INVALIDATE+COMPARE: A Timer-Free GPU Cache Attack Primitive

Zhenkai Zhang, *Clemson University;* Kunbei Cai, *University of Central Florida;*
Yanan Guo, *University of Rochester;* Fan Yao, *University of Central Florida;*
Xing Gao, *University of Delaware*

https://www.usenix.org/conference/usenixsecurity24/presentation/zhang-zhenkai

This paper is included in the Proceedings of the
33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the
33rd USENIX Security Symposium
is sponsored by USENIX.

INVALIDATE+COMPARE: A Timer-Free GPU Cache Attack Primitive

Zhenkai Zhang
Clemson University

Kunbei Cai
University of Central Florida

Yanan Guo
University of Rochester

Fan Yao
University of Central Florida

Xing Gao
University of Delaware

Abstract

While extensive research has been conducted on CPU cache side-channel attacks, the landscape of similar studies on modern GPUs remains largely uncharted. In this paper, we investigate potential information leakage threats posed by the caches in GPUs of NVIDIA's latest Ampere and Ada Lovelace generations. We first exploit a GPU cache maintenance instruction to reverse engineer certain key properties of the cache hierarchy in these GPUs, and then we introduce a novel GPU cache side-channel attack primitive named INVALIDATE+COMPARE that is designed to spy on the GPU cache activities of a victim in a timer-free manner. We further showcase the use of this primitive with two case studies. The first one is a website fingerprinting attack that can accurately identify the web pages visited by a user, while the second one uncovers keystroke data entered via a virtual keyboard. To our knowledge, these stand as the first demonstrations of timer-free cache side-channel attacks on GPUs.

1 Introduction

Over the years, dedicated graphics processing units (GPUs) have emerged as essential components in modern computer systems. On one hand, they are utilized to handle high-quality graphics rendering tasks, ensuring smooth frame rates with fine texture details and rich color depths. On the other hand, they provide the processing power required for executing a broad spectrum of compute-intensive applications, such as physical dynamics simulation and deep learning.

Undeniably, NVIDIA has established itself as the dominant player in the GPU market. In recent years, its products of the Ampere and Ada Lovelace generations (e.g., those in the RTX 30-/40-series) have been driving substantial revenue, thanks to their industry-leading performance and the high demand they enjoy across various sectors [13, 31, 44, 66].

However, despite the widespread adoption of these GPUs, their potential security issues, from the hardware perspective, have not been as thoroughly studied as those of CPUs. As an outstanding example, concerns about information leakage via CPU caches have received extensive research attention [2, 9, 10, 17, 25, 30, 47, 48, 51, 56, 69, 73], but similar problems are under-explored with respect to GPU caches. Since many tasks running on a GPU operate on sensitive information, such an oversight may lead to unanticipated security breaches.

Echoing the presumed implications, in this work, we aim to undertake a study on potential information leakage through caches in contemporary NVIDIA GPUs and demonstrate relevant side-channel attacks that compromise user confidentiality. Yet, to achieve this, several challenges specific to these GPUs need to be addressed.

The first challenge lies in the lack of knowledge about such GPU caches. To develop cache side-channel attacks, a level of understanding on certain properties of the cache is necessary (e.g., whether it is inclusive, exclusive, or non-inclusive and how its state can be deterministically manipulated). Although there exist a few reverse-engineering works on GPUs [20, 21, 33, 53, 71], most of them deal with obsolete models and all of them only concentrate on the cache structures without delving into the policies that dictate cache operations. Therefore, we must make efforts to bridge this knowledge gap.

The second one revolves around finding a reliable method to monitor GPU cache activities. Conventionally, CPU cache side-channel attacks exploit the timing discrepancies between cache hits and misses to infer cache access patterns. While the approach is largely successful, noise in timing measurements may degrade its effectiveness [2], which can also be an issue in the context of GPUs. Nevertheless, a more pronounced issue of using timers in GPUs is that unlike CPU timestamp counters, which increment in a frequency-invariant fashion [29], their GPU equivalents are susceptible to frequency shifts. This means that a cache hit at a high performance level may require as many cycles as a cache miss at a low level. For instance, an RTX 3080's frequency dynamically varies between 210MHz and 2100MHz, and at its peak frequency, an L2 cache hit takes about 530 cycles, while at around 550MHz, an L2 miss takes roughly the same amount. Certainly, the problems associated with timing can be bypassed if a timer-free attack primitive, similar to those for CPUs [2, 23, 74, 76], is available.

The third challenge stems from the manner in which GPUs handle the execution of concurrently running GPU programs. Unlike multi-core CPUs that allow multiple processes to run simultaneously, GPUs orchestrate the concurrent execution of programs using a time-sharing mechanism. The duration of a GPU time slice is long enough to accommodate a number of memory operations, and we observe that nearly all of the GPU cache sets are accessed within each time slice in common GPU workloads. As a result, merely deducing if a GPU cache set has been accessed or not is insufficient for mounting GPU cache side-channel attacks. We need an approach to extracting more fine-grained information from the GPU cache.

Taking advantage of a GPU cache maintenance instruction named `discard`, we have successfully addressed all of these challenges and formulated a novel, timer-free GPU cache side-channel attack primitive dubbed as INVALIDATE+COMPARE. Using this primitive, we demonstrate two side-channel attacks targeting the latest NVIDIA Ampere and Ada Lovelace GPUs. The first one is a website fingerprinting attack, similar to those described in [27, 38]. However, unlike these prior studies that exploited software flaws (already fixed) in GPU drivers, our attack is built on vulnerabilities of hardware cache, which is more difficult to thwart. The second attack we showcase is to infer specific keystrokes made by a user on a virtual keyboard. While it has been shown that typing on such a keyboard can be recovered via monitoring cache lines of graphics libraries on the CPU side [69], our attack reveals that such sensitive information can also be extracted from GPU cache. To our knowledge, these two case studies stand as the first timer-free cache side-channel attacks on GPUs.

The main contributions of this paper are:

- We leverage the incoherence among GPU L1 caches and the undocumented semantics of the `discard` instruction to form a new method for reverse engineering GPU cache properties. We show that the approach allows us to unveil certain key cache characteristics in the latest NVIDIA GPUs for the first time.
- We introduce INVALIDATE+COMPARE, which is a new GPU cache side-channel attack primitive and also the first of its kind to be timer-free. To put the primitive into action, we address challenges like how to synchronize cache monitoring steps with GPU context switching and how to measure fine-grained information regarding the degree of contention within a GPU cache set.
- We use the formulated attack primitive to conduct two case studies. In the first, we illustrate a highly accurate website fingerprinting attack that maintains its effectiveness over time, and in the second, we present a keystroke extraction attack against the OS's bundled virtual keyboard, showing its potential to steal a user's login password.

Responsible disclosure: We have disclosed our findings to NVIDIA, who has acknowledged our work.

2 Background

2.1 GPU Architecture

GPUs have transformed from specialized graphics rendering devices into highly programmable accelerators capable of executing a wide range of parallel workloads. The primary compute units in a modern GPU are called streaming multiprocessors (SMs), each of which comprises an array of simple cores. SMs are designed to execute groups of threads, referred to as warps, in a single-instruction multiple-thread (SIMT) fashion. In terms of NVIDIA GPUs, a warp consists of 32 threads. When multiple warps execute on an SM, they are scheduled by a hardware unit in the SM. In general, there are tens of SMs in a high-end GPU (e.g., NVIDIA RTX 3080 houses 68 SMs).

A GPU uses its on-board memory to hold data slated for processing. Such on-board GPU memory typically comprises several DRAM chips that are of special types tailored for high bandwidth (e.g., GDDR6). Each DRAM chip is connected with the GPU via a dedicated memory controller. Note that the GPU memory operates independently from the main memory on the CPU side and is managed in its own manner. Transferring data between the main memory and GPU memory is facilitated by the PCIe bus.

GPU memory access latency is very high (in fact much higher than that of main memory). To help counteract such significant latency, caches are used. A modern GPU usually has a two-level cache hierarchy. SMs have private L1 caches, a portion of which can be repurposed as scratchpad memory. All SMs share an L2 that serves as the last-level cache (LLC). In NVIDIA GPUs, the cache line size is 128B [22]. Similar to the CPU counterpart, the LLC of a GPU is also set-associative and physically tagged.

2.2 GPU Programming

GPUs can always be programmed using graphics rendering APIs (e.g., OpenGL [7] and Vulkan [6]). These APIs provide the framework for developing shaders, which are specialized GPU programs that dictate how graphics are generated and drawn on the screen. However, when using GPUs for broader computational tasks, a more general-purpose programming language is needed.

As a standard practice, NVIDIA GPUs are programmed with CUDA for general-purpose parallel computing [41]. In CUDA, GPU computational tasks are defined within functions known as kernels. When a kernel gets launched on a GPU, it is instantiated as a grid of thread blocks. Each thread block contains a number of threads and is assigned to an SM by the CUDA runtime for execution. The count of thread blocks and the threads in each block need to be specified upon launch.

While CUDA provides a high-level GPU programming paradigm, a low-level assembly-like language named Parallel

Table 1: GPU models tested in Section 3.

Grade	GPU Model	Generation	SM Cnt.	Memory Sz.	L2 Sz.†	Release
Consumer	RTX 3060	Ampere	28	12GB	3MB	Jan 2021
	RTX 3080	Ampere	68	10GB	5MB	Sep 2020
	RTX 4060	Ada Lovelace	24	8GB	24MB	May 2023
Server	A2	Ampere	10	16GB	2MB	Nov 2021
	A10	Ampere	72	24GB	6MB	Apr 2021

† Except for RTX 3080 whose L2 cache size is officially provided by NVIDIA [43], the data for the other models are sourced from TechPowerUp [62–65].

Thread Execution (PTX) can also be used to program NVIDIA GPUs [45]. PTX exposes more details about GPU internals and offers finer control over hardware resources. The `asm()` statements can be leveraged to embed arbitrary PTX code within a CUDA kernel function. Notice that PTX is not the actual assembly language used by NVIDIA GPUs. Instead, it plays the role of an intermediate representation, acting as an abstract ISA that is compatible across multiple generations of NVIDIA GPUs.

2.3 GPU Context

Regardless of the programming language or API being used, an instance of a GPU program in execution is referred to as a GPU context that is analogous to a CPU process. GPUs employ a time-sharing mechanism to concurrently run multiple GPU contexts. Basically, inside a GPU, there is a hardware scheduler that multiplexes the execution of GPU contexts by allocating each context a time slice. Within its time slice, a context can access available resources on the GPU.

