Generic and Automated Drive-by GPU Cache Attacks from the Browser

Lukas Giner
Graz University of Technology

Roland Czerny
Graz University of Technology

Christoph Gruber
Graz University of Technology

Fabian Rauscher
Graz University of Technology

Andreas Kogler
Graz University of Technology

Daniel De Almeida Braga
University of Rennes, CNRS, IRISA

Daniel Gruss
Graz University of Technology

ABSTRACT

In recent years, the use of GPUs for general-purpose computations has steadily increased. As security-critical computations like AES are becoming more common on GPUs, the scrutiny must also increase. At the same time, new technologies like WebGPU put easy access to compute shaders in every web browser. Prior work has shown that GPU caches are vulnerable to the same eviction-based attacks as CPUs, e.g., Prime+Probe, from native code.

In this paper, we present the first GPU cache side-channel attack from within the browser, more specifically from the restricted WebGPU environment. The foundation for our generic and automated attacks are self-configuring primitives applicable to a wide variety of devices, which we demonstrate on a set of 11 desktop GPUs from 5 different generations and 2 vendors. We leverage features of the new WebGPU standard to create shaders that implement all building blocks needed for cache side-channel attacks, such as techniques to distinguish L2 cache hits from misses. Beyond the state of the art, we leverage the massive parallelism of modern GPUs to design the first parallelized eviction set construction algorithm. Based on our attack primitives, we present three case studies: First, we present an inter-keystroke timing attack with high F_1-scores, *i.e.*, 82 % to 98 % on NVIDIA. Second, we demonstrate a generic, set-agnostic, end-to-end attack on a GPU-based AES encryption service, leaking a full AES key in 6 minutes. Third, we evaluate a native-to-browser data-exfiltration scenario with a Prime+Probe covert channel that achieves transmission rates of up to 10.9 kB/s. Our attacks require no user interaction and work in a time frame that easily enables drive-by attacks while browsing the Internet. Our work emphasizes that browser vendors need to treat access to the GPU similar to other security- and privacy-related resources.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; *Browser security*; *Information flow control.*

KEYWORDS

Side Channels, Cache Attacks, GPU computing

ACM Reference Format:
Lukas Giner, Roland Czerny, Christoph Gruber, Fabian Rauscher, Andreas Kogler, Daniel De Almeida Braga, and Daniel Gruss. 2024. Generic and Automated Drive-by GPU Cache Attacks from the Browser. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '24), July 1–5, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3634737.3656283

1 INTRODUCTION

In the last decades, Graphics Processing Units (GPUs) have seen an important evolution. While they were initially designed for the specific purpose of graphic rendering, most modern discrete GPUs offer the possibility of general-purpose computing. With the introduction of NVIDIA's CUDA [1] in 2007 and OpenCL [2] in 2009, GPUs have become commonplace for workloads that benefit from the massive parallelism they can offer. While the individual execution speed is still slow compared to recent CPUs, current-generation cards offer thousands of cores, enabling a huge performance boost for parallelizable operations.

The increasing number of use cases of general-purpose GPU computing includes computations on potentially secret information, e.g., neural networks [3] or cryptographic applications [4, 5]. Thus, general-purpose GPU computing also becomes a more interesting attack target. Recent research confirms these security concerns, as GPUs have become a recurrent target of side-channel attacks, exploiting various shared components [3, 6–12]. Furthermore, attackers may also leverage the GPU to attack other system components [13, 14]. As on CPUs, the GPU cache is a particularly interesting resource for side channels. Consequently, prior work also replicated well-known CPU cache side-channel attacks on GPUs [6, 10, 12, 15, 16], albeit only in native code so far.

While native code has direct access to a large variety of GPU APIs, e.g., CUDA, Vulkan, Metal, and Direct3D, acquiring native code execution is a significant hurdle for any attacker. Instead, the browser has become a more interesting attack vector, as users routinely run untrusted third-party code on their devices within the browser. Since GPU computing can also offer advantages for computations within websites, browser vendors decided to expose the GPU to JavaScript through APIs like WebGL and the upcoming WebGPU standard. WebGPU is not only available on desktop browsers but is also already partially supported on mobile

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ASIA CCS '24, July 1–5, 2024, Singapore, Singapore
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0482-6/24/07
https://doi.org/10.1145/3634737.3656283

devices in Chrome Canary version 117. As the future standard for web-based general-purpose interaction with GPUs, WebGPU aims to lay solid foundations for performance and security. The standard already has explicit mitigations against timing side channels [17], e.g., disabling timer access (making it a trusted feature), and mimicking the JavaScript mitigation against malicious use of the `SharedArrayBuffer` [18–21]. Previous work demonstrated native code side-channel attacks on GPUs, where the browser triggered L1 and L2 cache activity, e.g., through WebGL [12]. However, the feasibility of a browser-based GPU cache side-channel attack, targeting a victim running in native code or another browser window, nor the possibility of an attack with the upcoming WebGPU standard [22] have been demonstrated yet. Considering the ubiquitous attack surface browsers offer to attackers, we need to investigate the following questions:

Can GPU cache side-channel attacks also be mounted from within a restrictive browser environment using APIs like WebGPU? Can these attacks be made generic enough to work on the wide spectrum of GPU hardware? To what extent can an attacker leverage GPU parallelization to enhance attacks?

In this work, we answer these questions by presenting the first end-to-end cache side-channel attacks from within browsers, leveraging the new WebGPU standard. Despite the inherent restrictions of the JavaScript and WebGPU environment, we construct new attack primitives enabling cache side-channel attacks with an effectiveness comparable to traditional CPU-based attacks. Our attacks are generic and automated, in the sense that our 2 attack primitives automatically determine GPU-specific configuration parameters required for an attack, *i.e.*, the cache hit-miss threshold, the cache size, and the number of cache sets. Consequently, our attacks work on a wide variety of devices, which we demonstrate in our evaluation: We show that our 2 basic attack primitives work on 11 desktop GPUs from 5 different generations and 2 vendors, NVIDIA and AMD. We demonstrate that based on these, we can also identify cache sets and monitor cache set collisions directly from a browser on a variety of NVIDIA GPUs.

We introduce 3 techniques to exploit cache contention on the L2 cache of discrete GPUs from JavaScript via WebGPU compute shaders. First, we highlight that significant cache eviction, often induced by graphical rendering, can enable attackers to discern instances of re-rendering. Second, we implement a templating attack within the browser, designed to monitor memory access patterns. Lastly, we present the first Prime+Probe attack on discrete GPUs executed from a browser. For all 3 attacks, we evaluate whether using the GPU's parallelism improves the basic attacks. For the eviction set construction in particular, we extend the state of the art by leveraging the massive parallelism of modern GPUs with the first parallelized eviction set construction algorithm.

We evaluate our attacks in 3 distinct scenarios covering both low-frequency non-repeatable events, as well as repeatable and high-frequency events: an inter-keystroke timing attack, AES key extraction, and the establishment of a covert channel, all initiated from a browser, *i.e.*, through an attacker-controlled website. Our keystroke monitoring attack detects inter-keystroke timings with F_1-scores in the range of 82 % to 98 %, and a sampling time below 15 ms, fast enough to distinguish even very fast typing. We successfully extract AES keys in 6 min with a precision of 100 %. Our templating approach enables us to profile the T-tables in 13 s on average, with the remaining time (5.7 min) dedicated to the last round attack. We perform the attack on 2 recent GPUs, a NVIDIA RTX 3060 Mobile and a NVIDIA RTX 3060 Ti, with similar results. Lastly, we demonstrate a covert channel with true channel capacities between 7.3 kB/s and 10.9 kB/s on the NVIDIA RTX 2070 Super, NVIDIA RTX 3080 and NVIDIA RTX 3060 Ti.

Our attacks require no user interaction and work within a realistic time frame a user might spend on a website, e.g., in the range of multiple minutes. Therefore, they can easily be implemented as drive-by attacks, targeting arbitrary users while browsing the Internet. Furthermore, since our attacks are based on WebGPU, they are applicable to all operating systems and browsers implementing the WebGPU standard and, as we demonstrate, to a broad range of GPU devices. Consequently, it becomes clear that browser vendors need to reassess their approach to offer GPU access to untrusted websites without user consent. Instead, we recommend a security-centric interactive approach that is already applied to all other security- and privacy-related resources, such as the microphone and the camera.

In summary, our paper makes the following main contributions:

(1) We present the first end-to-end cache attacks on GPU caches from the browser, using the restrictive WebGPU API.
(2) We evaluate our attack primitives and attacks on a wide range of GPU architectures and explore where the massive parallelism of GPUs can improve attacks.
(3) Based on our insights, we develop the first parallel eviction set construction algorithm and the first Prime+Probe attack on the L2 cache of a single, dedicated GPU.
(4) We describe a novel templating approach that we use in an attack on an AES T-table GPU implementation. Using predictable LRU cache set eviction cascades on GPUs, our attack can skip the lengthy set-construction phase by exploiting only contention in sets that are actually used by the victim.

