 iterations, corresponding to $log_2(1024)$. Listing 4.3 shows the sender part of the set-order transmission.

4.1.3 Message Transmission

Once $\mathcal{S}$ and $\mathcal{R}$ agreed upon the sets used for the communication channel, the data transmission starts. If $\mathcal{S}$ wants to transmit a binary 1, it fills the corresponding cache set. Conversely, to transmit a binary 0 $\mathcal{S}$ does not access addresses of the corresponding cache set. $\mathcal{R}$ continuously *primes* all cache sets, waits for a short period, and then checks if the sets are still cached. $\mathcal{R}$ registers a binary 1 for evicted sets and a binary 0 for sets still residing in the cache. Figure 4.1 depicts the transmission of 4 bits over a covert channel using four sets.



(a) Cache set filled with eviction set

(b) Cache set after accesses from other tasks evicted the cache lines $1-3$. e_4 is the oldest entry in the cache set.

(c) Evicted cache after self-eviction. e_1 replaced e_4, e_2 replaced e_5 and so on. All accesses to e_i are cache misses.

(d) Accessing the eviction set in reverse for *probing* gives us the exact number of evicted addresses of e.

Figure 4.2: Comparison of same-order and alternating-order access for Prime+Probe for an 8-way cache set. Eviction set addresses are denoted as $e_{1...8}$, addresses from unrelated tasks as $n_{1...i}$. Numbers in the lower half represent the timestamps used for LRU. Accessing the eviction set in the same order (4.2c) leads to self-evictions. Back-forth order allows counting the number of evicted addresses (4.2d).

We opted for a one-way channel. Since there is no backchannel, we use a fixed length window of around 5ms for each data package. We evaluate different window lengths in Section 4.2. As described in Section 4.1.1, we synchronize $\mathcal{S}$ and $\mathcal{R}$ using the system's wall clock.

As discussed before, we observe lots of noise on the cache lines. To reduce the effects of noise, we use three techniques. First, we use a *majority vote* measurement. When multiple packages arrive within the duration of a package window and contain different values, the value received most frequently is considered.

Next, we access all addresses in alternating order to prevent self-evictions, as depicted in Figure 4.2. Self-evictions occur when we prime a cache set, and subsequent unrelated cache accesses evict some primed addresses in LRU order. Reaccessing the set for probing and starting with the first primed address triggers the eviction of the oldest cache entry, which is also among the next addresses to be probed. This sequence creates a domino effect of evictions. While probing indicates a complete set eviction, a single external access can initiate this eviction cascade.

By probing in reverse access order, the first address probed is the last one that was primed. Thus, probing is not affected by self-evictions. This reverse order probing, proposed by Tromer et al. [84], helps us count the evicted addresses accurately. With this information, we can filter out low-level noise and establish a threshold for the number

Table 4.1: Transmission results of our covert channel on three Nvidia GPUs. Transmission from native CUDA to WebGPU receiver in the browser, $n = 10$ [1].

		Configuration			CJAG	Bandwidth			Bit-Error-Rate	
	GPU	Sets_{tx}	Window ms	Reads/window $\bar{x}$	Set Tx s	Raw Byte/s	True $\bar{x}$ Byte/s	True σ Byte/s	$\bar{x}$ %	σ %
NVIDIA	RTX 2070 Super	1024	6.0	3.3	14.6	10 666.7	8963.0	360.8	2.4	0.7
		1024	5.0	2.2	14.0	12 800.0	8962.0	286.5	5.3	0.6
	RTX 3060 Ti	1024	6.0	2.9	16.4	10 666.7	9004.9	271.4	2.3	0.5
		1024	5.0	1.9	15.7	12 800.0	7272.0	1252.5	9.1	3.1
	RTX 3080	1024	5.0	2.7	27.8	12 800.0	10 897.5	698.3	2.2	1.0
		1024	4.0	1.9	28.2	16 000.0	5964.5	1048.6	15.9	2.7

of evictions required to be counted as a full eviction, thus enabling more fine-grained control.

The third technique we use to deal with noise is *differential transmission*. We use 2 sets for each transmitted bit. One set conveys the actual bit, and the other set the inverse. The transmission is discarded as noise if both sets convey the same information. The technique prevents periodic full cache evictions from being received as erroneous information. However, this effectively reduces the throughput to a maximum of 64 B per package for 1024 sets.

4.1.4 Implementation

We implement the sender in a native C++ application with CUDA. With CUDA, we can use the native high-resolution hardware timer. The disadvantage of CUDA is that it is only available for Nvidia. Furthermore, as our set-finding algorithm does not work for AMD GPUs, we can only construct a covert channel for Nvidia GPUs. However, since the presented concept does not rely on features exclusive to Nvidia GPUs, we assume that the same concept also works on AMD GPUs.

The receiver is implemented in JavaScript and WGSL. Like the set-finding implementation, most parts of the receiver algorithm run in JavaScript on the CPU. Naturally, *priming* and *probing* run in a WGSL shader on the GPU. To optimize message transmission, we sum up the transmitted bits for each package within the WGSL shader and implement the differential transmission concept directly in the shader.

4.2 Evaluation

We evaluate our covert channel on three Nvidia GPUs. We did not evaluate AMD GPUs, as our sender is a CUDA application and AMD does not support CUDA.

To compute the true channel capacity C, we use Shannon's formula [151]

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

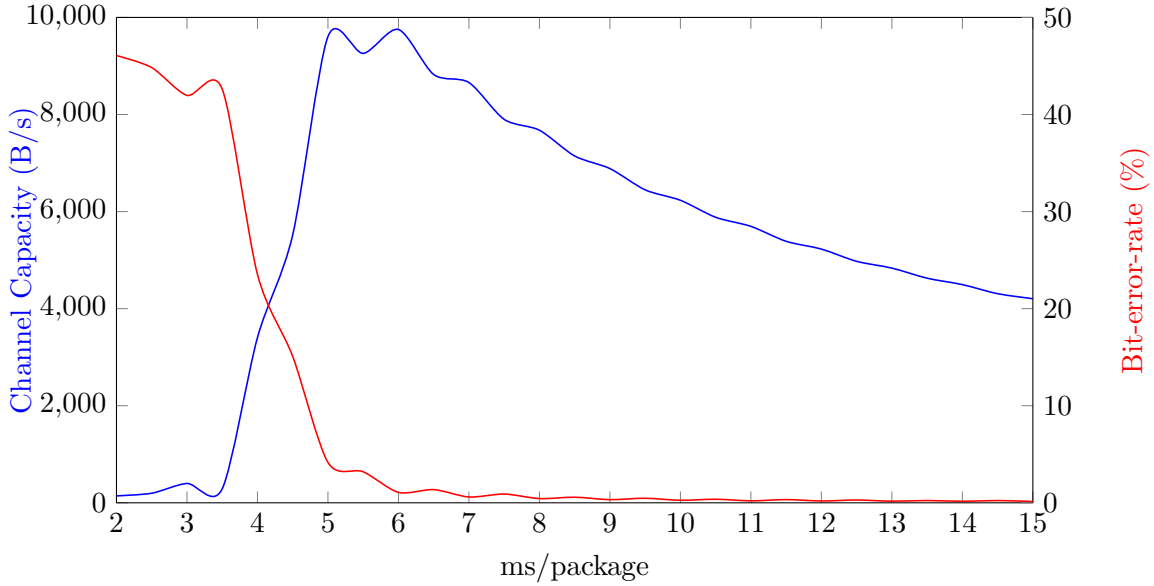


Figure 4.3: Comparison of bit-error-rate and true channel capacity (B/s) for different package window lengths (0.5 ms steps) on an Nvidia RTX 2070 Super ($n = 10$). 50 % bit-error-rate is equivalent to a random guess.