Although an NVIDIA GPU does not support simultaneous execution of multiple contexts, a feature of the CUDA runtime named multi-process service (MPS), when enabled, can merge several running CUDA programs into a single GPU context for better resource utilization. By default, MPS is disabled and users typically do not turn it on. Therefore, in the usual setup, GPU contexts of running CUDA programs are time-sliced. Moreover, MPS is specific to CUDA and does not combine graphics rendering GPU contexts.

3 GPU Cache Characteristics

In this section, we explore the cache hierarchy in NVIDIA GPUs and uncover the key properties that pave the way for the construction of our new attack primitive. Both the exploration and the primitive hinge on the use of a special GPU cache maintenance instruction, which we shall present first in the following discussion.

3.1 The `discard` PTX Instruction

Undoubtedly, the security community has been familiar with the `clflush` instruction in x86 CPUs and the various cache side-channel attacks it enables [8, 10, 69, 73]. Essentially, this unprivileged instruction can be used to flush a cache line from all the cache levels in an x86 processor, and if the cache line is dirty, it also ensures that the contents in the cache line are written back to the main memory [16].

In NVIDIA Turing and earlier GPUs, no instruction similar to `clflush` is available. On the other hand, starting with the Ampere microarchitecture, NVIDIA has introduced a new PTX instruction named `discard` to facilitate removing cache lines. The syntax[1] of this instruction is as follows:

```
discard.L2   [addr], 128;
```

where the *addr* operand specifies a GPU memory address that needs to align on a 128-byte boundary. According to the official documentation [45], the semantics of the `discard` instruction is to invalidate any data in the address range [*addr*, *addr* + 128) cached in L2 without writing the data back to GPU memory.

Although it is not specifically mentioned in [45], given the fact that the cache line size is 128B in NVIDIA GPUs [22], we can deduce that `discard` operates on a single cache line. Notice that, similar to the CPU side, the GPU also employs paging-based virtual memory, and the address specified by the *addr* operand is a virtual one. Naturally, GPUs also require virtual-to-physical address translation, the details of which have been revealed in [78].

If we compare the semantics of the CPU's `clflush` and the GPU's `discard` instructions as described in their respective official documentations, we can identify two main differences. First, `clflush` removes the target cache line from the entire cache hierarchy, while `discard` is only mentioned to remove the cache line from L2, which is the LLC in a GPU. Second, it is important to highlight that `clflush` writes modified data in the removed cache line back to memory, whereas `discard` does not write any data back to memory, even if the target cache line is dirty. From this aspect, `discard` bears resemblance to the privileged `invd` instruction in x86.

3.2 Inclusion Policy

NVIDIA has never disclosed any information on whether the L2 cache in its GPUs is inclusive, exclusive, or non-inclusive. Because the load instruction `ld` brings data into both L1 and L2 caches by default [45], L2 must not be exclusive. Interestingly, it is mentioned in [45] that L1 caches are not coherent. We know that the main reason for having an inclusive LLC is to simplify the design of cache coherence protocols, and thus we speculate that L2 in NVIDIA GPUs is non-inclusive. To shed light on this matter and also understand more on the `discard`'s effects, we perform the following experiment illustrated in Figure 1.

Suppose there is a data block `B` in GPU memory whose initial value is 0. To begin with, we arbitrarily choose two SMs, `X` and `Y`, to load `B` into L2 as well as their own L1 caches.

[1] The exact syntax of this instruction is `discard{.global}.`*`level`* `[`*`addr`*`], `*`size`*`;`. Nevertheless, `.global` can be omitted, *`level`* can only be `L2`, and *`size`* can only be 128.

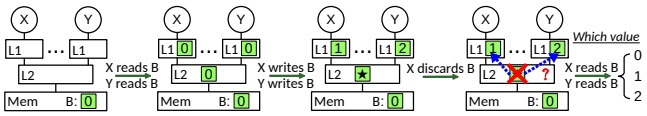


Figure 1: Experiment to further understand `discard` and determine if L2 is inclusive or non-inclusive. (★ is used to represent an indeterminate value, and Section 3.3 explains why it can happen.)

Next, we let `X` overwrite `B` with the value 1 and let `Y` overwrite `B` with the value 2. Since coherence is not maintained among L1 caches of SMs [45], the updates made by `X` and `Y` will be simultaneously kept in the corresponding L1 caches without invalidating each other. We then let `X` execute the `discard` instruction on `B` to remove it from L2. After that, we have `X` and `Y` read `B` again to examine which value each of them retrieves, respectively.

We conduct this experiment on multiple GPUs, which are listed in Table 1, including three very popular consumer-grade ones (e.g., RTX 3060) and two server-grade ones (e.g., A10). We observe a consistent result on all the GPUs, which is: `X` retrieves 0 and `Y` retrieves 2. From this result, we can draw the following conclusions. ❶ L2 in these GPUs is non-inclusive; otherwise, the removal of `B` from L2 should have enforced the invalidation of the corresponding cache line in `Y`'s L1. ❷ `discard` does not remove the target cache line from the entire cache hierarchy. Instead, it removes the cache line from L2 and also from the L1 cache of the SM issuing the instruction, which is not specified in the documented semantics. ❸ When an L1 cache miss occurs, the needed memory block is fetched from lower levels of the GPU memory hierarchy rather than the sibling L1 caches; if this were not the case, `X` should have obtained 2.

Notice that there is no mechanism in CUDA to synchronize concurrent execution on different SMs. To ensure that one SM will not perform an operation (e.g., `X` discards `B`) until the other SM has finished its required predecessor (e.g., `Y` modifies `B`), we insert a sufficiently large delay before the operation to enforce the intended ordering.

3.3 Write Policy

In the aforementioned experiment, it is clear that updates do not write through the whole cache hierarchy to the GPU memory (otherwise, `X` would not have retrieved 0). Nevertheless, we do not know how writes are handled between L1 and L2. To gain insight into this particular write policy, we conduct the procedure shown in Figure 2.

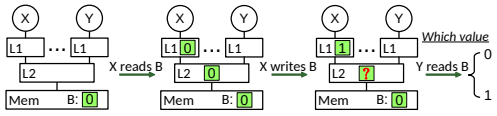


Figure 2: Experiment to check if L1 is write-back or write-through.

Similar to the study performed above, we utilize two SMs and a memory block with an initial value of 0. Instead of having both SMs read and modify `B`, here we only use `X` to do so. After `X`'s update, we let `Y` read `B` to check its value. (An auxiliary experiment in Appendix A also confirms that `B` has never been evicted from `X`'s L1 cache.) We find that `Y` always retrieves 1 regardless of the GPU and selected SMs.

This behavior is not surprising if considered from the CPU's perspective. However, as discussed earlier, a GPU's L1 caches are incoherent, and its L1 cache misses are serviced by L2 or GPU memory rather than by any sibling L1 caches. Therefore, such behavior manifests the fact that when `B` is written in L1, it is also updated in L2. In other words, the write policy employed by GPU L1 caches is *write-through*. (The whole write-through process should be atomic.) Surely, due to this design, if multiple SMs write to the same address concurrently, the final value in L2 depends on the order of these writes.

From our previous reasoning, it is not hard to deduce that L2 is a write-back cache. As a simple verification, after `X` writes 1 to `B`, we let `Y` sequentially access a very large array, and then have `X` use `discard` to invalidate `B`. When `X` reloads `B` from GPU memory, it obtains 1. If dirty cache lines in L2 were not written back to memory, `X` should have retrieved 0.

3.4 Write Allocation Policy

It is said that write-back caches often use the write allocate scheme (namely, when a write operation misses in the cache, the corresponding block is fetched and put in the cache for being updated), while write-through caches often use the no-write allocate scheme (namely, when a write operation misses in the cache, data is directly written to the lower level without being allocated space in the cache) [50]. Following the steps shown in Figure 3 and Figure 4, we can easily examine the write allocation policy used in GPU L1 and L2 caches, respectively.

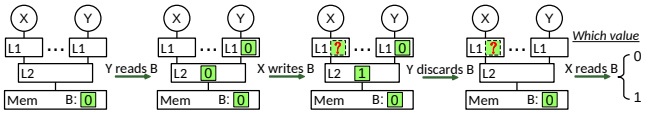


Figure 3: Experiment to examine the write allocation policy of L1.

As illustrated in Figure 3, `Y` loads `B` first, and then `X` writes 1 to `B`. In this case, `X` experiences an L1 cache miss when writing `B`. Subsequently, `Y` discards `B`, after which, `X` reads `B` to check its value. We find that the value read out by `X` is always 1. This observation indicates that L1, although write-through, employs the write allocate scheme; because if the no-write allocate one were used, `X` should have read `B` from GPU memory, which still has the initial value 0.

Similarly, as illustrated in Figure 4, we can engineer a situation where `X` will experience an L2 cache miss when it

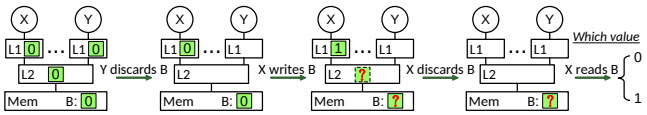


Figure 4: Experiment to examine the write allocation policy of L2.

modifies B. Despite a hit in L1, L2 always receives X's write operation due to the write-through property of L1. Thus, if L2 employs the no-write allocate policy, B in GPU memory will be updated; conversely, if L2 uses the write allocate scheme, B in memory will preserve its original value. To check if the value of B in GPU memory is updated, X can discard B first and then read it. We discover that L2 caches in all the tested GPUs use the write allocate scheme.

Exploiting timing differences between L1/L2 cache hits and misses, we can reach the same conclusions. Even though timing can be used for examining both inclusion and write allocation policies, we need to emphasize that it may not be easily used to infer the write policy.

3.5 L1 Cache Auto-Flushing

As outlined in Section 2.3, by default, GPU contexts are scheduled to take turns executing on the GPU. An interesting behavior we have noticed is that context switching automatically flushes all the L1 caches. The experiment shown in Figure 5 demonstrates this auto-flushing behavior.

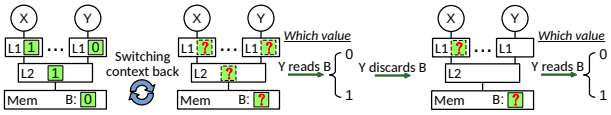


Figure 5: Experiment to demonstrate L1 cache auto-flushing.