Outline. Section 2 provides the background and Section 3 our threat model. Section 4 presents the primitives for Prime+Probe on the GPU from the browser. Section 5 explores an inter-keystroke timing attack. Section 6 evaluates our attack for an AES key recovery and Section 7 in a covert channel scenario. Section 8 discusses limitations and mitigations. Section 9 concludes.

2 BACKGROUND

2.1 GPU architecture

The architecture of discrete GPUs may vary by brand. Hereafter, we focus on giving an insight into discrete GPUs architecture, tackling both computation and memory management. We default to the concepts and notations adopted by NVIDIA, but similar concepts are used by other manufacturers, such as AMD.

A GPU consists of multiple Streaming Multiprocessors (SMs), called Compute Units (CUs) on AMD cards. Each SM has its dedicated memory subsystem, including shared memory (SM-local memory), caches, and functional units, to execute multiple threads in parallel, operating under the SIMD paradigm. On GPUs, threads are organized into *thread blocks* (also called *workgroups* on WebGPU) that are assigned their own SM when executed. SMs consist of multiple processing blocks (4 on recent NVIDIA and AMD GPUs). Each processing block is a separate SIMD execution unit with its

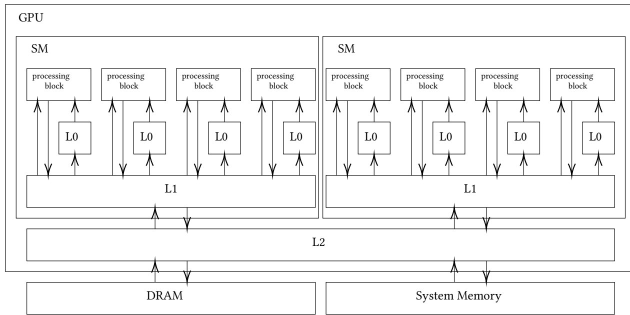


Figure 1: Modern GPUs (here NVIDIA) have an L0 cache per processing block, an L1 cache per SM, and a shared L2 cache.

own load and store units capable of running 32 threads in parallel. Thread blocks are divided into *warps*, groups of 32 threads that are scheduled on processing blocks. Processing blocks have warp schedulers, hardware schedulers that schedule *warps* in and out of the processing block. When a *warp* has to wait for a memory access or register dependencies, the warp scheduler schedules a different warp that is ready to execute to keep the SIMD units busy. The constant rescheduling of warps allows for latency hiding and, therefore, more efficient use of processing blocks [1].

Prior to the Volta architecture, all threads in a warp share the same instruction unit with a single program counter, *i.e.*, instructions execute in lockstep [23]. If threads in the same warp diverge, they are masked until they converge again. If some threads execute the if-branch and some execute the else-branch, the entire warp executes the if-branch and the else-branch with threads masked accordingly. Volta introduced independent thread scheduling, with a per-thread program counter and call stack, allowing the processing block to interleave execution of diverging branches [23].

Similar to CPUs, GPUs use caches to reduce the latency for memory accesses. Namely, each SM has a dedicated L1 cache that is shared between processing blocks, and each processing block has access to a smaller private L0 cache. Finally, GPUs share one global L2 cache (LLC) between the SMs. Figure 1 illustrates the cache hierarchy of Nvidia Turing GPUs. However, we note a few relevant peculiarities of GPU caches. First, there is no coherency protocol between caches, and maintaining coherency is the responsibility of developers. Second, unlike the classical 64-byte cache line in CPUs, GPUs' LLC commonly has 128-byte cache lines [24, 25].

2.2 GPU APIs

GPUs can be called through different APIs depending on the context. We distinguish two main API families: native APIs (e.g., OpenGL, Vulkan and CUDA), and web APIs (e.g., WebGL and WebGPU).

Native APIs. The most straightforward and efficient way to interact with a GPU is through dedicated native APIs. They enable the use of GPUs for either graphic rendering or generic computing. OpenGL was introduced in 1992 to support GPU-assisted rendering on Linux and Apple platforms, while Windows uses the Direct3D framework. For general-purpose computing, OpenCL, released in 2008, provides support for all major GPU vendors and is widely used. In 2007, NVIDIA released CUDA [1], a compute language specifically designed for NVIDIA GPUs. CUDA is supported by both consumer- and business-oriented NVIDIA GPUs. The high market share of NVIDIA GPUs and the ease of use of CUDA makes it the currently most widely used framework for general-purpose computation on GPUs. Apple recently dropped support for OpenGL in favor of Metal. Similarly, Vulkan was released in 2016 as a modern alternative to OpenGL. While these APIs have their differences, typical calls include operations on texture mappings, rasterization, and memory management on the GPU.

Web APIs. WebGL is the current baseline JavaScript API giving access to the GPUs rendering. As its name suggests, it was originally designed with a specific goal: graphic rendering in browsers. Hence, its API is limited and does not provide support for generic computations on the GPU [26]. This was the motivation behind the *WebGL 2.0 Compute* initiative [27]: "to bring compute shader support to the web via the WebGL rendering context". Due to the emergence of new native rendering APIs, the diminishing prominence of OpenGL, and the perceived constraints of WebGL, the project contributors decided to deprecate it [27] in favor of a more contemporary alternative, namely WebGPU.

Like WebGL, WebGPU provides access to the GPU graphics capabilities in the browser. It is, however, not a mere wrapper around OpenGL. More than that, it aims at being cross-platform and supporting modern graphic APIs, such as Vulkan, Metal, and DirectX, through JavaScript. Compared to WebGL, it offers a cleaner API, significantly better performance, and a more generic application range. At the time of writing, the standard is still under active deployment. However, the involvement of major browsers in this process, and the promising performance, foreshadow a widespread deployment in the next years. Chrome, Chromium, and Microsoft Edge already support WebGPU in their official release, and Firefox has it in its Nightly version [28]. Support of mobile GPUs is also in progress, with recent deployment on Android [29].

Developers can create rendering pipelines and manage GPU resources with WebGPU. WebGPU has its own shader language called WebGPU Shading Language (WGSL) to write custom shaders that are compiled at runtime. While WebGPU provides access to GPUs through native APIs, implementations of the standard may restrict the available GPU resources, e.g., memory and runtime, for security reasons. Without restrictions, big WebGPU workloads could significantly impact the useability of the host system [30], as most GPUs only allow for one active shader at a time.

2.3 Prime+Probe

In the last decades, microarchitectural attacks have been studied extensively. Prime+Probe [31, 32] is a cache-based attack that exposes the memory access patterns of a process by exploiting cache contention to leak the cache set accesses. This technique is particularly useful for attackers with limited control over the victim's machine, since it has low requirements and does not need shared memory or direct control over the cache with a flush instruction. Because of these weak assumptions, it is well-suited for browser-based attacks, where an attacker controls JavaScript on a web page [33, 34].

Assuming the attacker can execute code on the same processor as the victim, the attack works as follows. First, the attacker *primes* the cache by filling well-chosen cache sets with its own data. Then, they wait for the victim to make memory accesses. Finally, the

attacker *probes* their data to access the same cache sets as before. If the victim accessed one of the sets monitored by the attacker, they will have evicted some of the attacker's data, causing a longer latency in the probing phase. In the context of a covert channel, the attacker would run both sender and receiver, and use the contention on the cache sets to build the channel.

2.4 Related Work

Covert and side channels on GPUs. Naghibijouybari et al. [11] describe multiple covert and side-channel attacks on GPUs. Wu et al. [35] exploit rendering contention to perform side-channel attacks on browsers. Many works consider a spy outside of the targeted GPU. Jiang et al. [7–9] present a cache-based attack, a shared memory attack, and a bank-conflict attack, all leading to a key recovery attack on AES. Similarly, Ahn et al. [6] exploit cache conflicts to recover an AES key from a GPU implementation. While they rely on cycle-accurate timers, our attack works from the browser without a timer. In addition, their spy uses the native API, while we perform our attack from the browser. More similar to our approach, Dutta et al. [36] perform Prime+Probe on Intel's integrated GPU through contention on the LLC shared between the CPU and GPU with native OpenCL. They also demonstrate ring-bus interconnect covert channel reaching the LLC. Dutta et al. [16] present a cross-multi-GPU Prime+Probe covert channel based on L2 contention. Our threat model is different, assuming a spy co-located on the same GPU in a drive-by attack from the web.

Naghibijouybari et al. [10, 12] present the first attacks in the co-located setting. Their first work [10] presents an in-depth study of General Purpose GPUs and highlights various ways to build covert channels on GPUs using caches and functional units. In their following work [12], they demonstrate the ability of an attacker to implement website fingerprinting based on GPU memory usage and performance counters. They also demonstrate the practical impact of their attack by tracking keystrokes from users and recovering some internal parameters of a neural network running on the GPU. Wei et al. [3] present a similar approach, using the GPU context-switching impact on performance counters to enhance the leakage and recover the complete structure of a neural network.