where T is the transmission rate, and p the bit-error-rate. Table 4.1 compares the transmission results for different package window lengths. The raw bandwidth value represents the maximum transmission rate for the corresponding package window length with no errors in the transmission. We compute the maximum transmission rate using the package window length and the transmitted information per package. Since we use 1024 sets for the transmission and use 2 sets per transmitted bit for differential transmission, we get $^{1024}/_{8*2}$ =64 B per package. Therefore, the maximum transmission rate is computed with

$$T = 64\,\mathrm{B} * \frac{1000\,\mathrm{ms}}{package_{ms}}.$$

We observe a large variation in true channel capacity relative to conventional benchmarks. For the measurements, we tried to reduce noise as much as possible. However, due to the GPU's involvement in updates on-screen and overhead from WebGPU shader invocation, noise on the L2 cache is inevitable on a single-GPU system. The significant noise levels observed in WebGPU often result in errors during message transmission or, even worse, during the transmission of sets.

We found that window sizes ranging from 4 to 6 milliseconds provide the best balance between bit-error-rate and maximum transmission rate on the tested Nvidia GPUs. Compared to the other two devices, the Nvidia RTX 3080 allows for a slightly faster transmission due to its increased clock rate. We observe the highest true channel capacity on the Nvidia RTX 3080 with 10.9 kB and a bit-error-rate of 2.2 %. Even though our

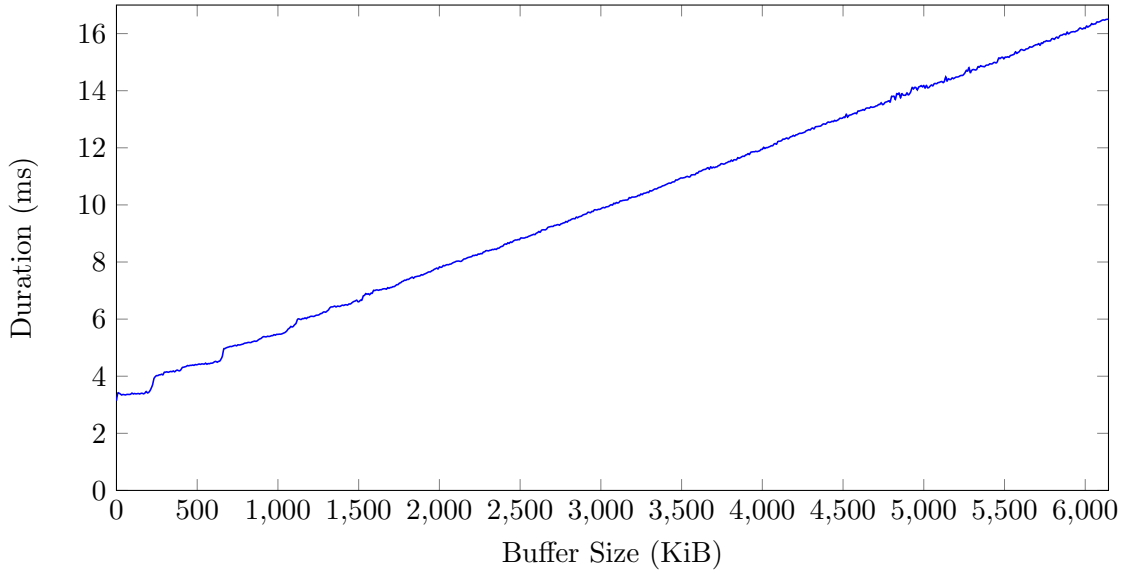


Figure 4.4: The invocation and waiting for the completion of a simple shader in WebGPU takes at least 3 ms on an Nvidia RTX 2070 Super. We observe similar values for other devices. When copying data to and from GPU buffers in addition to the invocation, the duration does not increase significantly for up to 200 KiB buffers. For larger buffers, we observe a linear increment in the overall duration.

channel is slower compared to prior work, we demonstrate that it is feasible to construct a covert channel receiver with the WebGPU API.

Figure 4.3 compares different package window lengths on an Nvidia RTX 2070 Super GPU. Window lengths below 3 ms result in bit-error-rates close to random guesses. This randomness is expected, as we invoke the receiver shader in WebGPU for every package, and the invocation alone takes around 3 ms. We observe the highest true channel capacity at around 5-6 ms for this GPU. Since the bit-error-rate is already close to 0 % for package lengths of 6 ms, increasing the package length further only reduces the true channel capacity.

Figure 4.4 shows the duration of copying data to a GPU buffer, invoking a shader, and copying back the result. The shader increments the first value in the buffer, so the buffer size does not influence the complexity and duration of the computation. For the 0 KiB buffer size, we did not copy buffers but waited for the completion of the shader. Figure 4.4 shows that the duration for invocations, including copying buffers up to around 200 KiB is very close to the duration of an invocation without buffer copies. Therefore, we can copy small buffers almost for free.

Chapter 5

Discussion and Related Work

In this chapter, we discuss the limitations and implications of our cache attack. First, we cover the supported devices and the limitations we faced during the evaluation. Then, we discuss two attacks presented in the paper by Giner et al. [1]. They use the fundamental building blocks discussed in this thesis to monitor inter-keystroke timings and recover an AES key with browser-based templating. Furthermore, we present related work and discuss how it compares to this thesis, highlighting similarities and distinctions. Finally, we discuss future work and potential mitigations against side-channel attacks in WebGPU.

5.1 Supported Devices and Limitations

In our study, we primarily focused on Nvidia devices due to limited access to AMD hardware, which resulted in worse results on AMD devices. The eviction-set-finding algorithm and the subsequent covert channel did not perform successfully on AMD devices. Nonetheless, the basic building blocks, such as the custom timer and the distinction between cache hits and cache misses, were also validated successfully on AMD devices. Therefore, we believe that more sophisticated attacks on AMD devices via WebGPU are feasible with additional focus on these systems. Currently, WebGPU support on mobile devices, particularly Android, is still in the early stages of development compared to desktop implementations. However, as WebGPU continues to develop, we anticipate potential browser-based attacks on mobile devices' GPUs as well.