At the outset, we prepare an incoherent cache hierarchy state in which B holds different values in the L1 caches of X and Y. This can be easily achieved by having Y load B first and then X write 1 to B. It is apparent that if B persistently resides in Y's L1 cache, Y will never see the value of B as 1. To induce GPU context switches, we create and execute a dummy CUDA program that has a PTX branch instruction infinitely looping on itself. Accordingly, the chance that L1 caches are evicted by other memory accesses is minimized. After the context is switched back, we find that Y, regardless of which SM it is, always sees the value of B as 1. This observation indicates that GPU context switching triggers L1 cache flushing.

To verify that GPU context switching does not flush L2 and write dirty cache lines back to memory, we subsequently let Y discard B, followed by a load operation on B. In this case, we find that Y retrieves 0, i.e., the original value of B. This implies the fact that L2 is not flushed; otherwise, Y should have consistently seen the same value no matter if discarding B was performed or not.

3.6 Associativity and Replacement

Due to its auto-flushing behavior, L1 will not pose challenges to cache side-channel attacks on GPUs, even though the LLC is non-inclusive. Consequently, we only focus on the L2 cache (i.e., the LLC) here for the sake of brevity.

Given a GPU, we begin by determining the associativity of its L2 cache. For this purpose, we derive a list of addresses $\{A_0, A_1, \cdots\}$ that are mapped to an identical L2 cache set (see Section 4.2 for the detailed procedure), and access them one-by-one to find how many of them the cache set can hold. For each GPU tested, we observe a peculiar behavior. ❶ If we access the addresses using only st, at most 7 of them can be kept in a cache set. ❷ If we access the addresses using only ld, up to 16 of them can be finally kept in a set. ❸ If we access the addresses using both ld and st, a cache set can still hold 16 blocks, but under certain access patterns, up to 8 can be brought in by st. Thus, we deduce that the associativity of L2 is 16 in NVIDIA GPUs, but a cache set's 16 ways are not used proportionately by loads and stores. We conjecture that NVIDIA aims to maintain a dirty cache line ratio $\leq$50% in its L2 design.

We further examine the dynamics of replacement within L2. When only st is used, we find that the replacement among the 7 dirty cache lines simply adheres to the LRU policy. Intriguingly, when only ld is used, replacements start appearing regularly as a cache set is being filled over 9 ways, but after all the 16 ways are occupied, the replacement behavior conforms to LRU (refer to Appendix B for more details).

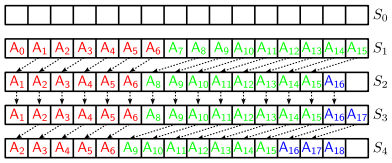


Figure 6: Priming an L2 cache set (writing $A_0, \cdots, A_6$ followed by reading $A_7, \cdots, A_{15}$) and then accessing other addresses.

Our attack primitive needs to prime cache sets while making certain cache lines in them dirty. Apparently, due to the limit on the number of dirty cache lines in a set, we cannot use st alone to achieve priming. In our case, we look to verify whether 7 stores followed by 9 loads, or 9 loads followed by 7 stores, can effectively populate a cache set, and we are also interested in the replacement behavior in response to new accesses after a cache set is populated.

Although filling an empty cache set with only ld requires more than 16 accesses (see Appendix B), we observe that a combination of 7 stores and 9 loads, adding up to 16 accesses, can effectively occupy an unused set. (The state S_1 in Figure 6 illustrates the case where 7 stores followed by 9 loads.) This implies that L2 may not manage dirty and clean cache lines in the same way.

When accessing new addresses after a cache set is filled

with 7 stores and 9 loads, we find an interesting phenomenon that the eviction patterns do not align with any known replacement policy, which is depicted in Figure 6. We can see that accessing A_{16} forces out not only the least recently used A_0 but also A_7. While A_7 has been accessed more recently than $A_1, \cdots, A_6$, it is the least recently read one via `ld`. Our hypothesis is that when the least recently used cache line is dirty and gets evicted, the least recently read one will also be evicted. The state transition from S_3 to S_4 in Figure 6 lends support to this hypothesis. Note that the opposite is not true. If the least recently used cache line is clean and becomes evicted, the least recently written one will not be evicted.

Despite the L2's unusual replacement behavior, we find that after cache lines in a primed cache set are evicted, the state can be reestablished by performing 7 stores and 9 loads again. This determinism suffices for the scope of this study, and the detailed analysis of its replacement policy is reserved for future research. Following standard nomenclature, the addresses corresponding to the 7 stores and 9 loads are said to form an eviction set, and these addresses are referred to as congruent.

4 INVALIDATE+COMPARE

Based on the learned properties of the GPU cache hierarchy, we formulate the INVALIDATE+COMPARE attack primitive for spying on a victim's GPU cache activity in a timer-free manner. Given an L2 cache set along with an eviction set $\{A_0, \cdots, A_6, A_7, \cdots, A_{15}\}$, the initial primitive consists of the following five steps:

Step 1: Write new values to addresses $A_0, \cdots, A_6$.
Step 2: Read the values at addresses $A_7, \cdots, A_{15}$.
Step 3: Wait for the context to be switched away and back.
Step 4: Execute the `discard` instruction on A_0.
Step 5: Read the current value at A_0 for comparison.
- If it is the original value, the victim did not access the cache set during its running period.
- If it is the newly written value, the victim accessed the cache set during its running period.

The rationale behind the primitive is that if the victim fetched any data into the cache set, the dirty cache line corresponding to A_0 would have been evicted before we attempted to invalidate it and hence the value within it would have been committed to the memory; otherwise, no dirty cache lines were evicted and when the one associated with A_0 became invalidated, the value within it would be lost in accordance with the `discard`'s semantics. Therefore, the value at A_0 after performing the invalidation can accurately reveal if the cache set is accessed by the victim. (The reason for the initial two steps has been discussed above in Section 3.6.)

In this primitive, after Step 1 and 2 prepare the cache set state, in Step 3, the attacker shall wait for the victim to carry out some GPU computation. In other words, the attacker needs to wait for the victim's GPU context to get scheduled and later the attacker's GPU context will regain control again. Hence, this step in essence is the attacker awaiting her own GPU context to be switched out and then back in.

An SM is capable of simultaneously spying on multiple cache sets, each of which is monitored by a separate thread. For NVIDIA GPUs, a warp consists of 32 threads. When there are multiple warps executing on an SM, they are scheduled by a hardware scheduler unit. Considering the availability of tens of SMs in a GPU, we are able to monitor hundreds of cache sets in parallel.

It is important to highlight that the primitive *in its current form* is intended only for illustrative purposes to aid in comprehension, and may not have practical utility in real-world scenarios. This is because we notice that all (or most of) the L2 sets appear to be accessed after a GPU context switch. To make the INVALIDATE+COMPARE primitive useful, we need to refine it, which will be described later in this section.

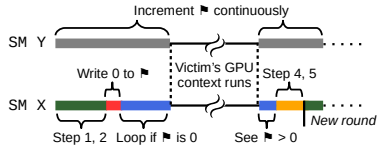
4.1 GPU Context Switch Detection

An important question we have yet to address pertains to the realization of Step 3. More precisely, how can we find out that the execution of the GPU context has been preempted and then resumed? Here we propose two approaches, and both of them are effective for Ampere GPUs, but only the second one is suitable for GPUs in the Ada Lovelace generation.

In the first approach, we slightly relax the timer-free claim by allowing the use of `%clock64` that is the GPU's timestamp counter. (The measurement phase consistently remains timer-free.) Our approach, as depicted in Figure 7a, keeps cycling through a tight loop until an iteration takes more time than a specified threshold `T`. This is indicative of an event where the GPU context has been switched away and resumed in that iteration, since the looping time is very stable in the absence of context switching. Furthermore, `delta` can even help us capture useful temporal information regarding competition under the GPU scheduling.

```
1 prev = 0;
2 start = clock64();
3 do {
4   delta = clock64() - start;
5   if (delta - prev > T)
6     break;
7   prev = delta;
8 } while (1);
```

(a) First approach.



(b) Second approach.

Figure 7: How to detect if the context is switched away and back.

While this method is effective on Ampere GPUs, it does not function as intended on Ada Lovelace GPUs. We discover that, unlike the counter in Ampere GPUs, `%clock64` in Ada Lovelace ones (e.g., RTX 4060) does not represent an actual

clock; instead, it acts as a logical clock specific to each GPU context. Consequently, during a context switch, `%clock64` of one context does not increase to account for the execution time of other GPU contexts.

The second approach, which does not rely on the timestamp counter, is built on the L1 auto-flushing behavior and some other special properties of the GPU cache hierarchy. In this approach, aside from the SMs tasked with monitoring cache sets (dubbed as spy SMs), an additional SM is designated to continuously increment a flag variable. This flag variable should be mapped to a non-monitored set to prevent interference with the monitoring process. After a spy SM has finished the first two steps, it writes 0 to the flag and then repeatedly checks it until the value is no longer zero. Given the L1's incoherent and write allocate nature, copies of the flag variable with two different values reside in the L1 caches of the spy SMs and the SM dedicated to incrementing the flag variable. Because of the L1's write-through nature, each increment to the flag results in an immediate update in L2. It is not hard to see that the spy SMs cannot observe the updated flag in L2 until the copies residing in their own L1 caches are flushed due to context switching.

The second approach is visualized in Figure 7b, where `X` is a spy SM and `Y` is the SM responsible for incrementing the flag. Note that the effectiveness of the approach is not influenced by whether or not the flag is evicted from L2 during the execution of the victim's GPU context. Similar to the `delta` value in the first approach, the flag value here can also be used for capturing temporal information. We shall underscore that the second approach works well on GPUs in both Ampere and Ada Lovelace generations.

4.2 Eviction Set Construction

The problem of finding congruent addresses to form eviction sets for CPU caches has been studied extensively [2, 30, 59, 68]. In terms of our study, we adapt the commonly used method in [30] by replacing its timing-based eviction checking with the function shown in Algorithm 1. The principle of this timer-free checking is essentially the same as that of the primitive, namely, the written value will not survive unless the cache line has been evicted before the invalidation.

Notice that, due to the limitation on the number of dirty cache lines in an L2 cache set (as discussed in Section 3.6), we need to run the method multiple times with respect to different GPU memory chunks to accumulate sufficient congruent addresses. While this is just a minor issue, a trickier problem we encounter is that a large amount of online noise (e.g., introduced by time-sharing) may cause the finding process to fail. Certainly, if it were possible to derive eviction sets offline only once and subsequently use them directly for online attacks, these inconveniences could be eliminated. We confirm that such a "one and done" approach is feasible.