Despite the groundbreaking nature of these works, our contributions differ in key aspects. All aforementioned contributions rely on the attacker having access to the native APIs of the GPU through CUDA or OpenGL. This enables them to monitor high-precision performance counters. Our attack works entirely from the browser using JavaScript, with the corresponding API limitations (e.g., we do not have an accurate timer). This results in better portability but also a weaker attacker in our threat model.

Browser-based cache attacks on GPUs. The growth of web-based API usage to offer GPU-enhanced rendering inadvertently enables attackers to run GPU-based attacks through JavaScript, bypassing its existing limitations. To our knowledge, all existing works exploit the GPU through the WebGL API. Frigo et al. [13] leverage the integrated GPU to mount Rowhammer attacks from browsers on mobile devices, using the WebGL timing APIs. In response, major browsers disabled this timer. Cronin et al. [15] presented a browser-based attack with assumptions similar to ours. They target SoC platforms and leverage system-level cache occupancy to build a covert channel and fingerprint websites. They differ from our work in multiple aspects. First, they focus on a SoC system and use contention on the system-level cache, which is shared between the CPU cores and its peripherals (namely the GPU) in ARM systems. This enables them to create contention from the CPU, whereas we consider spy and attacker to be co-located on the GPU. Second, the cache occupancy of the system-level cache is significantly different, resulting in different challenges to overcome. Finally, they exploit it using WebGL code, while we focus on its successor, WebGPU, which claims to consider and address the side-channels threat. Recently, Taneja et al. [37] demonstrated hybrid side channels on the CPU and GPU, based on how they adjust their frequency, power, and temperature depending on the workload. They demonstrate that GPUs exhibit instruction and data-dependent throttling. Their JavaScript attack assumes a victim in the browser but still relies on the ability of the attacker to access native APIs to monitor the power consumption and frequency of the GPU.

```
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);
} } }
else {
  start = atomicLoad(&timer);
  var c : u32 = atomicLoad(&buffer); //access
  if c != 0 { //prevent optimization
    return;
  }
  end = atomicLoad(&timer);
  atomicStore(&stop, stop_value);
}
```

Listing 1: Counting thread implementation in WGSL based on the global thread id and `atomic` operations.

3 THREAT MODEL

As we target WebGPU, our primary requirement is a browser with WebGPU support. As of writing, this includes Chrome releases since version 112, Chromium, Edge, and Firefox Nightly. By targeting web browsers, our threat model includes any scenario where a browser might run while sensitive information is being processed. Because the entire system usually shares the GPU, this can include anything rendered (such as websites or applications) and general-purpose computing operations. We show that our attack can be done in a drive-by manner, simply by visiting a website for a while. We assume that the victim will visit an attacker's page for several minutes, e.g., reading a blog with malicious WebGPU code. We do not assume that WebGPU provides any interface for hardware timers. To further constrain our attacker, we assume that WebGPU provides no workgroup memory in reaction to prior work [36]. In this paper, we attack dedicated NVIDIA and AMD GPUs, whereas some other works [13] have focused on integrated mobile GPUs.

Table 1: Timing thread counter value for the 98th and 5th percentile for L2 cache hits and misses, respectively, for a variety of GPUs and the methods add and store. A good threshold can be found when the distributions are clearly separable. n = 1 000 000 hits and misses were recorded each.

		Add		Store	
	GPU	$hit_{>98\%}$	$miss_{<5\%}$	$hit_{>98\%}$	$miss_{<5\%}$
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

4 WEBGPU PRIMITIVES

To build advanced cache attacks in WebGPU, we need several key primitives. The first is a timer accurate enough to reliably distinguish a cache hit from a miss. Using this timer, we can then detect cache size, cache activity, and build Prime+Probe eviction sets. While these primitives have been extensively studied on CPUs [31–33, 38], building them on GPUs involves some difficulties, especially from a browser. In this section, we detail these challenges and how to overcome them using WebGPU and minimal requirements.

4.1 Timing without clocks

Most previous works on GPUs are run natively and rely on high-precision timers or related performance counters for their measurements. However, WebGPU took explicit measures to preclude timing attacks, such as making `timestamp-query` optional and limiting interactions via shared buffers, similar to `SharedArrayBuffer` mitigations in browsers. To our knowledge, the shader language WGSL does not include any timers at this point. To present a generic primitive, we will construct our attacks without API-provided timers.

JavaScript encounters the challenge of imprecise timers, so prior works on the CPU had to consider similar constraints and employed counting threads [18, 19, 36, 39]. The idea is to set up a shared memory buffer and use a dedicated thread to constantly increment a shared variable. Another thread can then read this variable and interpret its value as a timer. When applying this concept to WebGPU, we face three challenges.

C1. Threads serialization. Different compute shaders cannot run at the same time. Therefore, the same shader needs to count on one thread and execute the attack code on another, as shown in Listing 1. While this would be straightforward on the CPU, GPU threads scheduled on the same processing block may run in *lockstep* on some architectures [40]. This means that if threads in the same warp need to execute different instructions (*warp divergence*), they would run sequentially, hindering the use of our counter as a timer.

C2. Memory coherency. Unlike for CPUs, GPUs have no automatic coherency guarantee in the memory hierarchy. Each SM manages a dedicated memory subsystem, so SMs may contain different copies of the same data in their L1 caches. Maintaining a coherent state by synchronizing the data is left to the developers. Therefore, without coherency, our counting thread would increment the timer in its private L1 cache, unobservable from the outside.

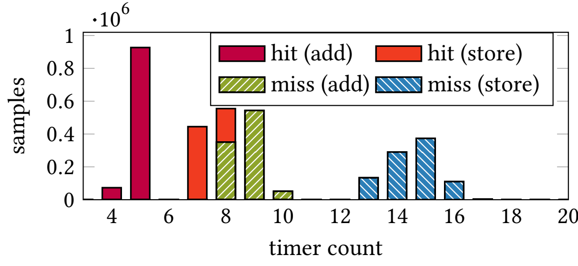
C3. Optimization. The WGSL compiler aggressively optimizes the code, such that a counting `while` loop may be replaced with the final result, and memory accesses may be replaced by registers.

Solutions. In their OpenCL implementation, Dutta et al. [36] solve the first challenge by executing enough counting threads to fill one or more warps. Then, they conduct the attack in a separate warp within the same SM, so each warp only executes the same branches, avoiding warp divergence. To address the other challenges, they simply store the counter in a shared memory region available to all threads in the same workgroup.

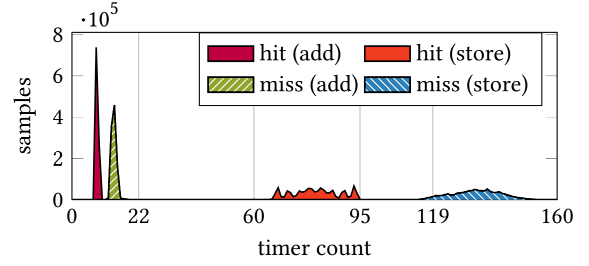
In line with our goal to get a portable and low-assumption attack, we suggest a more generic approach. To address **C1**, we set the workgroup size of the shader to 1, which prevents scheduling on the same processing block. Our solution demonstrates that even a strong security measure, like disabling shared memory, would not prevent timers in WGSL. We can solve **C2** and **C3** using `atomic` instructions (see Listing 1). First, they guarantee that memory accesses will not be turned into register accesses for optimization. Second, loads and stores performed by `atomic` instructions bypass the L1 cache to directly access the L2 cache, which enforces coherency. However, **C3** presents an additional challenge not solved by `atomic` operations. Sometimes the compiler will reorder or drop the measured load. We can prevent this by using the loaded value in a condition whose outcome the compiler does not know.

A minimal example of this approach is illustrated in Listing 1. One thread is chosen to be the timing thread via the global invocation id `global_id`, while the other can perform the attack. To spend as little time as possible reading the `stop` variable, the timing thread spends most of its time in a tight inner loop. All memory operations interacting with other threads are done with `atomic` instructions. The condition in Line 12 prevents compiler optimization of the load order or elimination of the target load.

Table 1 shows the hit-miss separation for both techniques on 11 different GPUs. We also find that on most cards, incrementing a local variable and updating it with `atomicStore` is significantly faster than using `atomicAdd`, since the latter is a blocking operation that requires waiting until the data is brought to the execution unit. However, this technique seems to be much less effective on AMD in general, but also on some Linux configurations (noted as Lin, as opposed to Windows default). Our testing revealed that this optimization may depend on multiple factors, such as the operating system and driver version (see section 8). This optimization is of course only applicable to data accesses in L2 cache that are not meant to be timed. Figure 2 also highlights this difference but confirms that L2 cache hits and misses are clearly distinguishable in either case. In the end, our timer primitive cannot be prevented without removing the `atomic` operations, and its accuracy is only limited by the time it takes to load from and store to the L2 cache.