Our experiments demonstrated that WebGPU enables generic attacks on GPUs directly from the browser since our basic building blocks for cache attacks in WebGPU were effectively evaluated across various devices. As the WebGPU standard is still evolving, we observe inconsistencies across different versions of WebGPU. However, since our attack does not even require a timer, it is unlikely that future versions will mitigate this class of attacks without significant performance deterioration or by imposing very restrictive API access, such as removing atomic operations.

We further observe differences across operating systems. We did not investigate the root cause for these differences. Potential causes include driver, browser, or the underlying framework, such as DirectX and Vulkan.

However, our findings confirm that our basic building blocks are sufficiently generic, as the critical timing difference between cache hits and cache misses was consistently observable across different operating systems.

5.2 Work based on our Framework

In this section, we discuss attacks from the published paper by Giner et al. [1]. The examples use the building blocks described in this thesis to build sophisticated attacks. First, we cover an inter-keystroke timing attack. In the second attack, they recover a full AES encryption key used in a native CUDA application from the browser.

5.2.1 Inter-Keystroke Timing Attack

When constructing the basic building blocks for GPU side-channel attacks using WebGPU, we noticed that certain external events often lead to nearly complete cache evictions. These events include refreshes of the frame buffer and screen updates when something is drawn. While these cache evictions may be undesirable for some cache attacks, they can also serve as a side channel themselves. Giner et al. [1] take advantage of the fact that these events indicate screen activity to construct an inter-keystroke timing attack. Prior work has shown that the timings between keystrokes carry a significant amount of information that can help to recover passwords [152, 74, 153, 121]. The described attack is a practical example where a victim enters a password on a static login page, and the attack code spies from another browser tab. Since the scenario is also realistic on mobile devices, the relevance of the attack will increase even more once WebGPU is fully supported on mobile devices. While the keystroke events occur with a low frequency, a high detection rate is crucial for the accuracy of the measured inter-keystroke timings. The approach by Giner et al. [1] is similar to an attack described by Naghibijouybari et al. [16].

The attack exploits the fact that the text box is re-rendered when a character is typed. As each re-rendering causes a significant portion of the cache to be evicted, an attacker can observe this. In their attack, an attacker continuously fills a buffer and measures the cache hitrate. A keystroke event is recorded if the hitrate is below a certain threshold. The size of the buffer is an important factor for the accuracy of the timing measurements. A larger buffer takes longer to fill, resulting in a slow detection rate. However, a rendering event might not evict a too small buffer, and the corresponding keystroke event is missed. Giner et al. [1] found that a buffer size of 35% of the cache size is a good tradeoff between detection and speed and works across various devices. Screen resolution, the size of the text box, and the zoom level contribute to the number of evicted cache lines. They further filter some events to improve the performance of the attack. Filtered events include events that are separated by less than 25 ms, as keystrokes are usually separated by a larger timeframe. Furthermore, they can filter the cursor's blinking, as this event occurs at predictable intervals. For example, they measured a cursor blinking interval of 530 ms on Windows.

The attack works across multiple Nvidia GPUs, reaching F_1 scores between 0.82 and 0.98. For the evaluation, they assume no other activity on the screen to minimize visual noise. This scenario is similar to static login pages of many websites. On AMD GPUs, they observe a high recall value similar to Nvidia cards but very low precision. They consider the attack mostly failed or severely degraded on AMD. They attribute the low

precision to either high levels of noise or, more likely, frequent misclassification of hits as misses due to the close timing differences between cache hits and cache misses as observed for AMD GPUs in Table 3.1.

5.2.2 Set-Agnostic AES Key Recovery

Recovering the encryption key from a vulnerable T-table AES implementation has become a common benchmark to demonstrate fine-grained information extraction using a side-channel attack [24, 79, 154]. This benchmark is also used to assess side channels on GPUs [14, 10, 11, 12]. Furthermore, prior work has proposed AES on GPUs since 2007 [8, 9].

The attack described by Giner et al. [1] again uses basic building blocks described in this work. In contrast to prior work, their attack is set-agnostic and requires no eviction set search. Instead, they follow a similar approach to the one used for the inter-keystroke timing attack. They focus on finding offsets within a buffer congruent with T-table lines used in the AES encryption.

Their attack assumes a native AES CUDA implementation that can be queried with an attacker's chosen key and plaintext. The implementation uses 4 combined T-tables for all rounds. Each T-table consists of 256 4 B entries. Combined with the 128 B cache line size of GPUs, this results in 32 cache lines filled with T-table entries.

The attacker spies from a WebGPU application and has access to the victim's ciphertext. Like the inter-keystroke timing attack, the attacker fills a buffer that occupies a significant portion of the cache. The first phase is a profiling phase to identify offsets within the buffer coherent with T-table lines. The size of this buffer depends on the GPU. After filling the buffer, the attacker triggers an AES encryption and checks which offsets were evicted by the encryption. They use a chosen key and crafted plaintexts to find relations between buffer offsets and table entries. Due to significant noise from kernel loading, they can identify $^{20}/_{32}$ offsets coherent to T-table lines on average.

In the second phase of the attack, they execute a traditional last-round attack [155, 156], specifically the non-elimination method described by Neve and Seifert [157]. They collect accesses to T-tables and the corresponding ciphertexts for each encryption by the victim. Combining the information about T-table lines not accessed during encryption with the corresponding ciphertext allows them to remove last-round key bytes that would have caused table accesses in the last round based on the ciphertext. They repeat this until less than 2^{40} candidates remain; then, they switch to an exhaustive search of the key.

As their AES implementation is CUDA-based, evaluation on AMD GPUs is infeasible. They evaluated the attack on 2 Nvidia GPUs and recovered the full AES key in about 6 minutes with a 100% success rate.

5.3 Related Work

In this section, we discuss related work. We discuss covert and side-channel attacks on GPUs from native APIs like Cuda and OpenGL. We then discuss side-channel attacks that target the GPU from the browser.

5.3.1 Covert and Side Channels on GPUs

In their recent survey, Naghibijouybari et al. [158] provide an overview of research on microarchitectural attacks on heterogeneous systems. In many works, the attacker is located on the CPU, spying on the targetted GPU.

Jiang et al. [10] present the first cache-based timing attack on GPUs. They demonstrate a complete AES key recovery through a timing channel in under 30 minutes with one million timing samples. The attacker sends encryption requests to the victim over the internet and collects the encrypted ciphertext and timing information. Their attack targets the L1 cache on Nvidia GPUs and relies on the dependency between unique cache line requests and kernel execution time. They assume a vulnerable AES encryption, where access to specific cache lines depends on input data and the encryption key.