First of all, unlike CPUs where the memory configuration

Algorithm 1: Timer-free eviction checking function.

```
inputs  : An address a and a set of addresses S.
output  : If a can be evicted by S, true; otherwise, false.
function check_eviction(a,S):
    write 0xdeadbeef to a;
    for each address e in S do
        write an arbitrary value (e.g., 0) to e;
    execute the discard instruction on a;
    read the current value v at a;
    if v is equal to 0xdeadbeef then
        return true;
    else
        return false;
```

can be altered, the memory setup in a GPU is not only static but also identical for all instances of the same model. In other words, any given physical address will be mapped into the same L2 cache set on two GPUs of the same model (e.g., an ASUS RTX 4060 and a PNY RTX 4060). Second, we observe that NVIDIA drivers follow a pattern of allocating physically contiguous GPU page frames and maintain a consistent starting address for memory allocation; namely, there is minimal randomization involved in this allocation process. Therefore, we can construct eviction sets against a GPU offline and use them online.

We should emphasize that while the driver begins memory allocation at low physical addresses, we construct eviction sets using memory chunks at a relatively high address (e.g., `0x20000000` in our case studies). This strategy reduces the likelihood of other GPU contexts occupying our required page frames. At the onset of an online attack, by trying different sizes of padding prior to memory allocation and testing if the prepared eviction sets work, we can identify the right location. It is worth mentioning that each padding size is a multiple of the GPU page size, which is 2MB. This large stride allows the search process to be completed swiftly.

We have also tried to derive the mapping function from the constructed eviction sets for each GPU we experimented with. Unfortunately, we have not been successful in this endeavor. We observe that the number of L2 cache sets in all the tested GPUs is not a power of 2, and they cannot be addressed using a group of XOR functions. We leave the full reverse engineering of the mapping functions for our future work.

4.3 Primitive Revision

We have mentioned that the attack primitive requires revision because it lacks practical effectiveness in its present form. Even though it is theoretically sound, we identify two primary issues that prevent the primitive from yielding useful information. The first issue is the non-negligible spatial overhead introduced by GPU context switching. For example, on an RTX 3060, we observe that a number of L2 cache sets appear to have been accessed upon switching back, in spite of the

fact that the other context belongs to a minimal GPU program running merely a dummy infinite loop. Second, contrary to some prior cache side-channel attacks on CPUs, where the attacker can game the scheduler of the OS or hypervisor [10, 77], we have not found a viable method for exerting the same influence over the GPU scheduler. As shown in Figure 8, each time slice allocated to a GPU context is not short, making it very likely that the victim accesses most of, if not all, the L2 cache sets during this period. Hence, providing information solely on whether a cache set is accessed or not barely offers any useful insight into the cache activities of the victim.

To address this problem, we propose a revision to the last two steps of the primitive. Instead of just checking whether a cache set has been accessed, our revised primitive aims to measure the degree of contention in a cache set. Note that one key advantage of our timer-free primitive over the traditional PRIME+PROBE is its ability to easily pinpoint which of the first few cache lines in a set have been evicted. Capitalizing on this feature, we modify the last two steps of the original method to reformulate the primitive as follows:

Step 4: Execute the `discard` instruction on $A_0, \cdots, A_6$.
Step 5: Read the current value at $A_0, \cdots, A_6$ for comparison.
- If A_0 is not evicted (i.e., the old value), it is 0.
- If A_0 is evicted and A_1 is not, it is 1.

  ⋮
- If A_5 is evicted and A_6 is not, it is 6.
- If A_6 is evicted, it is 7.

Essentially, rather than only carrying out the invalidation and comparison operations on A_0, we perform the invalidation operation across all seven addresses, $A_0, \cdots, A_6$, and identify the first one that is not evicted through comparison. We then use the position of the found address in the range to encode the intensity of contention in the corresponding cache set. For instance, if A_0 is not evicted, the intensity is encoded as 0 denoting the lowest, and if every one is evicted, the intensity is encoded as 7 denoting the highest.

Note that for heavy workloads, the contention intensity may always peak at its maximum of 7, which certainly does not convey much information. To enable capturing higher levels of contention in such cases, we can swap the order of Step 1 and Step 2, namely, we first read from $A_7, \cdots, A_{15}$ and then write to $A_0, \cdots, A_6$. In this variant, intensity 0 will be like an aggregate of all the original values from 0 to 7, while intensity 1 can be treated as an equivalent to a hypothetical previous intensity of 8, and so forth.

4.4 Execution Time v.s. Allocated Time Slice

Considering GPU context scheduling, it is crucial to examine whether the execution of the primitive can potentially exceed the allocated time slice before its completion. (If so, the effectiveness of attacks may be affected.) To this end, we compare the maximum duration needed to perform the primitive with the minimum time slice allocated to a GPU context.

As illustrated in Figure 7b, when the attacker's context is switched back, it first executes Step 4 and 5, followed by Step 1 through 3. Since Step 3 is just waiting to be scheduled away, the pertinent execution time is the cumulative time spent on Step 4, 5, 1, and 2. The worst-case scenario for this time occurs when all addresses of the eviction set are absent from the L2 cache, particularly in the revised primitive where all 7 stores need to be invalidated. Figure 8 shows the distributions of 10,000 execution times for the primitive on an RTX 3080. These times are measured in the worst-case scenario when monitoring 32, 256, and 2048 cache sets with 1, 8, and 64 SMs (i.e., each SM hosts one warp), respectively. We can find that all the times are less than 12,000 clock cycles, which is about 6μs. Since the steps of the attack primitive are executed immediately upon being rescheduled, as long as the allocated time slice exceeds 6μs, there is no concern about the primitive failing to complete.

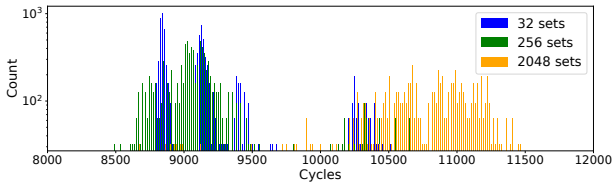
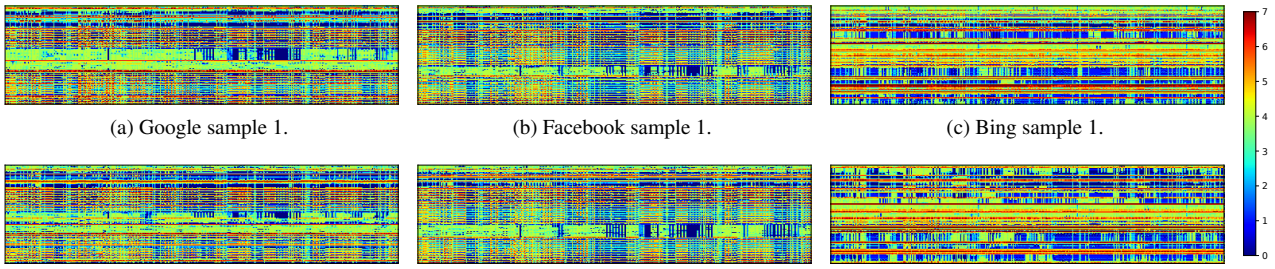


Figure 8: Distributions of primitive execution times on an RTX 3080 GPU. The y-axis uses a logarithmic scale.

Regarding the GPU context scheduler, we have not found any documentation providing information about its time slice length. Given the inevitable presence of scheduling overhead, we surmise that the time slice should not be less than 100μs. For an empirical validation, we run different GPU workloads and measure the allocated time slices in an approximate way (outlined in Appendix C). The analysis of the measured time slices reveals that only around 0.04% of them fall below 1ms, and among these, the shortest length observed is about 172μs. Therefore, we believe that the execution of the primitive can always complete successfully in each round.

5 Case Studies

We present two case studies to demonstrate the effectiveness of our INVALIDATE+COMPARE attack primitive. (Note that the reference to the attack primitive here corresponds to the revised version described in Section 4.3.) While the primitive is applicable to both consumer- and server-grade GPUs, the targets in these case studies are desktop applications, indicating that server-grade GPUs are not relevant to our scenarios. Accordingly, we perform the case studies using the consumer-grade GPUs listed in Table 1, namely the GeForce RTX 3060, 3080, and 4060.



(a) Google sample 1. (b) Facebook sample 1. (c) Bing sample 1.

(d) Google sample 2. (e) Facebook sample 2. (f) Bing sample 2.

Figure 9: Memorygrams corresponding to three commonly visited websites (the x-axis represents 512 cache sets of the RTX 4060 GPU, while the y-axis corresponds to time that progresses from top to bottom).

5.1 Threat Model

An attacker aims to extract certain sensitive information from a victim through their routine use of a personal computer. The victim's computer is equipped with an up-to-date NVIDIA GPU (that is in the Ampere or Ada Lovelace generation), where the latest driver and CUDA, with all disclosed vulnerabilities fixed, have been installed.[2] No special settings for the driver or CUDA are required (e.g., MPS may be turned on/off and the performance mode may be set as auto/fixed).

Similar to many other works [11, 19, 27, 38, 79], we assume that the attacker can run a piece of native code on the victim's computer with user-level privileges. Under this assumption, it is evident that the attacker has no problem retrieving any world-readable version information of software (e.g., the OS or web browser). While this native code setting may seem less consequential, we emphasize that its purpose is for showcasing the application of our primitive. A broader attack scenario will be discussed in Section 5.5.

We focus on two types of sensitive information here, which are website visits and virtual keystrokes. In the case study on stealing information about website visits, commonly referred to as website fingerprinting, we assume that the victim uses a modern web browser with default settings to regularly visit popular websites. The attacker can use the same browser to profile many possible ones to build a predictive model.

In the case study on retrieving keystrokes, we assume that the victim inputs data by tapping the visual keys of a virtual keyboard via a touchscreen or stylus. The virtual keyboard used by the victim is the default one of the corresponding OS. Once again, the attacker is assumed to be able to use the same virtual keyboard to build a predictive model.

5.2 Eviction Set Preparation

As a preparation step, we construct the eviction sets for the L2 cache of each GPU model offline. (See Section 4.2 for the reason of this process.) For each GPU model, we use a chunk of GPU memory, twice the size of the L2 cache, starting at the physical address `0x20000000` to identify the eviction sets. To demonstrate that the manufacturer has no bearing on this process, we tested two RTX 3080 GPUs from different brands (Founders Edition and Gigabyte) and two RTX 4060 GPUs, also from different vendors (MSI and PNY). Our results affirm that, given the same GPU model, the eviction sets remain consistent across vendors.