(a) AMD RX 6800 XT



(b) NVIDIA RTX 3080

Figure 2: WebGPU cache hit and cache miss histograms for different GPUs with counting thread for 1 million samples. Adding to a memory location provides less resolution than storing a register value. Higher counts show higher timer resolution.

4.2 Cache-Size Detection

As our goal is to show that virtually all WebGPU-enabled devices are affected by these generic attacks, we try to hardcode as few parameters as possible. An important parameter for all further sections is the cache size. It determines how many sets we can expect (Section 4.3) and lets us derive suitable buffer sizes for cache eviction detections (Section 4.4).

We assume standard LRU, as suggested in previous work [41], and fill the cache with a large array of 10 MB. The buffer size choice is motivated by our observation that most GPUs have below 8 MB of L2 cache. In the same shader execution, we now iterate over the array forward and then backward, counting hits. Changing direction avoids self-eviction of an entire set after a single miss and allows us to accurately measure the number of cache lines that remain in the cache. If the hit rate is very high (> 95 %), we increase the test size in steps to 40 MB, 80 MB, and 100 MB. This keeps measurement times low for most cards, while allowing accurate detection even for larger caches. Finally, we match this approximate size to the closest larger size within a list of known sizes.

Table 2 shows that for most cards, we can reliably determine the cache size in less than 400 ms. Of interest among the outliers is the NVIDIA RTX 3060 Ti. It reliably returns a size of 3 MB, and indeed, we never see any hits more than exactly 3 MB, though the official L2 cache size is 4 MB. A simple explanation is that both our NVIDIA RTX 3060 Ti models, only have 3 MB of L2 cache. Another possibility is that these cards have a different mapping function, and some part of their cache is only reachable for much larger total VRAM allocations. We will encounter this again in Section 4.3.

Table 2: Our WebGPU cache-size finding algorithm on a variety of GPUs, $n = 10$. With one exception, the correct size is almost always found on all cards.

		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

4.3 L2 Cache Eviction Set Construction

The next step in building a Prime+Probe attack is to find a set of addresses that map to the same cache set. To make sure this set of addresses replaces all the current entries in the cache set, the cardinality of the set should at least match the cache associativity W. We call this an *eviction set*. On CPUs, much work has been done to reverse engineer the mapping from virtual-to-physical addresses to cache sets [42–44]. Comparable work on GPUs [16, 41, 45–47] however has shown that their cache set mapping can be much more complex. Jain et al. [41] used a modified driver to reverse-engineer the hash functions for mapping addresses to both cache and VRAM on an NVIDIA GTX 1070 and 1080. However, our tests suggest these functions differ in newer generations of NVIDIA GPUs. In particular, many GPUs need to employ different non-linear (or linear, but different by address range) mapping functions due to their non-power-of-two cache and VRAM sizes. Additionally, AMD or even mobile GPUs may follow an entirely different scheme altogether. Like the work by Dutta et al. [16], we also do not have the advantage of relying on physically contiguous memory, or any specific page size.

In keeping with our generic approach, we do not attempt to rely on any known mapping functions or page sizes. Instead, we employ a generic set-finding algorithm based on prior work for CPU caches. Given a timer accurate enough to distinguish cache hits and misses, an attacker should be able to create eviction sets efficiently, similar to the methods presented by Qureshi and Purnal et al. [48, 49].

While this approach works well for CPUs, we encountered various challenges to efficiently port it to GPUs. Hereafter, we describe a novel approach to compute fast and reliable eviction sets on GPUs. In particular, we describe how to leverage the powerful parallelism that GPUs offer to speed up this process.

The basis of our implementation is the Group-Elimination Method (GEM) [48]. The goal of GEM is to find an eviction set for a target address. To this end, a large set of addresses $S >> W$ that evicts the target address is partitioned into $W + 1$ groups. As a full eviction set of W addresses must be contained in some combination of $\leq W$

```
1  𝓢 ← {1.5x cacheSize, 128B steps}
2  Buckets ← {{}}
3  while 𝓢 != {}
4    𝓑 ← 𝓢, 𝓟 ← 𝓑[0] ❶ //initialize B, select a pivot P
5    while |𝓑| > targetSize
6      𝓖 ← {}
7      do
8        shuffle(𝓑)
9        𝓑 ← 𝓑 ∪ 𝓖, 𝓖 ← 𝓑[0:1/2W|𝓑|], 𝓑 ← 𝓑\{𝓟 ∪ 𝓖}
10       access(𝓟), parallelAccess(𝓑) //access pivot, then B
11     while isCached(𝓟) ❷
12     if optCondition() 2b
13       hits ← accessAndMeasure({𝓑 ∪ 𝓟})
14       𝓑 ← 𝓑\hits //remove addresses not part of eviction sets
15   𝓑 ← 𝓑 ∪ 𝓟, 𝓢 ← 𝓢\𝓑
16   Buckets ← Buckets ∪ {𝓑}
```

Listing 2: Parallel Set Construction. This simplified pseudocode algorithm partitions an initial set of addresses S into several buckets B whose addresses do not share cache sets.

```
1  Buckets = ParallelSetConstruction()
2  originalBuckets ← Buckets, EvictionSets ← {}
3  Pivots ← {𝓑[0] | 𝓑 ∈ Buckets)} A
4  Buckets ← {{𝓑\𝓟} | (𝓑,𝓟) ∈ (Buckets,Pivots)}
5  while ∃𝓑 : 𝓑 ≠ {}
6    Gs ← {{}}
7    foreach 𝓑 ∈ Buckets : 𝓑 ≠ {}
8      if |𝓑| > 1000
9        𝓖 ← 𝓑[0:1/2W|𝓑|] B
10     else
11       𝓖 ← 𝓑[0] B2
12     Gs ← Gs ∪ {𝓖}, 𝓑 ← 𝓑\𝓖
13   AllHits = parallelMeasure(Pivots, Buckets) C
14   foreach (𝓟,Hits,𝓑,𝓖) ∈ (Pivots,AllHits,Buckets,Gs)
15     if isCached(𝓟)
16       𝓑 ← 𝓑 ∪ 𝓖
17       shuffle(B)
18     else
19       𝓑 ← 𝓑\Hits D
20       if |𝓑| == W //bucket has reduced down to one set
21         EvictionSets ← EvictionSets ∪ {𝓑 ∪ 𝓟}
22         𝓑 = originalBucket\{𝓑 ∪ 𝓟} //refill Bucket
23         shuffle(𝓑)
24         𝓟 ← 𝓑[0], 𝓑 ← 𝓑\𝓟  A
25       else if |𝓖| == 1 && |Hits| == W E
26         EvictionSets ← EvictionSets ∪ {𝓖 ∪ 𝓑} //free set!
```

Listing 3: Parallel Bucket Sifting. This algorithm sifts sets in parallel from the previously separated buckets.

out of the W groups, (at least) one group can be eliminated without affecting the eviction. GEM tries to remove each of the $W+1$ groups from the set S until one is found that does not influence the eviction of the target. This is repeated until only W addresses remain in S, forming an eviction set for the target address.

Our implementation differs from GEM in two significant ways. First, we aim to find all sets, and we, therefore, try to find more than one eviction set at a time. Similar to Prime+Prune+Probe [49], we make use of the predictable behavior of LRU for this. Second, we parallelize parts of the algorithm to multiple threads. Many constants in the following are empirically determined values that work on a variety of GPUs, not optimal values.

Parallel Set Construction. See Listing 2. When we access many addresses in parallel on the GPU (or the CPU), ordering between them is not guaranteed. This means that when a set is split between different threads, we can no longer expect to observe effects stemming from LRU. In effect, eviction *measurements* that rely on access order become meaningless. We, therefore, add a preprocessing step to the eviction-set construction algorithm.

The goal of preprocessing is to separate an initially large set S of addresses $s_i = |S|$ (1.5x the cache size in 128 B steps) into buckets with no overlapping sets. This partitioning facilitates the independent examination of each bucket for sets, circumventing inter-thread interference. The process follows a similar approach as GEM and is delineated in two main steps.

Starting with $B = S$, the first (❶) step involves selecting a random element from the set as our *pivot*. This pivot address guarantees the presence of at least one complete set within the bucket, although it probably contains several more. The second (❷) step consists in removing a portion of the set, *i.e.*, a group, and verifying if the pivot element is still evicted by the residual B. If eviction is not observed, we reiterate with another group. Contrary to GEM, we find that eliminating $1/2W|B|$ rather than $1/W+1$ better mitigates the excessive removal of elements in later steps (2b).