Naghibijouybari et al. [15] show the first covert channels on applications co-located on the same GPU. They reverse-engineered the hardware block scheduler and the warp-to-warp scheduler. With the information about the scheduler, they manipulate the scheduling algorithm to establish co-location between the spy and trojan. They exploit contention on shared resources, like L1 and L2 caches, functional units, and memory, to construct multiple covert channels. They use the GPU's parallelism to increase the channel's bandwidth.

Jiang et al. [11, 12] exploit bank conflicts from shared memory. On GPUs, every thread in a warp can access the shared memory, which serves as a cache to decrease memory access latency. They target an AES implementation that uses key-dependent memory accesses for table lookups. They create bank conflicts and measure the execution time of the encryption. As the table lookups depend on the key, they can infer the correct key by observing timing differences.

Naghibijouybari et al. [16] present the first general side-channel attack on GPUs. They present two attacks that spy on victim applications co-located on the GPU. They exploit the fine scheduling granularity to interleave the execution of their attacker code with the victim application. The first attack uses the graphical software stack to fingerprint websites, track user activities within websites, and infer keystroke timings for a password text box. They track re-rendering events and use the GPU's performance counters for memory usage accessible from OpenGL for the attack. For the second attack, they use the compute software stack. They use a CUDA spy application to derive internal parameters of a neural network model used by another CUDA application.

Liu et al. [159] demonstrate the feasibility of GPU side channels across virtual machines. Both virtual machines share one integrated Intel GPU. They construct an OpenCL-based probing application to generate contention on shared resources. By measuring the

execution time, they can identify the victim's workload from a small set of entertainment and deep learning workloads.

Wei et al. [6] extract more fine-grained structural secrets of a neural network model than Naghibijouybari et al. [16]. They assume a co-location of an attacker and a model developer training a DNN (Deep Neural Network) in a cloud scenario with two virtual machines. For the attack, they exploit a side channel based on context-switch penalties. They use Nvidia's performance counters for their attack. After Nvidia's driver update to restrict access to the performance counter API [160], they downgrade the driver on the attacker's virtual machine. Since only the attacker needs access to the timer, Nvidia's driver update does not mitigate this attack.

Ahn et al. [14] present a hybrid cache-collision timing attack on modern GPUs. They use a combination of cache collisions, chosen plaintext attack, and negative timing correlation to significantly reduce the number of samples required for a full AES key recovery.

Dutta et al. [13] present the first microarchitectural attack crossing the component boundaries between GPU and CPU on integrated GPU devices. They construct two covert channels. First, they use contention on Intel's LLC, which is shared between the CPU and the integrated GPU, to construct a Prime+Probe cache covert channel. The second covert channel exploits contention on the ring bus connecting the CPU and GPU. Furthermore, they use the GPU's parallelism to increase the communication bandwidth. Their proof-of-concept probes the entire LLC from the GPU.

Nayak et al. [145] show the first TLB-based covert channel on GPUs. They reverse-engineered the GPU's TLB to perform fine-grained management of translations and construct a Prime+Probe covert channel.

On-chip interconnects are channels between streaming multiprocessors. Ahn et al. [17] exploit contention on the shared on-chip interconnect channels to construct a covert channel. At the time of publication, their covert channel had one of the highest bandwidths observed on Nvidia Volta GPUs, reaching up to 24 Mbit/s.

Dutta et al. [110] demonstrate the first microarchitectural attacks on connected Nvidia GPUs. The GPUs are connected with Nvidia's custom interconnect, NVLink [161]. They reverse-engineered the interconnected cache and cause contention on the L2 cache of another GPU. They use this contention to develop a Prime+Probe-based covert channel between GPUs. Furthermore, they use the contention as a side channel to spy on the victim's behavior.

The attacks mentioned above rely on access to native APIs. Native APIs like CUDA and OpenGL offer access to high-precision performance counters and timers. However, when constructing a covert channel in WebGPU, we lack access to these high-performance counters and timers, which complicates the execution of the attack.

Several prior works [10, 14, 11, 13] do not assume co-location of attacker and victim on the same GPU. They use shared resources between CPU and GPU on systems with an integrated GPU to leak data or monitor the GPU's resource utilization using an application executing on the CPU.

5.3.2 Side-Channel Attacks on GPUs from the Browser

API access to the GPU from the browser enables attackers to mount attacks through JavaScript. To our knowledge, all known browser-based attacks exploit the GPU using the WebGL API. Our focus on WebGPU introduces distinct challenges, differing significantly from previous research that used WebGL.

Cronin et al. [19] present a browser-based GPU side-channel attack on ARM processors. Their cross-tab attack code runs in a background tab on an ARM device and performs a website fingerprinting attack. They use GPU.js [137] to transpile JavaScript functions to WebGL shader code for their attack code. Their attack kernel continuously computes on values in a buffer to cause contention on the system-level cache shared between SoC and GPU. They measure the duration of the computations and exploit the correlation between the duration and the memory used by each website to perform a website fingerprinting attack. Their work stands apart from ours by focusing on the system-level cache of integrated GPUs and utilizing WebGL instead of WebGPU. Additionally, their attacks are more coarse-grained.

Frigo et al. [18] construct all required primitives for GPU-based microarchitectural attacks on GPUs using the WebGL API in JavaScript. They construct a precise timer and use reverse-engineering techniques to gain information about the cache architecture and replacement policy of an integrated GPU. They use the timer and the gathered information for a side-channel attack that detects contiguous areas of physical memory from the GPU. They further develop the first web-based GPU-accelerated Rowhammer attack from a browser on mobile devices.

Taneja et al. [162] use native APIs to monitor the GPU's power consumption, temperature, and frequency to construct hybrid side channels. They exploit the instruction and data-dependent behavior of GPUs for history sniffing and website fingerprinting. One of their attacks involves a pixel-stealing technique from the browser that does not require native APIs. They construct a specially designed SVG filter that triggers unique frequency throttling behaviors based on the hamming weight of black and white pixels. By exploiting the data-dependent execution times of this filter, they create a side-channel attack capable of extracting pixel data from the victim. Their attack does not directly use any browser GPU API like WebGL or WebGPU and is more coarse-grained than ours. They exploit the data-dependent execution time of instructions on the GPU and do not directly target microarchitectural buffers like caches.

5.4 Future Work

As general-purpose computing on GPUs becomes more accessible and computations on sensitive data more common, research on possible vulnerabilities on GPUs becomes even more critical. Even though research on microarchitectural attacks on GPUs is less evolved than their counterpart on CPUs, prior work has shown that several attack techniques also apply to GPUs. In general, less is known about the microarchitectural details of GPUs compared to CPUs. As prior works reverse-engineered several microarchitectural

details of specific GPUs, they found that various aspects differ not only between vendors but also among devices from different generations of the same vendor [112, 163, 64]. The feasibility of applying CPU attack principles on shared resources on GPUs [145] hints at more possible microarchitectural attacks on GPUs. Therefore, more research must be conducted to reverse-engineer microarchitectural components within GPUs.