Note that, in terms of RTX 3060, we have identified 1152 distinct eviction sets for its L2 cache. This equates to 2.25MB (i.e., $1152 \times 16 \times 128\text{B} = 2.25\text{MB}$), not the size of 3MB as reported by TechPowerUp [64]. In the absence of NVIDIA's official documentation, it is unclear which size is the correct one. Nevertheless, this discrepancy does not impact the case studies we conduct. With respect to other models, the L2 sizes match the reported ones (e.g., we find 2560 eviction sets for RTX 3080's L2, which is exactly 5MB).

5.3 Website Fingerprinting

As our first case study, we demonstrate how to identify the web pages browsed by a user, essentially mounting a website fingerprinting attack. Information regarding website visits is typically viewed as significant to privacy, given its potential to reveal a person's sensitive aspects (e.g., political affiliations, health conditions, and special hobbies). Several prior studies have exploited vulnerabilities in NVIDIA drivers to achieve this objective [27, 38], but as the corresponding software flaws have been addressed, their approaches are no longer effective.

5.3.1 Investigation

Over time, a variety of website fingerprinting techniques have been proposed, like those in [1, 12, 14, 19, 47, 49, 52, 56, 67]. Among the existing techniques, several exploit information leakage over cache hierarchies. Ours bears a resemblance to these works (e.g., that of Oren *et al.* [47]), yet with differences. While their methods rely on a (high-resolution) timer to use primitives such as PRIME+PROBE to capture the CPU cache

[2] At the time of this writing, the latest GPU driver version is 535.113.01 and the latest CUDA version is 12.2.

behavior, we use our timer-free attack primitive to measure GPU cache contentions.

The potential to leverage the monitored GPU cache behavior for website fingerprinting arises from the fact that modern web browsers, like Chrome and Firefox, enlist the GPU's assistance for rendering web pages. In effect, when the GPU helps render a web page, the majority of its computational tasks are contingent upon the contents and design of that web page. For example, the rasterization process that converts web page elements (e.g., text, images, and vector graphics) into pixels happens entirely on the GPU, and the compositing process that assembles the separately rasterized layers of a web page into the final image for display is also performed by the GPU. Their workloads depend significantly on the contents of the web page being rendered and how it is designed.

Our website fingerprinting technique leverages contentions in GPU L2 cache sets. We find that during the rendering of a web page, nearly every cache set will be accessed by the browser's GPU context within each of its allocated time slices, but the number of used cache lines varies among different sets. The INVALIDATE+COMPARE primitive empowers us to capture this behavior since it measures the contention intensity within the monitored cache sets. Note that, as the rendering process can extensively occupy the L2 cache space, we choose to use the variant with reading 9 addresses first followed by writing 7 addresses.

Figure 9 demonstrates the traces of contention intensity in 512 cache sets of the RTX 4060 GPU when three popular websites are being browsed in Chrome. A trace shown in Figure 9 contains 128 INVALIDATE+COMPARE measurements. (Each measurement is a vector of 512 elements whose value range is from 0 to 7.) To follow the established parlance [47, 56], we also refer to these traces as memorygrams. It is apparent that such memorygrams can provide certain insights into the contention behavior of GPU cache during the web page rendering process. More importantly, it is not difficult to observe that such memorygrams can be utilized to effectively distinguish between these three websites.

5.3.2 Evaluation

We evaluate our technique against a set of 50 websites, which are detailed in Appendix D. These websites are popular in the English-speaking world and chosen according to Similarweb rankings. As the main purpose of this case study is to illustrate the practical use of our attack primitive, we believe that a closed-world setting will suffice at this point. (The major challenge in open-world settings is how to achieve reliable open set recognition, which falls outside the scope of this paper.) In addition, due to Chrome's dominant market share, we focus our evaluation solely on this browser. The OS used in our evaluation is Ubuntu 20.04.

For each website, we gather 100 memorygrams similar to the ones depicted in Figure 9. Regardless of the GPU model, we choose to let one SM monitor 64 L2 cache sets (i.e., two warps running on an SM). Due to the relatively small L2 sizes in RTX 3060 and 3080, we can monitor all of their cache sets using 18 and 40 SMs, respectively. However, there are 12,288 cache sets in an RTX 4060 GPU but only 24 SMs, and thus we just monitor 1472 of them using 23 SMs with the last SM used for context switching detection (see Section 4.1).

Each memorygram consists of the contention intensity measurements in 5 seconds. As we have 50 websites, our data set has 5,000 memorygrams in total. For the task of classifying memorygrams into respective websites, we use the standard DenseNet-121 model. To assess the performance of our website fingerprinting, we conduct a 5-fold cross-validation on our collected data set. For each fold, we determine its accuracy, precision, and recall. As all websites are treated equally in our evaluation, we employ macro-averaging when calculating precision and recall (see Appendix D).

Table 2: Results of the 5-fold cross-validation.

	Accuracy			Precision			Recall		
	Max.	Min.	Avg.	Max.	Min.	Avg.	Max.	Min.	Avg.
RTX 3060	98.9%	98.1%	98.4%	98.9%	98.1%	98.4%	99.0%	98.2%	98.5%
RTX 3080	99.3%	98.5%	98.9%	99.3%	98.5%	98.9%	99.4%	98.6%	98.9%
RTX 4060	99.5%	98.5%	99.0%	99.5%	98.5%	99.0%	99.5%	98.6%	99.0%

The results of the 5-fold cross-validation are presented in Table 2. From the results, we can observe that the accuracy in each fold on any GPU is >98% and the average accuracy over the 5 folds is >98% as well, which highlights the effectiveness of our website fingerprinting technique. Meanwhile, it is noted that both the precision and recall in each fold are also >98%, which indicates that when a memorygram is classified as belonging to a website, there is a >98% chance it is correct and when a memorygram originates from a website, there is a >98% chance it will be accurately recognized as such.

To substantiate that the underlying platforms will not affect the results as long as the same GPU model is used, we evaluate the fingerprinting performance using two separate computer machines, M_1 and M_2, whose configurations are specified in Table 3. Both of them are equipped with an RTX 3080 GPU. We highlight that the RTX 3080 GPU cards are of different brands (Gigabyte and Founders Edition).

Table 3: Computers equipped with different RTX 3080 cards.

Machine	GPU Brand	CPU	Motherboard	Main Mem.
M_1	Gigabyte	AMD Ryzen 5 5500	Asus Prime B450M-A II	16GB
M_2	Founders Ed.	Intel Core i3-10100	ASRock H570 Steel Legend	32GB

On the machine M_1, 100 memorygrams per website are collected, and a DenseNet-121 model is trained using these memorygrams. On the machine M_2, 20 memorygrams per website are collected, and we apply the trained model to classify such data. Figure 16 in Appendix E shows the resulting confusion matrix. In summary, the average accuracy is 97.7%, precision is 97.7%, and recall is 97.8%, when tested with 1,000 samples from M_2. It is evident that the results comparatively align with those of the 5-fold cross-validation.

In reality, most websites update their page content very frequently, often on a daily basis or even on an hourly basis. An interesting scenario to investigate is whether the model, once trained, can maintain its effectiveness in fingerprinting websites for a longer term. To assess this, we train a DenseNet-121 model with the initial 5,000 memorygrams of our RTX 3080. Subsequently, for each website, we gather an additional 10 memorygrams on the 3rd, 5th, and 7th days after the original data collection for testing. Figure 10 presents the results.

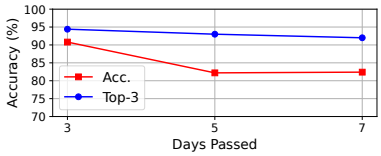


Figure 10: Results of effectiveness testing over time.

As anticipated, there is some degradation in performance, but it still maintains a substantial level of effectiveness. Even after one week, our accuracy can still reach 82.40%. Aside from trying to exactly fingerprint the websites, we evaluate its Top-3 accuracy as well, which remains high and relatively stable over time (around 93%). It is intriguing to see that for certain websites tending to update their contents frequently, such as YouTube and BBC, our model is still able to accurately identify them. Irrespective of frequent content changes on these websites, their overall page layout or structure does not alter as much. For example, the number and positions of pictures and/or videos are stable but only the materials shown by them are changed. Our conjecture is that this structural consistency is the main contributing factor to the model's fingerprinting effectiveness over time, as it can lead to similar web page rendering workload and patterns.

5.4 Virtual Keystroke Extraction

In the second case study, we aim to recover more fine-grained and sensitive information beyond merely website visit data. To this end, we focus on the keystrokes made by a user on a virtual keyboard (also known as an on-screen keyboard) and show that INVALIDATE+COMPARE enables us to steal them.

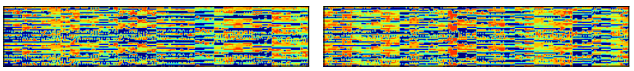
Note that using a virtual keyboard is not unusual in practice. For example, it is common and crucial among people with special needs (e.g., those with disabilities), and it is treated as a countermeasure against keyloggers (even though its effectiveness is debatable [40]). Moreover, with the proliferation of touchscreen-based laptops and monitors, virtual keyboards are becoming increasingly popular for day-to-day tasks. Major consumer-facing OSes all provide default virtual keyboards as an accessibility feature.

5.4.1 Investigation

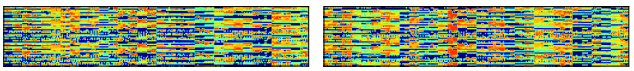
In a GUI-based modern OS, its windowing system (e.g., X11 or Wayland in Linux) provides the protocol and mechanics for graphics rendering and input handling. Built on top of this, the desktop environment (e.g., GNOME or KDE) offers the GUI interface. (Roughly speaking, the desktop environment is an application running under a windowing system.) Usually, the default virtual keyboard bundled with the OS is an integrated feature of the desktop environment [5].

Windowing systems are often designed with considerations for efficient graphics rendering, and they generally attempt to avoid re-rendering the entire screen when only a small portion of the screen changes. Take X11 with the DAMAGE extension as an example.[3] (This extension is incorporated in nearly all display server implementations of X11, such as the widely used Xorg.) When a graphical element in an application running under this setup is modified, the DAMAGE extension identifies the affected region and marks it as "dirty". Compositing window managers compatible with X11 system will then focus on re-compositing only the "dirty" region of the surfaces to enhance efficiency.