We also incorporate several optimizations not found in GEM. Until B diminishes to $3/4s_i$, we exploit parallelism by accessing all set elements concurrently using 30 threads, measuring only the pivot at the end. When $|B| < 1/6s_i$ or on every fourth iteration when $|B| < 3/4s_i$ (optCondition is met, Line 12), we measure not just the pivot but all other elements . We only do this sparingly because measuring elements in addition to accessing them has a significant overhead, and, as mentioned, LRU-related observations can't be parallelized while sets are still unknown. However, this enables a crucial optimization: the removal of all set elements that register a cache hit (2b). Given a consistent access sequence and a cache replacement policy approximating LRU, all persisting elements in B are now part of full eviction sets.

This procedure can be iterated until B is below a predetermined threshold. Empirical evaluations suggest a bucket size of 3500 (equivalent to $145-206$ sets) works for the majority of GPUs. Upon completion, B is subtracted from S. We are left with a bucket of the desired size, exclusively containing eviction sets. The residual segment of S does not include overlapping sets with the bucket. Repeating the previous steps ensures that the final buckets consist only of non-overlapping eviction sets, collectively representing nearly all cache sets.

Parallel Bucket Sifting. With full eviction sets sorted into roughly equal buckets, we can now begin to extract single sets from them. Since there is no more overlap between the cache sets in the buckets, we can now run measurements on them in parallel. For the NVIDIA RTX 3080 and its 5 MB cache for example, the previously mentioned target bucket size produces around 16 buckets of 160 sets each, with up to 24 addresses per set. This means we can start a loop on our 16 input buckets with the following broad steps running in parallel for each bucket. First, we once again shuffle the elements in each bucket B and pick a pivot element to find an eviction set for (A). Second, like before, we remove groups of $1/2W|B|$ elements until we find one that doesn't affect the pivot's eviction (B). Third, to determine eviction, we measure the access latency for all addresses remaining in all buckets in parallel. (C) Fourth, addresses in buckets that show cache hits are also removed, such that all remaining addresses

still form eviction sets within B (D). Buckets are shrunk in parallel this way until a bucket's size goes below 1000 elements. At this point, instead of $1/2W|B|$, we remove only a single element per loop (B2). This allows us to make use of the *cascading eviction effect* of the LRU replacement policy: when we remove only one address, that together with W other addresses in B forms an eviction set, those W addresses will now show up as cache hits (E). In effect, we have found an entire eviction set in a large bucket B by removing a single element. This allows us to *sift* out many sets for "free" while trimming the bucket to find the pivot element's eviction set. We continue decreasing the bucket size until either a complete eviction set for the pivot remains, or some false measurement has left us with an incomplete set. At this point, we refill the bucket with all discarded addresses that could not be attributed to a complete set and start again at step one. The algorithm terminates when either all buckets are empty, or no new sets have been found for too long.

This sifting method is so effective, in fact, that it finds significantly more sets than the number of pivots chosen. On our NVIDIA RTX 3080, for example, we might search 17 buckets for eviction sets with 90 chosen pivots (an average of 5.2 bucket "refills"), but sift out 2465 sets on the way. The change at 1000 elements represents an empirically found trade-off between fast bucket-shrinking and a high amount of sets found through sifting. When the number is too high, the time to find sets will needlessly increase, as most sets start with an average of 24 addresses, but can only be detected when just 16 are left in the bucket. When it is too low, many sets are lost to the sifting method through the removal of many elements.

Combining these optimizations, we can map most sets in the L2 cache of all NVIDIA GPUs in WebGPU in a reasonable time frame, as shown in Table 3. The notable exception is the NVIDIA RTX 4090, as the enormous cache size presented problems not found in other cards. Likewise, both AMD cards fail this important step to further attacks and are therefore not included in the more advanced attacks. One possible explanation is that the timing difference to other cards, which can already be seen in Table 1, causes more noise, as the hit and miss distributions are closer together. While we believe that from the basic timing difference, it is clear that all our attacks *could* run on these cards, we only had temporary and time-restricted remote access to these GPUs, which did not allow for analyzing the underlying problem. The NVIDIA RTX 3060 Ti also sticks out, as it consistently finds close to 1536 sets even when looking for 2048. This is consistent with 3 MB of L2 cache found in our experiments (see Section 4.2).

We see that the percentage of sets we find varies along with the time, though a majority of sets can almost always be found within 5 minutes. With this additional primitive, attackers can implement Prime+Probe to build a covert channel, as we show in Section 7, or execute some other cache attack, e.g., Rowhammer [13].

4.4 Full Cache Evictions

One of the first observations while measuring cache hit rates on GPUs is that some events evict a sizeable portion of the cache. Whenever an element on the screen is redrawn or the frame buffer is refreshed for some other reason, this occupies a large part of the cache. Depending on the total size of the L2 cache and what is being drawn, this may even evict the entire cache. On the one hand, this presents as noise during some attacks; each measurement that happens after a draw event is tainted. On the other hand, these evictions are indicators of activity on screen and can therefore be used as a side channel to user activity. We describe an inter-keystroke timing attack based on this primitive in Section 5.

Table 3: Our WebGPU set-construction algorithm on a variety of GPUs, n = 10. All but one card reliably find > 80% of sets.

	GPU	Sets: Overall	Sets: Found $\bar{x}$ %	Sets: Found σ %	Runtime: $\bar{x}$ min	Runtime: 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

5 FULL-EVICTION KEYSTROKE MONITORING

Starting from the observation that drawing elements on screen evicts a significant part of the cache, we build an attack that records inter-keystroke timings by observing cache contention. As shown in prior work [50–53], inter-keystroke timings carry significant information and can lead to password recovery.

While subsequent sections of this paper present conventional benchmarks for high-frequency side channels, this section focuses on low-frequency benchmarking. Despite the infrequent occurrence of events, achieving a high detection rate is crucial for accurately measuring inter-keystroke timings. In addition, keystroke profiling represents a practical application of our attack, as our setup mirrors the most prevalent end-user scenario: a computer equipped with a single discrete GPU engaged in internet browsing. As the full WebGPU standard becomes increasingly integrated into mobile devices, this scenario will gain further relevance in the future. Our approach for this attack is similar to Naghibijouybari et al. [12].

5.1 Construction

The attack is based on the following observation: for each character typed, the text box is re-rendered. We can measure this as the eviction of a certain amount of the cache, up to the entire cache, correlated with the size of the rendered area. To see this effect, we use a buffer that covers a part of the cache size and repeatedly measure its hit rate. Whenever we see a hit rate below a well-chosen threshold, e.g., 50 %, we record the timestamp as an event. The time resolution of this attack is determined by how fast our attacking shader can complete its measurement, which is determined by the total buffer size. Though on some GPUs we see that a small percentage of the cache is already enough to observe keystrokes, we find that for most, 35 % is a good tradeoff between detection and speed. The screen resolution, size of the text box and zoom level all contribute to the amount of evicted cache lines.

After recording raw traces, we filter based on two observations. First, very close measurements (<25 ms difference) are unlikely to

be separate keystroke events. Second, after a short break in typing, the cursor starts blinking at a 530 ms interval on Windows. Filtering these sources of noise removes most false positives. Figure 3 shows the trace of an attacker typing at varying speeds compared to the ground truth on our NVIDIA RTX 3080. We can see that while we measure some spurious events, most timings are accurate.

5.2 Evaluation

We tested this attack with a small text box and generated input directly injected from javascript, randomly drawing inter-keystroke timings from distributions similar to the patterns observed by Song et al. [50]. Table 4 shows the tested GPUs and their F_1 scores and inter-keystroke timing errors. During this test, no other visual noise was present, similar to the static login pages of many websites. The consistently high recall shows that virtually no keystrokes are missed on most cards. However, even after filtering, the recall shows that there is a low average of false positives for most cards. AMD once again behaves differently. Despite the high recall, with the low precision, we can consider this attack mostly failed or severely degraded. The results suggest either a high level of noise or, more likely, frequent misclassification of hits as misses due to the close timing differences visible in Table 1.

An interesting example for the timing resolution is the NVIDIA RTX 4090. Because of its large L2 cache of 72 MB, simply measuring cache contention requires a disproportionately large measurement set. This is because the cache footprint of a text box does not increase with the cache size. While all other cards easily reach a sampling rate below 15 ms, the huge buffer means that each measurement takes more than 200 ms, making an inter-keystroke timing attack with this method impractical.

We also observe that on Windows, the blinking of the cursor causes slightly less eviction than a typed character. One possible explanation is that instead of re-rendering the entire text box, the cursor is drawn on top.

6 AES KEY RECOVERY WITH BROWSER-BASED TEMPLATING

Recovering AES keys from vulnerable T-table implementations has become a benchmark for assessing how fine-grained side-channel attacks and microarchitectural attacks are. This case study has also been adopted in GPUs [6–9]. Additionally, AES has been proposed as a use case for general-purpose GPU computing since 2007 [4, 5].