As the shared memory model of GPUs is fundamentally different from that of CPUs and sharing data between applications is not common practice, we do not expect to see Flush+Reload-based attacks on GPUs. Even though GPUs offer no instruction to flush instructions from specific cache lines, attacks on cache line granularity cannot be ruled out, as research on CPUs has shown [74].

5.5 Mitigations

The WebGPU standard incorporates several mitigations against side-channel attacks, such as using an imprecise timer [22]. Despite these measures, we constructed an effective timer by utilizing coherent memory shared between concurrent threads, facilitated by atomic operations. Removing atomic operations would invalidate our timer; however, they are crucial for numerous common workloads due to their role in synchronization and other critical functions.

We have demonstrated that side-channel attacks are feasible in WebGPU using basic assumptions. These assumptions are essential to how current software is built, and removing them would have significant consequences. Therefore, we propose treating access to the GPU in the browser like access to other privacy- and security-relevant resources such as camera and microphone. This measure could help prevent GPU side-channel attacks and stop the covert misuse of GPU computing resources for activities such as cryptomining.

Chapter 6

Conclusion

In this thesis, we developed the first side-channel attack from the browser using the WebGPU API. Our attack is generic and self-configuring and requires no user interaction. Since the WebGPU API was designed with mitigations against side-channel attacks in mind, we had to overcome several challenges, like the absence of timers.

We implemented a counting thread that serves as a timer for other GPU threads. We use this timer to automatically determine a baseline threshold value to distinguish cache hits and cache misses. With these two building blocks, we can automatically detect the cache size and, subsequently, the number of cache sets of the underlying GPU hardware. These building blocks and parameters are essential to mount a Prime+Probe cache attack and were successfully tested on 12 GPUs from 5 generations and 2 vendors.

To demonstrate the susceptibility of WebGPU to cache attacks, we used the building blocks to construct a Prime+Probe cache covert channel. The covert channel communicates data from a native CUDA application to an app in the browser and reaches a true channel capacity of 10.9 kB/s. For a Prime+Probe attack, we had to find eviction sets in the browser. We constructed the first parallel eviction-set-finding algorithm in the browser. The algorithm finds more than 80% of eviction sets on all but one tested Nvidia GPU in 2 to 12 minutes. Our attack requires no user interaction and runs on a variety of GPUs.

We demonstrated that access to general-purpose computing on the GPU from the browser can be a powerful tool for attackers. We have shown that it is possible to carry out remote, generic, and fully automated attacks on GPU caches from a restricted browser environment. As users regularly run untrusted third-party applications in browsers, it is easier for an attacker to execute WebGPU code on the victim's machine than to obtain native code execution. Therefore, we recommend that access to the GPU is treated like other security- and privacy-related resources, such as the microphone and camera. Furthermore, we urgently need to deepen our understanding of GPU vulnerabilities to counter future side-channel attacks preemptively.

Bibliography

[1] Lukas Giner, Roland Czerny, Christoph Gruber, Fabian Rauscher, Andreas Kogler, Daniel De Almeida Braga, and Daniel Gruss. "Generic and Automated Drive-by GPU Cache Attacks from the Browser". In: *AsiaCCS*. 2024.

[2] Chris J. Thompson, Sahngyun Hahn, and Mark Oskin. "Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis". In: *MICRO*. 2002.

[3] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques For High-Performance Graphics And General-Purpose Computation*. Addison-Wesley Professional, 2005.

[4] Nvidia. *CUDA Zone*. 2024. URL: `https://developer.nvidia.com/cuda-zone`.

[5] The Khronos Group. *OpenCL*. 2024. URL: `https://www.khronos.org/opencl/`.

[6] Junyi Wei, Yicheng Zhang, Zhe Zhou, Zhou Li, and Mohammad Abdullah Al Faruque. "Leaky DNN: Stealing Deep-Learning Model Secret with GPU Context-Switching Side-Channel". In: *DSN*. 2020.

[7] Nvidia. *GPU Cloud Computing*. 2024. URL: `https://www.nvidia.com/en-us/data-center/gpu-cloud-computing/`.

[8] Cihangir Tezcan. "Optimization of Advanced Encryption Standard on Graphics Processing Units". In: *IEEE Access* 9 (2021), pp. 67315–67326.

[9] Takeshi Yamanouchi. *GPU Gems 3 - AES Encryption and Decryption on the GPU*. 2007. URL: `https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-36-aes-encryption-and-decryption-gpu`.

[10] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. "A Complete Key Recovery Timing Attack on a GPU". In: *HPCA*. 2016.

[11] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. "A Novel Side-Channel Timing Attack on GPUs". In: *GLSVLSI*. 2017.

[12] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. "Exploiting Bank Conflict-based Side-channel Timing Leakage of GPUs". In: *ACM TACO*. 2019.

[13] Sankha Baran Dutta, Hoda Naghibijouybari, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. "Leaky Buddies: Cross-Component Covert Channels on Integrated CPU-GPU Systems". In: *ISCA*. 2021.

[14] Jaeguk Ahn, Cheolgyu Jin, Jiho Kim, Minsoo Rhu, Yunsi Fei, David Kaeli, and John Kim. "Trident: A Hybrid Correlation-Collision GPU Cache Timing Attack for AES Key Recovery". In: *HPCA*. 2021.

[15] Hoda Naghibijouybari, Khaled N Khasawneh, and Nael Abu-Ghazaleh. "Constructing and Characterizing Covert Channels on GPGPUs". In: *MICRO*. 2017.

[16] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. "Rendered Insecure: GPU Side Channel Attacks are Practical". In: *CCS*. 2018.

[17] Jaeguk Ahn, Jiho Kim, Hans Kasan, Leila Delshadtehrani, Wonjun Song, Ajay Joshi, and John Kim. "Network-on-Chip Microarchitecture-based Covert Channel in GPUs". In: *MICRO*. 2021.

[18] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. "Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU". In: *S&P*. 2018.

[19] Patrick Cronin, Xing Gao, Haining Wang, and Chase Cotton. "An Exploration of ARM System-Level Cache and GPU Side Channels". In: *ACSAC*. 2021.

[20] W3C. *WebGPU Shading Language*. 2024. URL: `https://www.w3.org/TR/webgpu/`.

[21] Mozilla. *WebGPU API*. 2024. URL: `https://developer.mozilla.org/en-US/docs/Web/API/WebGPU_API`.

[22] W3C. *WebGPU Shading Language - Timing Attacks*. 2024. URL: `https://www.w3.org/TR/webgpu/#security-timing`.