With respect to a virtual keyboard, when a key is activated, it normally exhibits certain local visual effects (e.g., shading the surroundings of the key). Consequently, the windowing system should focus on re-rendering only that particular area. Note that windowing systems typically leverage the power of GPUs for graphic rendering, and they also maintain various buffers in GPU memory, one of which in fact corresponds to the desktop environment. When a virtual keystroke is made, it is conceivable that only the parts of this buffer associated with the "dirty" regions (i.e., those affected by the keystroke) are accessed. This implies that distinct keys may access different GPU memory addresses. Therefore, we hypothesize that it is possible to produce distinguishable traces in the GPU cache when pressing different keys.



(a) Key 'a' sample 1. (b) Key 'b' sample 1.



(c) Key 'a' sample 2. (d) Key 'b' sample 2.

Figure 11: Reshaped memorygrams corresponding to pressing keys 'a' and 'b' on the GNOME virtual keyboard under the default X11 windowing system when an RTX 3080 GPU is used.

As evidence supporting our hypothesis, Figure 11 depicts two pairs of memorygrams generated on an RTX 3080 when pressing the letters 'a' and 'b' under the X11 windowing system. Note that the original memorygrams have a dimension of 5×2560, because we monitor 2560 L2 cache sets and the keystroke visual effect rendering induces about 5 GPU context switches. For clarity in visualization, we reshape them into

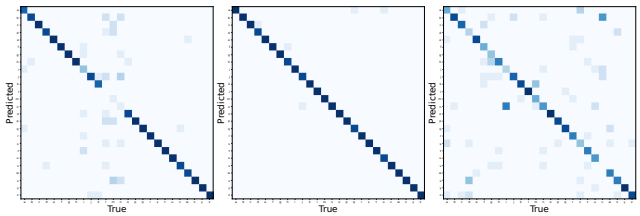
[3] The newer Wayland windowing system has the concept of "dirty" regions handling from the ground up.

50×256. Moreover, we have filtered the level 7 intensity to prevent visual clutter. While there are similarities among them, it shall not be hard to find that discernible and repeatable patterns do exist to classify these two keys.

5.4.2 Evaluation

We carry out our evaluations on a Linux platform running Ubuntu 20.04. By default, Ubuntu 20.04 uses X11 as its windowing system.[4] In terms of the desktop environment, we use the default GNOME coming with the OS.

The GNOME virtual keyboard displays only letters every time it is launched. As a first step, we investigate how accurately the memorygrams can be used to deduce the pressed letters. For this purpose, we collect 100 memorygrams for each letter, which are used to train a DenseNet-121 model. Given a GPU, the number of monitored cache sets and the number of warps on each SM are consistent with our previous case study. Subsequently, we capture another 10 memorygrams per letter for testing. The resulting confusion matrices are presented in Figure 12.



(a) RTX 3060. (b) RTX 3080. (c) RTX 4060.

Figure 12: Heat maps corresponding to the testing results based on memorygrams for ‘a’ to ‘z letters.

From the confusion matrices, we can observe that letters, in most cases, are distinguishable. The average accuracies for RTX 3060, 3080, and 4060 are 83.1%, 98.1%, and 67.7% respectively. In comparison, a random guess will yield an accuracy of just 3.8%. This disparity underscores the information leakage from GPU cache activities during the rendering of keystroke visual effects on the screen.

For the RTX 3060 GPU, we note a peculiar trend: the recognition performance for letters ‘l’, ‘m’, and ‘n’ is substantially lower than for the other letters. Conversely, the RTX 3080 outperforms both, delivering the best results. It is interesting to note that while RTX 3080 has a larger cache size than RTX 3060, it is significantly smaller than that of RTX 4060. However, the recognition performance of RTX 4060 is the lowest.

The lower performance of the RTX 4060 relative to the RTX 3060 and 3080 can be attributed to the fact that we only monitored 1472 cache sets for the RTX 4060, whereas all cache sets are monitored for the other two GPUs. While this configuration suffices for coarse-grained tasks like website fingerprinting, fine-grained tasks necessitate the selection of more appropriate cache sets. We believe that by identifying the most relevant ones, the performance on the RTX 4060 can be substantially improved.

To access the numbers, users have to utilize the switch key located at the bottom left, and when activated, the numbers replace the first row of 10 letters exactly. We aim to determine if the switch action and the numbers can be differentiated from the letters (especially those in the first row) using our memorygrams. To this end, we use 100 samples each for the letters, numbers, and both switching actions (to numbers and back to letters) to train a DenseNet-121 model, and another 10 samples for testing.

Considering the action of switching to or from numbers, our data set comprises 37 classes. For the GPUs RTX 3060, 3080, and 4060, the average accuracies achieved are 68.6%, 77.3%, and 67.3%, respectively. Intriguingly, despite the numbers and the letters from the first row occupying the same screen position, they are distinguishable to some extent. (The corresponding confusion matrices are given in Figure 15 in Appendix E.) We believe the distinguishability between characters, such as ‘p’ and ‘0’, is due to each being mapped to distinct buffer addresses in the GPU memory.

A further point of interest is the contrasting performance trends between the GPUs. While both RTX 3060 and 3080 exhibit a decrease in accuracy, the RTX 4060’s performance remains very consistent. This stability can be attributed to RTX 4060’s very large cache size, which has the capacity to house the entire frame buffer (even though we have not chosen the most appropriate ones).

Next, we evaluate if we can extract a user’s system login password entered via the virtual keyboard. For demonstration purposes, we select four commonly used and pwned passwords as listed by the National Cyber Security Centre [39]. These four passwords are given in Table 4.

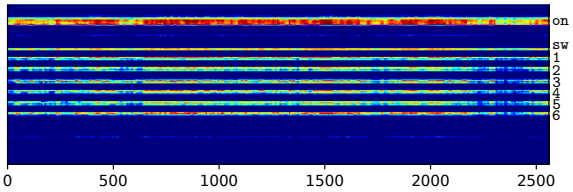


Figure 13: GPU L2 contention intensity trace, where the x-axis gives cache sets and the y-axis gives the time (derived from the recorded context switch times as stated in Section 4.1).

We capture the GPU L2 contention intensity traces when the victim is prompted to input their system login password. This scenario can occur frequently because, by default, Ubuntu, along with many other Linux distributions, goes to a black screen after 5 minutes of inactivity. Upon returning, the user is asked to provide the login credentials. Figure 13 illus-

[4] Although newer versions of Ubuntu, e.g., 22.04, have made Wayland the default windowing system, there are compatibility issues with NVIDIA GPUs. As a result, when NVIDIA GPUs are detected, the system will automatically switch back to X11.

trates a trace captured on an RTX 3080 GPU corresponding to the input of the password shown in the first entry of Table 4. From this trace, it is evident that the individual memorygrams can be readily distinguished. The topmost bar signifies the system's awakening from the black screen state, followed by the second bar which indicates the act of toggling the number/letter switch.

Table 4: Results of inferring four commonly used login passwords. A correct inference is denoted by ✓and an incorrect one is denoted by ✗. The symbol '@' indicates the action of switching between letters and numbers.

Password	Keystrokes	RTX 3060	RTX 3080	RTX 4060
123456	@123456	✓✓✗✗✓✗✓	✓✗✓✓✓✓✓	✓✓✗✓✗✓✓
qwerty	qwerty	✓✗✓✓✗✓	✓✓✗✓✗✓	✓✓✗✓✗✓
abc123	abc@123	✓✗✓✓✗✓✓	✓✗✓✓✗✓✓	✓✗✓✗✗✓✓
1qaz2wsx	@1@qaz@2@wsx	✓✗✓✗✓✓✓✗✓✗✓✓	✓✗✓✗✓✓✓✗✓✗✓✓	✓✓✓✗✓✓✓✓✗✗✓✓

For our evaluation, we employ a DenseNet-121 model, training it with 300 samples for each character. An analysis of Table 4 reveals that we can indeed decipher many segments of these passwords. A significant portion of the errors stems from the confusion between numbers and letters. The most reliably recognized keystroke is the switching between letters and numbers.

5.5 Discussion

A common limitation across both case studies is that they require the execution of native code with user-level privileges on the victim's machine. However, it is essential to note that such a condition is not always necessary for employing the primitive in some more practical scenarios. For example, we find that many major cloud service providers (e.g., Azure [34]) offer VMs powered by NVIDIA vGPUs, and in this context, our INVALIDATE+COMPARE primitive can be leveraged to facilitate cross-vGPU side-channel attacks without requiring native code execution on the victim's VM. Moreover, a number of companies have adopted virtual desktop infrastructures sharing high-end GPUs to support their daily business operations [42, 46], creating additional venues where the primitive can be used for potential side-channel attacks.

We should also mention that unlike website fingerprinting, the demonstrated virtual keystroke extraction is a more fine-grained attack, whose success greatly depends on whether the victim's on-screen keyboard uses the same GPU memory area as the one used by the attacker during profiling. While this is difficult to control when the GPU is actively in use, we have discovered that when the system is idle for a certain period, the X11 server reduces its usage of GPU physical memory to a fixed level. Upon activation, the system normally utilizes the same GPU memory area for the virtual keyboard, which creates a favorable scenario for the attacker to perform the keystroke extraction.

6 Countermeasures

To prevent the exploitation of our timer-free attack primitive, we propose several possible countermeasures. The first one is to simply disable the use of the `discard` instruction. To this end, we can directly remove the support for this PTX instruction in the compiler toolchain. However, this straightforward patch can be easily circumvented by altering instructions in the compiled binary files to match the encoding of `discard`. An alternative method is to enable the NVIDIA driver to inspect user programs for `discard` instructions and reject the launch of their kernels if detected. While such an in-driver defense raises the bar for attackers, it can still be bypassed by modifying instructions in GPU code pages at run time.

A more effective solution is to disable this instruction in hardware. Nevertheless, due to the lack of publicly accessible information, it is uncertain whether the instruction can be disabled via a GPU firmware update, or if a less desirable hardware modification is necessary. Even if a firmware update can achieve the objective, the potential ramifications for existing software still need to be determined.

As the root cause of the examined information leakage stems from observable contentions in the GPU L2 cache, the other mitigation direction we can work on is to seal off this source. Given the fact that GPU contexts cannot execute in parallel, our focus narrows to managing the serialized use of the L2 cache. One feasible way to achieve this is to let the runtime flush the contents in L2 every time GPU contexts are scheduled. While clean cache lines can be directly invalidated, dirty ones need to be written back to GPU memory. A primary concern with this approach lies in its potential impact on GPU performance. However, as L2 flushing is only triggered upon scheduling GPU contexts, an event that typically occurs on the millisecond level, our anticipation is that the performance degradation may be limited.