Unlike previous research, our method adopts a set-agnostic approach, eliminating the need to understand cache set sizes or mappings. Traditional set-based strategies would require extensive profiling of cache sets and mapping of T-table accesses. Our methodology bypasses this initial step, focusing on locating addresses congruent with T-table lines.

6.1 Threat Model

Like earlier, we assume our attacker embeds some malicious JavaScript in a webpage the victim is browsing for several minutes. The victim runs a GPU-based AES implementation that can be queried for encryption with a chosen plaintext and key. The attacker aims to recover the AES key used by a victim. This scenario could be found in the case of an SFTP server, where the chosen plaintext and key represent downloading our own file. In order to implement a last-round attack, we assume the attacker has access to the victim's ciphertexts, but not the plaintext or the key.

6.2 AES Implementation Details

The native encryption service is an AES CUDA implementation, which uses combined T-tables for all rounds. This increases the difficulty of the attack compared to implementations that use separate tables for the last round, as all other rounds influence cache hits on table entries. As GPU cache line width is usually 128B, and each table is composed of 256 4-byte entries, each table fits in exactly 8 cache lines, for a total of 32 cache lines filled with table entries.

6.3 Attack Methodology

Our strategy, akin to the keystroke attack (Section 5), involves allocating a large buffer to occupy a significant cache portion, executing an AES encryption, and then identifying evicted buffer offsets using Prime+Probe. In an ideal scenario with a minimalistic AES kernel, evicted offsets would correlate with the table or the encryption's inputs and outputs. This is because, from our observation, GPUs implement a deterministic LRU-like eviction policy. This means that when one address from a full set is evicted, measuring all others in the same order used to place them in the set will cause a cascade of cache misses, as each new access will result in a miss, overwriting the next address. In practice, we find that kernel loading introduces substantial cache occupancy, leading to measurement noise. Our primary challenge is discerning the tables amidst this noise, and profiling each table's cache lines to track their access. We employ chosen keys and specially crafted plaintexts to deduce the relationship between our offsets and table entries. With this mapping, we can execute the traditional last-round attack [54, 55]. Hereafter, we delve into each step and the optimizations we employed to achieve a reliable key recovery in a drive-by manner.

Profiling T-Tables. The initial *profiling* phase allocates a sizable array (optimal size varies across models, even with similar cache sizes) in the browser, ensuring kernel loading and encryption evict specific offsets, and templates the AES encryption's memory accesses. Using random plaintexts, we would expect a random distribution of the access to each entry of the tables, thereby causing a predictable eviction frequency of the set-congruent offsets (at a frequency of 0.995). On the contrary, the offsets that are set-congruent to the memory required for kernel loading would be evicted every time, and other noise artifacts should be sparse. Our differential access templating, using a fixed key and chosen plaintexts, enhances profiling reliability and efficiency.

The strategy involves pre-generating 32 plaintexts p_i with a fixed key, ensuring the encryption of each plaintext access all but one cache line within the tables, and a reference plaintext p_r, which encryption accesses all cache lines. Comparing memory accesses during the encryption of p_i and p_r reveals offsets congruent to cache line i. This process identifies offsets that are set congruent to each cache line, though some cache lines may remain undetected due to kernel loading *noise*. This refined offset list streamlines the attack, focusing on a reduced subset of offsets.

Last Round Attack. As we are performing a last-round attack, the only requirement is that we can make measurements during

Table 4: Efficacy of WebGPU inter-keystroke timing detection on a variety of GPUs for 100 keystrokes.

		Performance Metrics			False		True	Interval Error		
		F_1 score	precision	recall	Positive	Negative	Positive	$\bar{x}$	σ	median
	GPU							ms	ms	ms
AMD	RX 6800 XT	0.27	0.16	0.99	533	1	99	133.58	101.18	121.50
AMD	RX 6900 XT	0.29	0.17	0.99	490	1	99	126.05	119.22	86.00
NVIDIA	GTX 1070	0.97	0.99	0.96	1	4	96	−0.28	4.25	0.00
NVIDIA	GTX 1650	0.82	0.70	0.99	42	1	99	12.13	20.43	1.00
NVIDIA	GTX 1660 Ti	0.87	0.78	0.98	27	2	98	5.73	26.52	0.00
NVIDIA	RTX 2070 SUPER	0.86	0.78	0.97	28	3	97	19.81	57.81	0.00
NVIDIA	RTX 3060 Mobile	0.98	0.99	0.98	1	2	98	0.01	1.49	0.00
NVIDIA	RTX 3060 Ti	0.97	0.98	0.97	2	3	97	1.76	21.12	0.00
NVIDIA	RTX 3080	0.94	0.92	0.96	8	4	96	1.27	14.44	0.00
NVIDIA	Quadro P620	0.98	0.99	0.97	1	3	97	−5.57	54.90	0.00

Table 5: Evaluation of the AES Last Round Attack (LRA) on two NVIDIA cards. All values are average across $n = 50$ runs.

GPU	Measurements (x1000)	Time (min) Profiling	LRA	Total
RTX 3060 Mobile	9.3±1.6	0.23±0.1	5.7±0.9	6.0±1.0
RTX 3060 Ti	9.8±3.4	0.23±0.1	5.9±1.9	6.1±2.0

encryptions by the victim, and observe the ciphertext. The attack aligns with the non-elimination method presented by Neve and Seifert [56]. The idea is, given a collection of ciphertexts and the access to T-tables entries that happened during the encryptions, to remove possible values on the key bytes by looking for cache lines that were *not* accessed in the process. For each cache line not accessed during the encryption, we can remove all last-round key bytes that would have resulted in a memory access during the last round, based on the ciphertext value. Given the cache line size of GPUs, we get up to 2^4 bit of information on the last-round key every time a cache line is not accessed.

The more cache lines we can monitor, the more likely we are to reduce the search space for the last-round key. Once we get below 2^{40} candidates, we switch to an exhaustive search of the key.

6.4 Evaluation

For our evaluations, we focus on a CUDA-based target implementation, rendering evaluations on AMD cards infeasible. Somewhat breaking with the theme of this work, the nature of this attack necessitates some parameter adjustments, which adds complexity and extends the evaluation duration compared to other attacks. Therefore, we settle on evaluating our attack on two recent cards: NVIDIA RTX 3060 Ti and NVIDIA RTX 3060 Mobile. We run all our experiments on Ubuntu 22.04 and Chromium 117 but we also have observed consistent results Chrome versions 112 to 115.

Table 5 showcases our findings. It highlights the average duration of the attack's primary steps and the mean number of encryptions required for successful key recovery. Notably, both cards yield similar results, recovering the key in 6 min. The average encryptions needed are 9300 and 9800, respectively. The uniformity of results across GPUs, coupled with the low standard deviation, underscores the stability and reproducibility of our attack.

Profiling the T-table takes on average 13 s. The variability in this phase predominantly stems from the inconsistent repetition of profiling until an optimal buffer size is identified, enabling sufficient eviction observation. Typically, a single profiling session lasts 6 s. Once profiling is complete, the same session can be repurposed to divulge multiple AES keys, thereby reducing the attack duration to the sample collection time needed in the concluding step.

Profiling often does not provide a complete mapping for every cache line. The disparities in measurements and time allocated for the last-round attacks correlate directly with the number of cache lines we can monitor. On average, we can spy on $^{20}/_{32}$ cache lines. The NVIDIA RTX 3060 Ti exhibits marginally less consistent results, occasionally mapping fewer cache lines, leading to an elongated attack duration and increased standard deviation. Our evaluation on both cards consistently had a 100 % success rate.

7 A PRIME+PROBE COVERT CHANNEL

A covert channel is a channel that is constructed on top of some shared resource that is not meant for data transmission. This allows an attacker to transmit data between two domains that should be isolated or strictly monitored. Because both sender and receiver work together to transmit data, covert channels are a valuable benchmark for any side channel's bandwidth. In a traditional Prime+Probe cache covert channel, the sender transmits bits by *priming* (evicting) cache sets to transmit a binary 1, which the receiver can later detect by *probing* (measuring) its own lines in the same set. With our reliable timer and a method to find the required eviction sets (see Section 4), we can now construct a Prime+Probe cache covert channel for the L2 cache.

The sender is a C++ application that uses CUDA. In this scenario, it is a malicious application without network privileges but with access to the GPU. The sender's goal is to exfiltrate sensitive data via a GPU covert channel. The receiver runs in a website the user visits at the same time. This may be a legitimate website with injected malicious JavaScript, or a website the user is led to in some way.