[23] Moinuddin K. Qureshi. "New Attacks and Defense for Encrypted-Address Cache". In: *ISCA*. 2019.

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

[25] Colin Percival. *Cache missing for fun and profit*. 2005. URL: `https://www.daemonology.net/papers/htt.pdf`.

[26] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud." In: *NDSS*. 2017.

[27] Chris McClanahan. *History and Evolution of GPU Architecture*. 2010. URL: `https://picture.iczhiku.com/resource/paper/WyiWoZOZoDZWgvxN.pdf`.

[28] Thomas Scott Crow. *Evolution of the Graphical Processing Unit*. 2004. URL: `https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=8f68c39e3925275e1d5a881619fb37944b0c4e31`.

[29] Ian Buck. *The Evolution of GPUs for General Purpose Computing*. 2010. URL: `https://www.nvidia.com/content/gtc-2010/pdfs/2275_gtc2010.pdf`.

[30] Martin Rumpf and Robert Strzodka. "Using Graphics Cards for Quantized FEM Computations". In: *VIIP*. 2001.

[31] Timothy J Purcell, Ian Buck, William R Mark, and Pat Hanrahan. "Ray Tracing on Programmable Graphics Hardware". In: *SIGGRAPH*. 2005.

[32] E Scott Larsen and David McAllister. "Fast Matrix Multiplies using Graphics Hardware". In: *SC*. 2001.

[33] David Luebke, Mark Harris, Naga Govindaraju, Aaron Lefohn, Mike Houston, John Owens, Mark Segal, Matthew Papakipos, and Ian Buck. "GPGPU: General-Purpose Computation on Graphics Hardware". In: *SC*. 2006.

[34] Mark Harris. "Mapping Computational Concepts to GPUs". In: *SIGGRAPH*. 2005.

[35] Nvidia. *NVIDIA GeForce 8800 Architecture Technical Brief*. 2006. URL: https://www.nvidia.co.uk/content/PDF/Geforce_8800/GeForce_8800_GPU_Architecture_Technical_Brief.pdf.

[36] Jon Peddie. *AMD's Unified Shader GPU History*. 2023. URL: https://www.computer.org/publications/tech-news/chasing-pixels/amd-unified-shader-gpu-history.

[37] Nvidia. *Nvidia Ampere GA102 GPU Architecture*. 2020. URL: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.1.pdf.

[38] Harris Mark Durant Luke Giroux Olivier and Stam Nick. *Inside Volta: The World's Most Advanced Data Center GPU*. 2017. URL: https://developer.nvidia.com/blog/inside-volta/.

[39] Nvidia. *Nvidia Tesla V100 GPU Architecture*. 2017. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[40] Nvidia. *Pascal MMU Format*. 2016. URL: https://nvidia.github.io/open-gpu-doc/pascal/gp100-mmu-format.pdf.

[41] Nvidia. *Kernel Profiling Guide*. 2024. URL: https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#l2-cache.

[42] AMD. *RDNA Architecture*. 2019. URL: https://www.amd.com/system/files/documents/rdna-whitepaper.pdf.

[43] Naznin Fauzia, Louis-Noël Pouchet, and P Sadayappan. "Characterizing and Enhancing Global Memory Data Coalescing on GPUs". In: *CGO*. 2015.

[44] Bingchao Li, Jizeng Wei, Jizhou Sun, Murali Annavaram, and Nam Sung Kim. "An Efficient GPU Cache Architecture for Applications with Irregular Memory Access Patterns". In: *ACM TACO*. 2019.

[45] Laszlo A. Belady. "A study of replacement algorithms for a virtual-storage computer". In: *IBM Systems journal* 5.2 (1966), pp. 78–101.

[46] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. "Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite". In: *SERC*. 2004.

[47] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C Steely Jr, and Joel Emer. "SHiP: Signature-based Hit Predictor for High Performance Caching". In: *MICRO*. 2011.

[48] Akanksha Jain and Calvin Lin. "Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement". In: *ACM SIGARCH Computer Architecture News* 44.3 (2016), pp. 78–89.

[49] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. "Applying Deep Learning to the Cache Replacement Problem". In: *MICRO*. 2019.

[50] Subhash Sethumurugan, Jieming Yin, and John Sartori. "Designing a Cost-Effective Cache Replacement Policy using Machine Learning". In: *HPCA*. 2021.

[51] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript". In: *DIMVA*. 2016.

[52] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. "Adaptive Insertion Policies for High Performance Caching". In: *ACM SIGARCH Computer Architecture News* 35.2 (2007), pp. 381–391.

[53] Sun Microsystems, Inc. *OpenSPARC T2™ Supplement*. 2007. URL: `https://www.oracle.com/technetwork/systems/opensparc/t2-14-ust2-uasuppl-draft-hp-ext-1537761.html`.

[54] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. "RELOAD + REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks". In: *USENIX Security*. 2020.

[55] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 2011.

[56] Fatemeh Kazemi Hassan Abadi and Saeed Safari. "Performance and area aware replacement policy for GPU architecture". In: *ICCKE*. 2014.

[57] Amr Abdelhai Mourad, Saleh Mesbah, and Tamer F. Mabrouk. "A Novel Approach to Cache Replacement Policy Model Based on Genetic Algorithms". In: *WorldS4*. 2020.

[58] Michel Cekleov and Michel Dubois. "Virtual-Address Caches. Part 1: Problems and Solutions in Uniprocessors". In: *IEEE Micro* 17.5 (1997), pp. 64–71.

[59] Alan Jay Smith. "Cache Memories". In: *ACM Computing Surveys (CSUR)* 14.3 (1982), pp. 473–530.

[60] Burton H Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors". In: *Communications of the ACM* 13.7 (1970), pp. 422–426.

[61] Dong Hyuk Woo, Mrinmoy Ghosh, Emre Özer, Stuart Biles, and Hsien-Hsin S Lee. "Reducing Energy of Virtual Cache Synonym Lookup using Bloom Filters". In: *CASES*. 2006.

[62] George Taylor, Peter Davies, and Michael Farmwald. "The TLB slice—A Low-Cost High-Speed Address Translation Mechanism". In: *ISCA*. 1990.

[63] Hamdy Abdelkhalik, Yehia Arafa, Nandakishore Santhi, and Abdel-Hameed A Badawy. "Demystifying the Nvidia Ampere Architecture through Microbenchmarking and Instruction-level Analysis". In: *HPEC*. 2022.

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

[65] Michael Andersch, Jan Lucas, Mauricio A LvLvarez-Mesa, and Ben Juurlink. "On Latency in GPU Throughput Microarchitectures". In: *ISPASS*. 2015.

[66] Joel Hestness, Stephen W Keckler, and David A Wood. "A Comparative Analysis of Microarchitecture Effects on CPU and GPU Memory System Behavior". In: *IISWC*. 2014.