Additionally, given that eviction sets can be prepared a single time offline and then utilized across all future online attacks, it is undeniable that the task of an attacker is considerably eased. This convenience arises from the way NVIDIA drivers allocate GPU page frames, favoring physical contiguity and consistently starting the allocation from the same address. To neutralize this benefit for attackers, NVIDIA should consider introducing a degree of randomness into its drivers for GPU memory allocation.

7 Related Work

Extensive studies have been conducted on cache side-channel attacks in the past two decades, and the majority of them target cache hierarchies in CPUs like x86 and ARM [4, 32]. Overall, these studies exploit various attack primitives, which can be classified into eviction-based and flush-based types. PRIME+PROBE is the representative of the eviction-based attack primitives [48], and it can be used for leaking

information across domains established by browser sandboxes [47, 55], SGX enclaves [35, 54, 70], VMs [15, 17, 30], and even networks [25, 61]. In terms of flush-based primitives, FLUSH+RELOAD [10, 73] is the most notable one and has been employed in a number of attacks [9, 18, 24, 28, 69]. Flush-based primitives typically can offer finer leakage granularity than eviction-based ones, but they need to have shared memory with the victim. Our attack primitive, in spite of using a flush-like instruction on GPUs, is fundamentally eviction-based.

Traditionally, cache side-channel attack primitives require a high-resolution timer to distinguish between hits and misses. As exceptions to this rule, several timer-free ones have been devised, like PRIME+ABORT [2], DPRIME+DABORT [23], S^2C [74], and `mwait`-based primitives [76]. While these primitives target x86 and ARM CPUs, ours is developed for NVIDIA GPUs.

With GPUs becoming indispensable in modern computing, their security issues have garnered much attention. There have been works that examine website fingerprinting on GPUs [27, 38, 57, 60, 72, 75]. While some of them capitalize on software vulnerabilities present in device drivers [27, 38], others hinge on coarse-grained contention, whether over PCIe [57, 60] or on the rendering pipeline [72]. These studies largely sidestep delving into the GPU's microarchitectural intricacies, which is a primary emphasis of this paper.

On the other hand, techniques for keystroke extraction via GPUs have also been investigated [38, 60, 72]. These studies exploit the dependence of inter-keystroke timings on key distance and location to deduce which key is pressed [36, 58]. In contrast, our work is similar to approaches that leverage different cache activities to directly extract the keys [9, 69]. Moreover, a GPU keylogger that reads the keyboard buffer over PCIe bus has been described in [26].

Besides our work, there exist other studies on exploiting data caches of discrete GPUs for information leakage, all of which are built on the traditional PRIME+PROBE primitive [3, 37]. In [37], several covert channels for data exfiltration are constructed, but they require operation under MPS. In [3], other than covert channel attacks, an application fingerprinting case study is presented, albeit limited to only 6 applications. Note that restricting access to timers, or adding noise to them, which are used on CPUs, can also be applied to GPUs to mitigate such PRIME+PROBE-based attacks; however, these countermeasures will not affect attacks utilizing our timer-free primitive.

8 Conclusion

In this paper, we have unveiled certain previously-unknown characteristics of NVIDIA GPU caches for the first time and introduced a new GPU cache side-channel attack primitive. This new primitive enables effective spying on GPU cache activities without relying on timers. Capitalizing on this approach, we have successfully orchestrated website fingerprinting and keystroke extraction attacks on NVIDIA's latest Ampere and Ada Lovelace GPUs. To our knowledge, these represent the first timer-free cache side-channel attacks on GPUs. Our work highlights the need to scrutinize the information leakage possibilities within GPU caches against microarchitectural attacks.

Availability. The proof-of-concept implementation of our primitive, including the method for constructing GPU L2 cache eviction sets, is available at https://github.com/0x5ec1ab/invalidate-compare.git.

Acknowledgments

This work is supported in part by the NSF (CNS-2147217, CNS-2340777, and CNS-2054657). The authors thank the anonymous reviewers and shepherd for their comments and suggestions that help us improve the quality of the paper.

References

[1] Xiang Cai, Xin Cheng Zhang, Brijesh Joshi, and Rob Johnson. Touching from a Distance: Website Fingerprinting Attacks and Defenses. In *CCS '12*.

[2] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *USENIX Security '17*.

[3] 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 '23*.

[4] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8:1–27, 2018.

[5] GNOME. https://www.gnome.org/.

[6] Khronos Group. Vulkan - Cross platform 3D Graphics. https://www.vulkan.org/.

[7] Khronos Group. OpenGL - The Industry Standard for High Performance Graphics. https://www.opengl.org/.

[8] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA '16*.

[9] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security '15*.

[10] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *S&P '11*.

[11] Berk Gulmezoglu, Andreas Zankl, Thomas Eisenbarth, and Berk Sunar. PerfWeb: How to Violate Web Privacy with Hardware Performance Events. In *ESORICS '17*.

[12] Jamie Hayes and George Danezis. k-fingerprinting: A Robust Scalable Website Fingerprinting Technique. In *USENIX Security '16*.

[13] Andrew Heaton. Nvidia Sold Out of RTX 4090 Graphics Cards within Two Weeks. https://gamerant.com/nvidia-sold-out-rtx-4090-graphics-cards-two-weeks/.

[14] Andrew Hintz. Fingerprinting Websites Using Traffic Analysis. In *PET '02*.

[15] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache Attacks Enable Bulk Key Recovery on the Cloud. In *CHES '16*.

[16] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[17] 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 '15*.

[18] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a Minute! A fast, Cross-VM Attack on AES. In *RAID '14*.

[19] Suman Jana and Vitaly Shmatikov. Memento: Learning Secrets from Process Footprints. In *S&P '12*.

[20] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. Dissecting the NVIDIA Turing T4 GPU via Microbenchmarking. *CoRR*, abs/1903.07486, 2019.

[21] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *CoRR*, abs/1804.06826, 2018.

[22] Mahmoud Khairy, Zhesheng Shen, Tor Aamodt, and Timothy Rogers. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *ISCA '20*.

[23] Sowoong Kim, Myeonggyun Han, and Woongki Baek. DPrime+DAbort: A High-Precision and Timer-Free Directory-Based Side-Channel Attack in Non-Inclusive Cache Hierarchies using Intel TSX. In *HPCA '22*.

[24] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre Attacks: Exploiting Speculative Execution. In *S&P '19*.

[25] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical Cache Attacks from the Network. In *S&P '20*.

[26] Evangelos Ladakis, Lazaros Koromilas, Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. You Can Type, but You Can't Hide: A Stealthy GPU-based Keylogger. In *EuroSec '13*.

[27] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities. In *S&P '14*.

[28] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security '18*.

[29] Chen Liu, Abhishek Chakraborty, Nikhil Chawla, and Neer Roggel. Frequency Throttling Side-Channel Attack. In *CCS '22*.

[30] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-Level Cache Side-Channel Attacks are Practical. In *S&P '15*.

[31] Zhiye Liu. Nvidia's RTX 4080 Tops Newegg's List of Best-Selling GPUs. https://www.tomshardware.com/news/nvidia-rtx-3080-top-newegg-gpu.

[32] Yangdi Lyu and Prabhat Mishra. A Survey of Side-Channel Attacks on Caches and Countermeasures. *Journal of Hardware and Systems Security*, 2:33–50, 2018.

[33] Xinxin Mei and Xiaowen Chu. Dissecting GPU Memory Hierarchy Through Microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, jan 2017.

[34] Microsoft. NVadsA10 v5-series. https://learn.microsoft.com/en-us/azure/virtual-machines/nva10v5-series.

[35] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *CHES '17*.

[36] John V Monaco. SoK: Keylogging Side Channels. In *S&P '18*.

[37] Hoda Naghibijouybari, Khaled N. Khasawneh, and Nael Abu-Ghazaleh. Constructing and Characterizing Covert Channels on GPGPUs. In *MICRO '17*.

[38] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered Insecure: GPU Side Channel Attacks Are Practical. In *CCS '18*.

[39] NCSC. https://www.ncsc.gov.uk/static-assets/documents/PwnedPasswordsTop100k.txt.

[40] Leo Notenboom. Will Using an On-Screen Keyboard Stop Keyloggers? https://askleo.com/will_using_an_on_screen_keyboard_stop_keyboard_loggers_and_hackers/.

[41] NVIDIA. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[42] NVIDIA. GPU-Powered Virtual Workstations Offer Greater Performance and Flexibility. https://resources.nvidia.com/en-us-media-and-entertainment/vgpu-media-entertain.

[43] NVIDIA. NVIDIA Ampere Architecture Whitepaper. https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.1.pdf.

[44] NVIDIA. NVIDIA Announces Financial Results for Fourth Quarter and Fiscal 2022. https://nvidianews.nvidia.com/news/nvidia-announces-financial-results-for-fourth-quarter-and-fiscal-2022.

[45] NVIDIA. Parallel Thread Execution ISA Version 8.1. https://docs.nvidia.com/cuda/parallel-thread-execution/.

[46] NVIDIA. vGPU and AECO Solution Showcase. https://images.nvidia.com/content/Solutions/data-center/vgpu-aeco-solution-showcase.pdf.

[47] 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 '15*.

[48] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA '06*.

[49] Andriy Panchenko, Fabian Lanze, Jan Pennekamp, Thomas Engel, Andreas Zinnen, Martin Henze, and Klaus Wehrle. Website Fingerprinting at Internet Scale. In *NDSS '16*.

[50] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc.

[51] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In *CCS '21*.

[52] Vera Rimmer, Davy Preuveneers, Marc Juarez, Tom Van Goethem, and Wouter Joosen. Automated website fingerprinting through deep learning. In *NDSS '18*.

[53] Peter Van Sandt and Zhe Jia. Dissecting the Ampere GPU Architecture through Microbenchmarking. https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s33322/.

[54] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA '17*.

[55] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses. In *USENIX Security '21*.

[56] 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 '19*.

[57] Mert Side, Fan Yao, and Zhenkai Zhang. LockedDown: Exploiting Contention on Host-GPU PCIe Bus for Fun and Profit. In *EuroS&P '22*.

[58] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing Analysis of Keystrokes and Timing Attacks on SSH. In *USENIX Security '01*.

[59] Wei Song and Peng Liu. Dynamically Finding Minimal Eviction Sets Can Be Quicker Than You Think for Side-Channel Attacks against the LLC. In *RAID '19*.