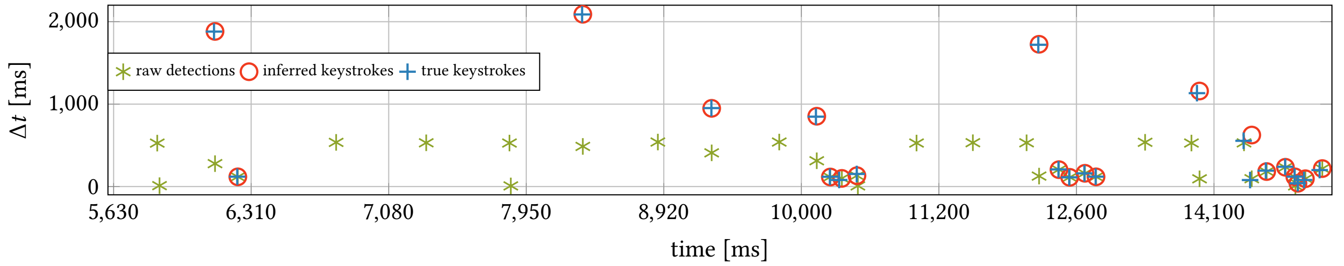


Figure 3: Inter-keystroke timing recovery on a NVIDIA RTX 3080. The raw activity detections (green) show prominent cursor blinking that can be filtered out very well, which leaves an accurate trace of inter-keystroke timings (red).

7.1 Construction

We write the **Sender** $\mathcal{S}$ in C++ and CUDA, making full use of the native high-resolution timer. The browser-based **Receiver** $\mathcal{R}$ uses a combination of JavaScript and WGSL. Using our eviction set construction (see Section 4.3), $\mathcal{R}$ starts by mapping all cache sets.

Setup - CJAG. As neither $\mathcal{R}$ nor $\mathcal{S}$ have absolute labels for their respective eviction sets, the first step is to communicate the shared sets from $\mathcal{S}$ to $\mathcal{R}$. For this, we implement a GPU-friendly version of the cache jamming agreement (CJAG) proposed by Maurice et al. [57]. In CJAG, $\mathcal{S}$ alternates between *jamming* a set, *i.e.*, evicting it continuously for some time, and *probing* the set for a slightly longer period. Meanwhile, $\mathcal{R}$ probes all sets continuously until the jammed set is detected. Then, $\mathcal{R}$ switches to a longer period of *jamming*, so that $\mathcal{S}$ knows the set has been received and moves on.

Unlike the CJAG approach on the CPU, distinct shaders do not execute concurrently on the GPU, making simultaneous detection and jamming infeasible. Rather than employing shaders that continuously loop through jamming or detection, we need to segment them into single invocations. Depending on the frequency of driver interruptions, we might otherwise see long shader executions that rarely interface with each other.

Additionally, we want to use a large number of sets (e.g., 1024). Serial transmission, as implemented in CJAG, is, therefore, impractical. Instead, we enhance the CJAG framework by leveraging the inherent parallelism of our GPU, enabling both $\mathcal{S}$ and $\mathcal{R}$ to jam and detect all sets concurrently. Here, copying to and from shaders is the main bottleneck, with 3 ms on average. Thus, the time difference between measuring a single set versus 64 sets per shader invocation is marginal. So, we combine both as a trade-off and measure sets in parallel on 16 threads. At this stage, $\mathcal{S}$ also swaps out any sets that are not detected from $\mathcal{R}$'s jamming, thus ensuring that all sets are fully functioning for both parties. After a selection of 1024 sets has been communicated, $\mathcal{S}$ switches to jamming on only half of all sets. The specific half is dictated by the current bit in the index number of its cache sets. In this way, $\mathcal{S}$ can transmit the order of all 1024 sets in $\log_2(1024) = 10$ steps by jamming different 512 sets for each bit. After all sets have been communicated, data transmission can begin. Table 6 shows that it takes 14 s to 28 s to transmit 1024 sets.

Transmission. After the set jamming agreement has been completed, the transmission is entirely one-way. We opt for a channel design where sender and receiver are synchronized with the wall clock. In native C++, this provides at least µs accuracy, while in browsers, this is limited to 100 µs. We choose a default transmission window length of 5 ms for our packets. This length is limited not only by the accuracy of the timer but also by the time it takes for a shader to run. To compensate for the long packet duration, we use the GPU's parallelism to transmit on 1024 sets concurrently.

While eviction for $\mathcal{S}$ is as simple as accessing many addresses in parallel in a loop, $\mathcal{R}$ still needs to measure time. Challenge **C1** (see Section 4.1) means that we need to separate each parallel thread into different workgroups to prevent lockstep execution. Additionally, individual sets always need to be measured by the same thread, as ordering between these accesses is crucial for the eviction policy.

As observed in Section 4.4, GUI-related events can introduce undesirable noise. Similarly, the operating system can deschedule $\mathcal{S}$ for periods of time. To reduce such noise, we adopt the following strategies. First, we employ a *majority vote measurement* approach where each set is measured as often as possible within a transmission window. By counting evictions and non-evictions, we obtain the result through a majority vote. Second, we access the addresses within each set in an *alternating order*. This ensures a consistent read from the most to the least-recently-used cache line, precluding the cascade of self-evictions that would arise if the oldest cache line were evicted. This lets us determine how much of a set was evicted and easily identify low-level noise. Lastly, we use a *differential measurement scheme*. Here, a pair of sets transmit 1 bit, and measurements wherein neither or both sets are evicted are discarded. In a valid transmission, precisely one set is evicted for every pair, effectively halving our transmission rate and resulting in a total packet length of 64 B. Consequently, the raw transmission speed is fixed by the parameters to a default of 12.8 kB/s.

7.2 Evaluation

We evaluate the covert channel on 3 NVIDIA GPUs; the RTX 2070 SUPER, 3060 Ti and 3080. The GTX 1070 and Quadro P620's Pascal architecture does not support all the instructions used by the CUDA sender. AMD cards do not support CUDA, though as set-finding fails on AMD (see Section 4.3) the attack would not work either way. The GTX 1650 and GTX 1660 Ti both support the instructions as well as set-finding, but we could not reliably establish communication because of malfunctioning jamming detection in CUDA.

Table 6 shows the configuration and transmission details for all tested devices. We can see that as we shrink the transmission window, average reads in the window go down, and the error rate increases. At 4 ms, the NVIDIA RTX 3080 shows a decrease in true channel capacity compared to 5 ms for this reason. Because

Table 6: Transmission results of our Prime+Probe covert channel from a native CUDA sender to a WebGPU receiver in the browser, $n = 10$. True channel capacity can vary widely either due to general noise, incorrect set transmission during CJAG or too few correct reads per window, *i.e.*, the number of correctly received pairs within the transmission window.

		Configuration			CJAG	Bandwidth			Bit Error Ratio	
	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 it's higher clock speed, the 3080 supports faster transmission than the two other cards. Its average true bandwidth in its fastest configuration is, therefore, 10.9 kB/s, at a BER of 2.2 %. Though our channel is non-optimal and slower than prior work, it clearly demonstrates the viability of using WGSL code embedded in a website as a covert channel receiver.

8 DISCUSSION

Supported Devices. Our research primarily targets recent NVIDIA GPUs, leading to worse results on AMD cards, as we only had very limited access. Despite these architectural differences, WebGPU clearly enables generic cache attacks from browsers. At the time of writing, WebGPU is already integrated into Android's Chrome Canary, though some features are not yet available. Once parity is achieved, the potential for browser-based GPU attacks could significantly increase.

Limitations. We evaluated our proof-of-concept on various operating systems using Chrome and Chromium versions 112-117. Despite identifying functional combinations for all devices, the WebGPU implementation remains inconsistent, as evidenced by our experiments. The same code might succeed in one version and unexpectedly fail in another, potentially due to variations in WebGPU's code compilation beyond user control. We observed notable differences between Linux and Windows (see Table 1). While the exact cause—whether driver, browser, or WebGPU's underlying framework (e.g., Vulkan vs. DirectX)—remains unclear, the fundamental time discrepancy supports the viability of these attacks.

Countermeasures. The attacks shown in this paper are generic and rely on only a few assumptions. Nevertheless, steps can be taken to limit the attack surface. As already suggested in the current WebGPU draft, timers can be made optional, very coarse, or ideally removed altogether [17]. However, as we have shown, as long as coherent memory is available between concurrent threads, it is possible to construct a timer. If, however, the coherency mechanism (in our case, `atomic` operations) were to be changed, such a timer would quickly fail. Of course, this could cause normal workloads to malfunction unless specifically redesigned.

The simplest and most effective solution against a drive-by attack scenario is, in our opinion, to treat GPU access in the browser as a sensitive resource, like microphone or camera access, that requires permission before use. For WebGL and WebGPU, this is not currently the case (Firefox 114, Chrome 115, Chromium 117). This would also prevent malicious parties from stealthily using local computing resources for, e.g., cryptomining.

Disclosure. We have disclosed our results to Mozilla, AMD, NVIDIA and the Chromium team.