[67] Xiaolong Xie, Wei Tan, Liana L Fong, and Yun Liang. "CuMF_SGD: Fast and Scalable Matrix Factorization". In: *arXiv preprint arXiv:1610.05838*. 2016.

[68] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. "A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware". In: *Journal of Cryptographic Engineering* 8 (2018), pp. 1–27.

[69] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. "A Survey of Microarchitectural Side-channel Vulnerabilities, Attacks, and Defenses in Cryptography". In: *ACM Computing Surveys (CSUR)* 54.6 (2021), pp. 1–37.

[70] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. "Predicting Secret Keys via Branch Prediction". In: *CT-RSA*. 2007.

[71] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. "BranchScope: A New Side-Channel Attack on Directional Branch Predictor". In: *ACM SIGPLAN Notices* 53.2 (2018), pp. 693–707.

[72] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks". In: *USENIX Security*. 2018.

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

[74] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches". In: *USENIX Security*. 2015.

[75] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. "Cross-Tenant Side-Channel Attacks in PaaS Clouds". In: *CCS*. 2014.

[76] Zhenghong Wang and Ruby B Lee. "Covert and Side Channels Due to Processor Architecture". In: *ACSAC*. 2006.

[77] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. "Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds". In: *CCS*. 2009.

[78] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. "An Exploration of L2 Cache Covert Channels in Virtualized Environments". In: *CCSW*. 2011.

[79] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. "ARMageddon: Cache Attacks on Mobile Devices". In: *USENIX Security*. 2016.

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

[81] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications". In: *CCS*. 2015.

[82] Zhenyu Wu, Zhang Xu, and Haining Wang. "Whispers in the Hyper-Space: High-Bandwidth and Reliable Covert Channel Attacks Inside the Cloud". In: *IEEE/ACM Transactions on Networking* 23.2 (2015), pp. 603–615.

[83] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. "Cross-VM Side Channels and Their Use to Extract Private Keys". In: *CCS*. 2012.

[84] Eran Tromer, Dag Arne Osvik, and Adi Shamir. "Efficient Cache Attacks on AES, and Countermeasures". In: *Journal of Cryptology* 23 (2010), pp. 37–71.

[85] Yuval Yarom, Daniel Genkin, and Nadia Heninger. "CacheBleed: a timing attack on OpenSSL constant-time RSA". In: *Journal of Cryptographic Engineering* 7 (2017), pp. 99–112.

[86] 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.

[87] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. "ASLR on the Line: Practical Cache Attacks on the MMU." In: *NDSS*. 2017.

[88] Alex Christensen. *WebKit Bugzilla - Reduce resolution of performance.now*. 2015. URL: `https://bugs.webkit.org/show_bug.cgi?id=146531`.

[89] Kunihiko Sakamoto. *Reduce resolution of performance.now to prevent timing attacks*. 2015. URL: `https://chromium.googlesource.com/chromium/blink/+/34c7884f58ae24b174179e1e9aec29c96e3c8d21`.

[90] Boris Zbarsky. *Clamp the resolution of performance.now() calls to 5us, because otherwise we allow various timing attacks that depend on high accuracy timers*. 2015. URL: `https://hg.mozilla.org/integration/mozilla-inbound/rev/48ae8b5e62ab`.

[91] David Kohlbrenner and Hovav Shacham. "Trusted Browsers for Uncertain Times". In: *USENIX Security*. 2016.

[92] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript". In: *FC*. 2017.

[93] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. "The Clock is Still Ticking: Timing Attacks in the Modern Web". In: *CCS*. 2015.

[94] The Chromium Projects. *Mitigating Side-Channel Attacks*. 2018. URL: `https://www.chromium.org/Home/chromium-security/ssca/`.

[95] Filip Pizlo. *What Spectre and Meltdown Mean For WebKit*. 2018. URL: `https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/`.

[96] Microsoft Edge Team. *Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer*. 2018. URL: `https://blogs.windows.com/msedgedev/2018/01/03/speculative-execution-mitigations-microsoft-edge-internet-explorer/#b8Y70MtqGTVR7mSC.97`.

[97] Wagner Luke. *Mitigations landing for new class of timing attack*. 2018. URL: `https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/`.

[98] Yinzhi Cao, Zhanhao Chen, Song Li, and Shujiang Wu. "Deterministic Browser". In: *CCS*. 2017.

[99] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. "Robust Website Fingerprinting Through the Cache Occupancy Channel". In: *USENIX Security*. 2019.

[100] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *USENIX Security*. 2014.

[101] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing—and its Application to AES". In: *S&P*. 2015.

[102] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. "C5: Cross-Cores Cache Covert Channel". In: *DIMVA*. 2015.

[103] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks". In: *USENIX Security*. 2016.

[104] Sarani Bhattacharya and Debdeep Mukhopadhyay. "Curious case of Rowhammer: Flipping Secret Exponent Bits using Timing Analysis". In: *CHES*. 2016.

[105] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. "Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR". In: *MICRO*. 2016.

[106] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "Systematic Reverse Engineering of Cache Slice Selection in Intel Processors". In: *DSD*. 2015.

[107] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B Lee, and Gernot Heiser. "Mapping the Intel Last-Level Cache". In: *IACR Cryptology ePrint Archive, Report 2015/905* (2015).

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

[109] Kirill A. Shutemov. *pagemap: do not leak physical addresses to non-privileged userspace*. 2015. URL: `https://lwn.net/Articles/642074/`.

[110] 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: *ISCA*. 2023.

[111] Saksham Jain, Iljoo Baek, Shige Wang, and Ragunathan Rajkumar. "Fractional GPUs: Software-Based Compute and Memory Bandwidth Reservation for GPUs". In: *RTAS*. 2019.

[112] Xinxin Mei and Xiaowen Chu. "Dissecting GPU Memory Hierarchy Through Microbenchmarking". In: *IEEE Transactions on Parallel and Distributed Systems* 28.1 (2016), pp. 72–86.

[113] Pepe Vila, Boris Köpf, and José F Morales. "Theory and Practice of Finding Eviction Sets". In: *S&P*. 2019.

[114] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. "Attack Directories, Not Caches: Side-Channel Attacks in a Non-Inclusive World". In: *S&P*. 2019.

[115] Haehyun Cho, Penghui Zhang, Donguk Kim, Jinbum Park, Choong-Hoon Lee, Ziming Zhao, Adam Doupé, and Gail-Joon Ahn. "Prime+Count: Novel Cross-world Covert Channels on ARM TrustZone". In: *ACSAC*. 2018.

[116] David Gullasch, Endre Bangerter, and Stephan Krenn. "Cache Games – Bringing Access-Based Cache Attacks on AES to Practice". In: *S&P*. 2011.