[60] Mingtian Tan, Junpeng Wan, Zhe Zhou, and Zhou Li. Invisible Probe: Timing Attacks with PCIe Congestion Side-channel. In *S&P '21*.

[61] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Packet Chasing: Spying on Network Packets over a Cache Side-Channel. In *ISCA '20*.

[62] TechPowerUp. NVIDIA A10 PCIe. https://www.techpowerup.com/gpu-specs/a10-pcie.c3793.

[63] TechPowerUp. NVIDIA A2 PCIe. https://www.techpowerup.com/gpu-specs/a2-pcie.c4112.

[64] TechPowerUp. NVIDIA GeForce RTX 3060 12 GB. https://www.techpowerup.com/gpu-specs/geforce-rtx-3060-12-gb.c3682.

[65] TechPowerUp. NVIDIA GeForce RTX 4060. https://www.techpowerup.com/gpu-specs/geforce-rtx-4060.c4107.

[66] Mark Tyson. Nvidia RTX 3060 Begins its Reign as the Most Popular GPU. https://www.tomshardware.com/news/nvidia-rtx-3060-begins-its-reign-as-the-most-popular-gpu.

[67] Pepe Vila and Boris Kopf. Loophole: Timing Attacks on Shared Event Loops in Chrome. In *USENIX Security '17*.

[68] Pepe Vila, Boris Köpf, and José F Morales. Theory and Practice of Finding Eviction Sets. In *S&P '19*.

[69] Daimeng Wang, Ajaya Neupane, Zhiyun Qian, Nael B Abu-Ghazaleh, Srikanth V Krishnamurthy, Edward JM Colbert, and Paul Yu. Unveiling your keystrokes: A Cache-based Side-channel Attack on Graphics Libraries. In *NDSS '19*.

[70] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *CCS '17*.

[71] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying GPU Microarchitecture through Microbenchmarking. In *ISPASS '10*.

[72] Shujiang Wu, Jianjia Yu, Min Yang, and Yinzhi Cao. Rendering Contention Channel Made Practical in Web Browsers. In *USENIX Security '22*.

[73] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security '14*.

[74] Jiyong Yu, Aishani Dutta, Trent Jaeger, David Kohlbrenner, and Christopher W. Fletcher. Synchronization Storage Channels (S^2C): Timer-less Cache Side-Channel Attacks on the Apple M1 via Hardware Synchronization Instructions. In *USENIX Security '23*.

[75] Zihao Zhan, Zhenkai Zhang, Sisheng Liang, Fan Yao, and Xenofon Koutsoukos. Graphics Peeping Unit: Exploiting EM Side-Channel Information of GPUs to Eavesdrop on Your Neighbors. In *S&P '22*.

[76] Ruiyi Zhang, Taehyun Kim, Daniel Weber, and Michael Schwarz. (M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels. In *USENIX Security '23*.

[77] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *CCS '12*.

[78] Zhenkai Zhang, Tyler Allen, Fan Yao, Xing Gao, and Rong Ge. TunneLs for Bootlegging: Fully Reverse-Engineering GPU TLBs for Challenging Isolation Guarantees of NVIDIA MIG. In *CCS '23*.

[79] Zhenkai Zhang, Sisheng Liang, Fan Yao, and Xing Gao. Red Alert for Power Leakage: Exploiting Intel RAPL-Induced Side Channels. In *ASIA CCS '21*.

A Supplement to Write Policy Study

To confirm that B is never evicted from X's L1 cache in the experiment presented in Section 3.3 (see Figure 2), we proceed with the following three steps *after* Y *reads* B (recall that here Y always retrieves 1) – First, Y discards B. Next, Y reads B to check its value. Finally, X reads B to check its value.

We consistently observe that after Y executes the `discard` instruction on B, its subsequent read returns 0, as expected since B is now fetched from GPU memory. Notably, X consistently retrieves the value 1 for B. If B had been evicted from X's L1 cache, X would also retrieve 0, echoing Y's read value after invalidation of B. Therefore, this indicates that B remains in X's L1 cache throughout the experiment presented in Section 3.3.

B Replacements during Set Population

In the case where only `ld` is used, we surprisingly observe replacements in the course of filling a set, as illustrated in Figure 14. Starting from an empty cache set, we first read from 8 addresses (i.e., $A_0, A_1, \cdots, A_8$), reaching the state S_0. Interestingly, when loading from another address A_9, the cache line corresponding to A_0 is evicted, even though the cache set is only half full. However, when reading from A_{10}, no eviction occurs until the subsequent load. This process continues until the cache set reaches the filled state (i.e., the state S_{14}). After this point, the cache set's replacement behavior using `ld` aligns with the LRU policy. Notably, the initial replacement pattern does not match any known policy.

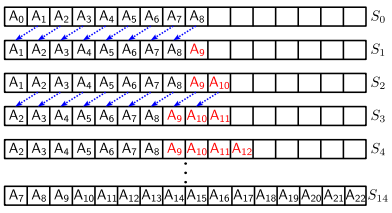
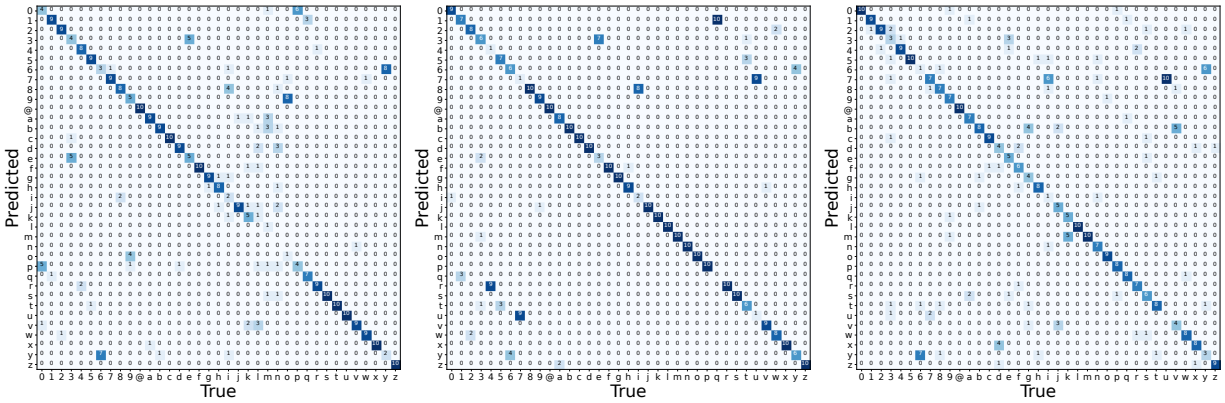


Figure 14: Populating an L2 cache set using `ld`.



(a) RTX 3060. (b) RTX 3080. (c) RTX 4060.

Figure 15: Confusion matrix corresponding to the testing results based on memorygrams for 'a' to 'z letters, '0' to '9 numbers, and switch action between letters and numbers (shown as '@').

C Time Slice Measurement

We cannot determine the exact allocated time slices, but we can approximate them using the first context switch detection approach depicted in Figure 7a. The `delta` from this approach provides a tight upper bound for the allocated time slice. We assessed the time slices using this method across several applications, such as GPU benchmarks, neural network training, games, and web browsers. We have observed that GPU benchmarks and games tend to have large time slices. However, our primary concern is identifying the minimum time slice, which we find to be generated predominantly by web browsers.

D Website List

Table 5 lists the websites that are used in our evaluations on website fingerprinting. Since there are 50 websites, the macro-averaged precision is calculated as $\frac{1}{50}\sum_{i=1}^{50} P(i)$ and the macro-averaged recall is calculated as $\frac{1}{50}\sum_{i=1}^{50} R(i)$, where $P(i)$ and $R(i)$ represent the precision and recall for the i^{th} website, respectively.

Table 5: List of fingerprinted websites

1: www.accuweather.com	2: www.adobe.com	3: www.amazon.com
4: www.aol.com	5: www.apple.com	6: www.bankofamerica.com
7: www.bbc.com	8: www.bestbuy.com	9: www.bing.com
10: www.booking.com	11: www.chase.com	12: www.cnn.com
13: www.craigslist.org	14: www.discord.com	15: www.duckduckgo.com
16: www.ebay.com	17: www.espn.com	18: www.etsy.com
19: www.facebook.com	20: www.fandom.com	21: www.foxnews.com
22: www.google.com	23: www.homedepot.com	24: www.imdb.com
25: www.instagram.com	26: www.linkedin.com	27: www.live.com
28: www.max.com	29: www.msn.com	30: www.netflix.com
31: www.nfl.com	32: www.nytimes.com	33: www.office.com
34: www.quora.com	35: www.reddit.com	36: www.roblox.com
37: www.slack.com	38: www.spotify.com	39: www.tumblr.com
40: www.twitch.tv	41: www.twitter.com	42: www.usatoday.com
43: www.walmart.com	44: www.weather.com	45: www.whatsapp.com
46: www.wikipedia.org	47: www.yahoo.com	48: www.youtube.com
49: www.zillow.com	50: www.zoom.us	

E Confusion Matrices

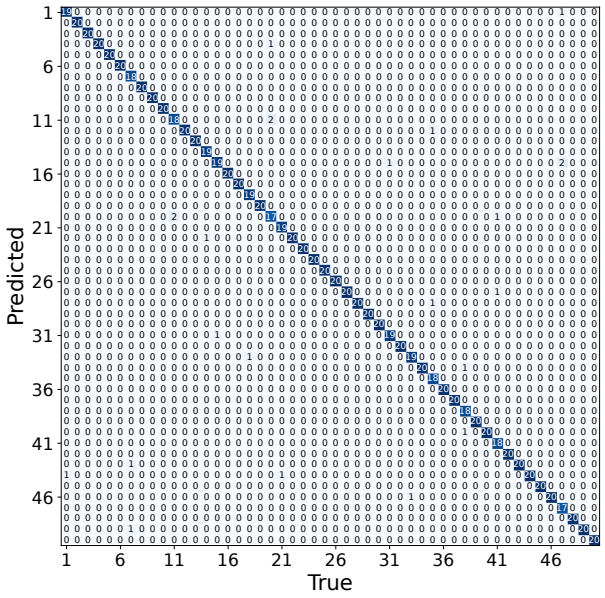


Figure 16: Confusion matrix corresponding to using the model trained with data from M_1 to classify 1,000 samples from M_2. The numbers on the axes correspond to the websites in Table 5.