9 CONCLUSION

GPUs have become a ubiquitous computation resource and as such require increased security scrutiny. We showed that it is possible to mount powerful GPU cache side-channel attacks directly from within the browser. We demonstrated that our basic attack primitives are generic and automated to the extent that we can run them without manual intervention on a set of 11 desktop GPUs from 5 different generations and 2 vendors, running in the browser through WebGPU. We showed that the massive parallelism of modern GPUs can be leveraged in parallelized eviction set construction algorithms. Our three case studies emphasized the relevance of our work: Our inter-keystroke timing attack, with F_1-scores between 82 % and 98 %, exposes sensitive user input to an attacker. Our set-agnostic end-to-end attack on GPU-based AES encryption leaks full AES keys in 6 min, showing that also cryptographic secrets are exposed to browser-based attackers. Our native-to-browser Prime+Probe covert channel shows that the bandwidth of this channel can reach average transmission rates of up to 10.9 kB/s. Since our attacks require no user interaction, they can be implemented as dangerous drive-by attacks. Thus, we conclude that GPU access should be treated as a similar security and privacy risk as other devices and resources that require explicit user consent.

ACKNOWLEDGMENTS

This research is supported in part by the European Research Council (ERC project FSSec 101076409), and the Austrian Science Fund (FWF project NeRAM I-6054-N). Additional funding was provided by a generous gift from Red Hat. Any opinions or recommendations expressed are those of the authors and do not necessarily reflect the views of the funding parties. We also thank Gregor Heindl for his generous donation of time and hardware.

REFERENCES

[1] NVIDIA, "CUDA C++ Programming Guide," 2023.
[2] Khronos, "OpenCL," 2023. [Online]. Available: https://www.khronos.org/opencl/
[3] J. Wei, Y. Zhang, Z. Zhou, Z. Li, and M. A. A. Faruque, "Leaky DNN: Stealing Deep-Learning Model Secret with GPU Context-Switching Side-Channel," in *DSN*, 2020.
[4] C. Tezcan, "Optimization of advanced encryption standard on graphics processing units," *IEEE Access*, vol. 9, pp. 67 315–67 326, 2021.
[5] T. Yamanouchi, "GPU Gems 3 - AES Encryption and Decryption on the GPU," 2007. [Online]. Available: https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-36-aes-encryption-and-decryption-gpu
[6] J. Ahn, C. Jin, J. Kim, M. Rhu, Y. Fei, D. Kaeli, and J. Kim, "Trident: A hybrid correlation-collision GPU cache timing attack for AES key recovery," in *HPCA*, 2021.
[7] Z. H. Jiang, Y. Fei, and D. Kaeli, "A complete key recovery timing attack on a GPU," in *HPCA*, 2016.
[8] ——, "A novel side-channel timing attack on GPUs," in *Proceedings of the on Great Lakes Symposium on VLSI*, 2017, pp. 167–172.
[9] ——, "Exploiting bank conflict-based side-channel timing leakage of gpus," *ACM TACO*, 2019.
[10] H. Naghibijouybari, K. N. Khasawneh, and N. B. Abu-Ghazaleh, "Constructing and characterizing covert channels on GPGPUs," in *MICRO*, 2017.
[11] H. Naghibijouybari, E. M. Koruyeh, and N. B. Abu-Ghazaleh, "Microarchitectural Attacks in Heterogeneous Systems: A Survey," *ACM Comput. Surv.*, vol. 55, no. 7, pp. 142:1–142:40, 2023.
[12] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, "Rendered Insecure: GPU Side Channel Attacks are Practical," in *CCS*, 2018.
[13] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU," in *S&P*, 2018.
[14] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom, "SGAxe: How SGX fails in practice," 2020.
[15] P. Cronin, X. Gao, H. Wang, and C. Cotton, "An Exploration of ARM System-Level Cache and GPU Side Channels," in *ACSAC*, 2021.
[16] S. B. Dutta, H. Naghibijouybari, A. Gupta, N. B. Abu-Ghazaleh, A. Marquez, and K. J. Barker, "Spy in the GPU-box: Covert and Side Channel Attacks on Multi-GPU Systems," in *ISCA*, 2022.
[17] W3C, "WebGPU - W3C Working Draft - Timing attacks," 2023. [Online]. Available: https://www.w3.org/TR/webgpu/#security-timing
[18] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the Line: Practical Cache Attacks on the MMU." in *NDSS*, 2017.
[19] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript," in *FC*, 2017.
[20] A. van Kesteren, "Safely reviving shared memory," 2020. [Online]. Available: https://hacks.mozilla.org/2020/07/safely-reviving-shared-memory/
[21] Mozilla, "SharedArrayBuffer," 2012. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer
[22] W3C, "WebGPU Security Considerations," 2023. [Online]. Available: https://www.w3.org/TR/webgpu/#security-considerations
[23] NVIDIA, "NVIDIA Tesla v100 GPU architecture," 2017. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf
[24] ——, "Kernel Profiling Guide," 2023.
[25] AMD, "AMD RDNA Whitepaper," 2023. [Online]. Available: https://www.amd.com/system/files/documents/rdna-whitepaper.pdf
[26] K. Group, "WebGL Specification," https://registry.khronos.org/webgl/specs/1.0.3/, 2023.
[27] K. W. W. Group, "WebGL 2.0 Compute," https://registry.khronos.org/webgl/specs/latest/2.0-compute/, 2021.
[28] Mozilla, "WebGPU API," 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebGPU_API
[29] Google, "Chrome ships WebGPU," 2023. [Online]. Available: https://developer.chrome.com/blog/webgpu-release/
[30] W3C, "WebGPU," 2023. [Online]. Available: https://www.w3.org/TR/webgpu
[31] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *S&P*, 2015.
[32] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: the Case of AES," in *CT-RSA*, 2006.
[33] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications," in *CCS*, 2015.
[34] A. Shusterman, A. Agarwal, S. O'Connell, D. Genkin, Y. Oren, and Y. Yarom, "Prime+Probe 1, JavaScript 0: Overcoming browser-based side-channel defenses," in *USENIX Security*, 2021.
[35] S. Wu, J. Yu, M. Yang, and Y. Cao, "Rendering Contention Channel Made Practical in Web Browsers," in *USENIX Security*, 2022.
[36] S. B. Dutta, H. Naghibijouybari, N. Abu-Ghazaleh, A. Marquez, and K. Barker, "Leaky buddies: Cross-component covert channels on integrated cpu-gpu systems," in *ISCA*, 2021.
[37] H. Taneja, J. Kim, J. J. Xu, S. van Schaik, D. Genkin, and Y. Yarom, "Hot Pixels: Frequency, Power, and Temperature Attacks on GPUs and ARM SoCs," in *USENIX Security*, 2023.
[38] P. Vila, B. Köpf, and J. Morales, "Theory and Practice of Finding Eviction Sets," in *S&P*, 2019.
[39] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache Attacks on Mobile Devices," in *USENIX Security*, 2016.
[40] W3C, "WebGPU Shading Language - Terminology and Concepts," 2023. [Online]. Available: https://www.w3.org/TR/WGSL/#uniformity-concepts
[41] S. Jain, I. Baek, S. Wang, and R. Rajkumar, "Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.
[42] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse Engineering Intel Complex Addressing Using Performance Counters," in *RAID*, 2015.
[43] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Systematic reverse engineering of cache slice selection in Intel processors," in *Euromicro Conference on Digital System Design*, 2015.
[44] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser, "Mapping the Intel Last-Level Cache," *Cryptology ePrint Archive, Report 2015/905*, 2015.
[45] X. Mei and X. Chu, "Dissecting GPU memory hierarchy through microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, 2016.
[46] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, "Dissecting the NVidia Turing T4 GPU via microbenchmarking," *arXiv:1903.07486*, 2019.
[47] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA volta GPU architecture via microbenchmarking," *arXiv:1804.06826*, 2018.
[48] M. K. Qureshi, "New attacks and defense for encrypted-address cache," in *ISCA*, 2019.
[49] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede, "Systematic Analysis of Randomization-based Protected Cache Architectures," in *S&P*, 2021.
[50] D. X. Song, D. Wagner, and X. Tian, "Timing Analysis of Keystrokes and Timing Attacks on SSH," in *USENIX Security*, 2001.
[51] D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches," in *USENIX Security*, 2015.
[52] M. Lipp, D. Gruss, M. Schwarz, D. Bidner, C.-m.-t.-n. Maurice, and S. Mangard, "Practical Keystroke Timing Attacks in Sandboxed JavaScript," in *ESORICS*, 2017.
[53] J. Monaco, "SoK: Keylogging Side Channels," in *S&P*, 2018.
[54] J. Bonneau and I. Mironov, "Cache-collision timing attacks against AES," in *CHES*, 2006.
[55] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! A fast, Cross-VM attack on AES," in *RAID*, 2014.
[56] M. Neve and J.-P. Seifert, "Advances on access-driven cache attacks on AES," in *SAC*. Springer, 2007.
[57] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. Alberto Boano, S. Mangard, and K. Römer, "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud," in *NDSS*, 2017.