[117] Berk Gülmezoğlu, Mehmet Sinan Inci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "A Faster and More Realistic Flush+Reload Attack on AES". In: *COSADE*. 2015.

[118] Yuval Yarom and Naomi Benger. "Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack". In: *IACR Cryptology ePrint Archive* (2014).

[119] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. "Flush, Gauss, and Reload - A Cache Attack on the BLISS Lattice-Based Signature Scheme". In: *CHES*. 2016.

[120] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. "Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices". In: *CCS*. 2016.

[121] John V Monaco. "SoK: Keylogging Side Channels". In: *S&P*. 2018.

[122] Daimeng Wang, Ajaya Neupane, Zhiyun Qian, Nael B Abu-Ghazaleh, Srikanth V Krishnamurthy, Edward JM Colbert, and L Yu Paul. "Unveiling your keystrokes: A Cache-based Side-channel Attack on Graphics Libraries". In: *NDSS*. 2019.

[123] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown". In: *arXiv preprint arXiv:1801.01207*. 2018.

[124] 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.

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

[126] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. "ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks". In: *ACM SIGPLAN Notices* 51.4 (2016), pp. 743–755.

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

[128] The Khronos Group. *Vulkan*. 2024. URL: https://www.vulkan.org/.

[129] Apple. *Accelerate graphics and much more with Metal*. 2024. URL: https://developer.apple.com/metal/.

[130] Microsoft. *Direct3D*. 2024. URL: https://learn.microsoft.com/en-us/windows/win32/direct3d.

[131] The Khronos Group. *WebGL 2.0 Compute*. 2021. URL: https://registry.khronos.org/webgl/specs/latest/2.0-compute/.

[132] The Khronos Group. *OpenGL*. 2024. URL: https://www.opengl.org/.

[133] Hassan Mujtaba. *GPU Market Rebounds In Q2 2023: AMD, NVIDIA & Intel See Increased Shipments, Discrete GPU Up By 12.4%*. 2023. URL: https://wccftech.com/gpu-market-rebounds-q2-2023-amd-nvidia-intel-increased-shipments-discrete-gpus-up/.

[134] TT Consultants. *Nvidia's Market Value Soars to 11x Intel's on Less Than Half the Sales*. 2024. URL: https://ttconsultants.com/nvidias-market-value-soars-to-11x-intels-on-less-than-half-the-sales/.

[135] 1kg. *Nvidia's CUDA Monopoly*. 2023. URL: https://medium.com/%5C@1kg/nvidias-cuda-monopoly-6446f4ef7375.

[136] Nvidia. *Compute Capabilities*. 2024. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capability.

[137] gpujs. *GPU.js*. 2021. URL: https://github.com/gpujs/gpu.js.

[138] The Khronos Group. *OpenGL ES Overview*. 2024. URL: https://www.khronos.org/opengles/.

[139] Gregg Tavares. *WebGL2 GPGPU*. 2021. URL: https://webgl2fundamentals.org/webgl/lessons/webgl-gpgpu.html.

[140] Yang Gu. *Unlock the Potential of AI and Immersive Web Applications with WebGPU*. 2023. URL: https://medium.com/intel-tech/unlock-the-potential-of-ai-and-immersive-web-applications-with-webgpu-4a1cff079178.

[141] Corentin Wallez, Brandon Jones, and François Beaufort. *WebGPU: Unlocking modern GPU access in the browser*. 2023. URL: https://developer.chrome.com/blog/webgpu-io2023.

[142] Google. *Dawn, a WebGPU implementation*. 2024. URL: https://dawn.googlesource.com/dawn.

[143] Rust Graphics Mages. *wgpu*. 2024. URL: https://github.com/gfx-rs/wgpu.

[144] Surma. *WebGPU - All of the cores, none of the canvas*. 2022. URL: https://surma.dev/things/webgpu/index.html.

[145] Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. "(Mis)managed: A Novel TLB-based Covert Channel on GPUs". In: *AsiaCCS*. 2021.

[146] W3C. *WebGPU*. 2024. URL: https://www.w3.org/TR/WGSL/#uniformity-concepts.

[147] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Malware Guard Extension: Using SGX to Conceal Cache Attacks". In: *DIMVA*. 2017.

[148] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. "Systematic Analysis of Randomization-based Protected Cache Architectures". In: *S&P*. 2021.

[149] Rafael Hector Saavedra-Barrera. *CPU performance evaluation and execution time prediction using narrow spectrum benchmarking*. University of California, Berkeley, 1992.

[150] Rafael H Saavedra and Alan Jay Smith. "Measuring Cache and TLB Performance and Their Effect on Benchmark Run Times". In: *IEEE Transactions on Computers* 44.10 (1995), pp. 1223–1235.

[151] Claude Elwood Shannon. "A Mathematical Theory of Communication". In: *The Bell System Technical Journal* 27.3 (1948), pp. 379–423.

[152] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. "Timing Analysis of Keystrokes and Timing Attacks on SSH". In: *USENIX Security*. 2001.

[153] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. "Practical Keystroke Timing Attacks in Sandboxed JavaScript". In: *ESORICS*. 2017.

[154] Daniel J Bernstein. *Cache-timing attacks on AES*. 2004. URL: http://cr.yp.to/papers.html#cachetiming.

[155] Joseph Bonneau and Ilya Mironov. "Cache-Collision Timing Attacks Against AES". In: *CHES*. 2006.

[156] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. "Wait a Minute! A fast, Cross-VM Attack on AES". In: *RAID*. 2014.

[157] Michael Neve and Jean-Pierre Seifert. "Advances on Access-Driven Cache Attacks on AES". In: *SAC*. 2006.

[158] Hoda Naghibijouybari, Esmaeil Mohammadian Koruyeh, and Nael Abu-Ghazaleh. "Microarchitectural Attacks in Heterogeneous Systems: A Survey". In: *ACM Computing Surveys (CSUR)* 55.7 (2022), pp. 1–40.

[159] Sihang Liu, Yizhou Wei, Jianfeng Chi, Faysal Hossain Shezan, and Yuan Tian. "Side Channel Attacks in Computation Offloading Systems with GPU Virtualization". In: *SPW*. 2019.

[160] Nvidia. *Security Bulletin: NVIDIA GPU Display Driver - February 2019*. 2019. URL: `https://nvidia.custhelp.com/app/answers/detail/a_id/4772`.

[161] Nvidia. *NVLink and NVLink Switch*. 2024. URL: `https://www.nvidia.com/en-us/data-center/nvlink`.

[162] Hritvik Taneja, Jason Kim, Jie Jeff Xu, Stephan Van Schaik, Daniel Genkin, and Yuval Yarom. "Hot Pixels: Frequency, Power, and Temperature Attacks on GPUs and Arm SoCs". In: *USENIX Security*. 2023.

[163] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. "Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking". In: *arXiv preprint arXiv:1804.06826*. 2018.