Invalidate+Compare: A Timer-Free GPU Cache Attack Primitive

USENIX Security Symposium (August 14–16, 2024)

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 Delawares

Agenda

1. Motivation
2. Background
3. Experiments & Observations
4. Threat Model
5. Attack
6. Case Studies
7. Conclusion

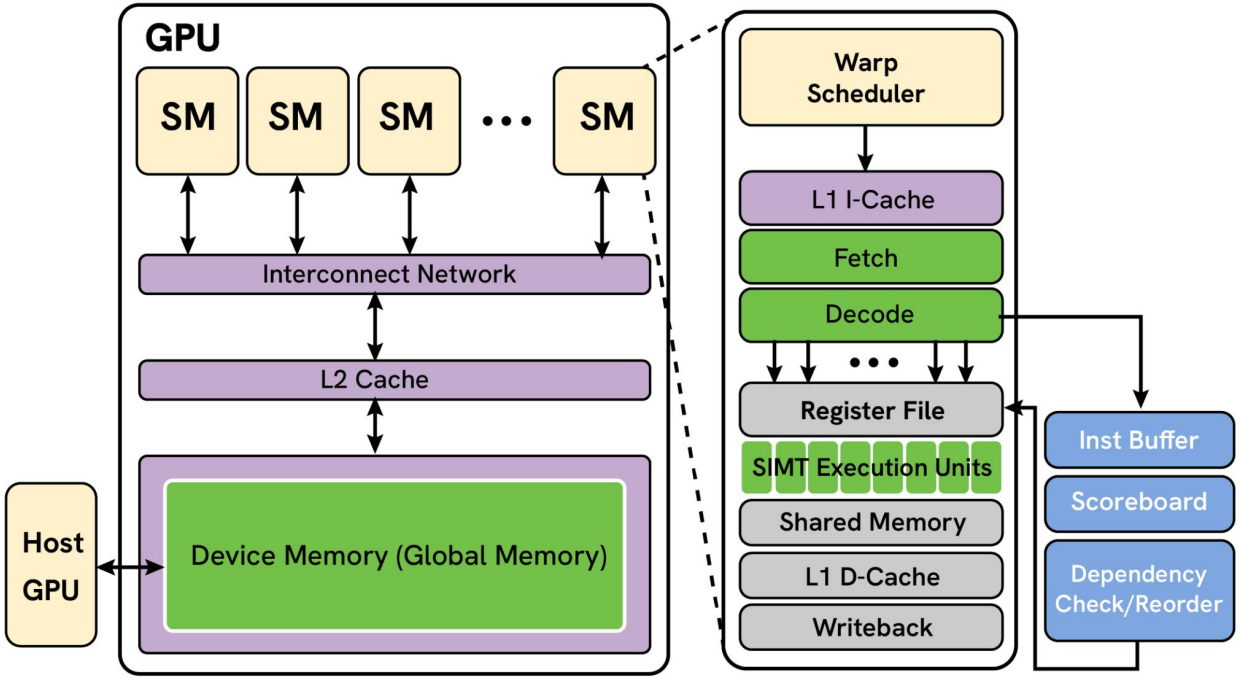
Motivation – The rise of GPUs

Over the **last few years**, GPUs **have made** a significant impact not only in gaming, but also in **dynamic simulation** and **deep learning**.

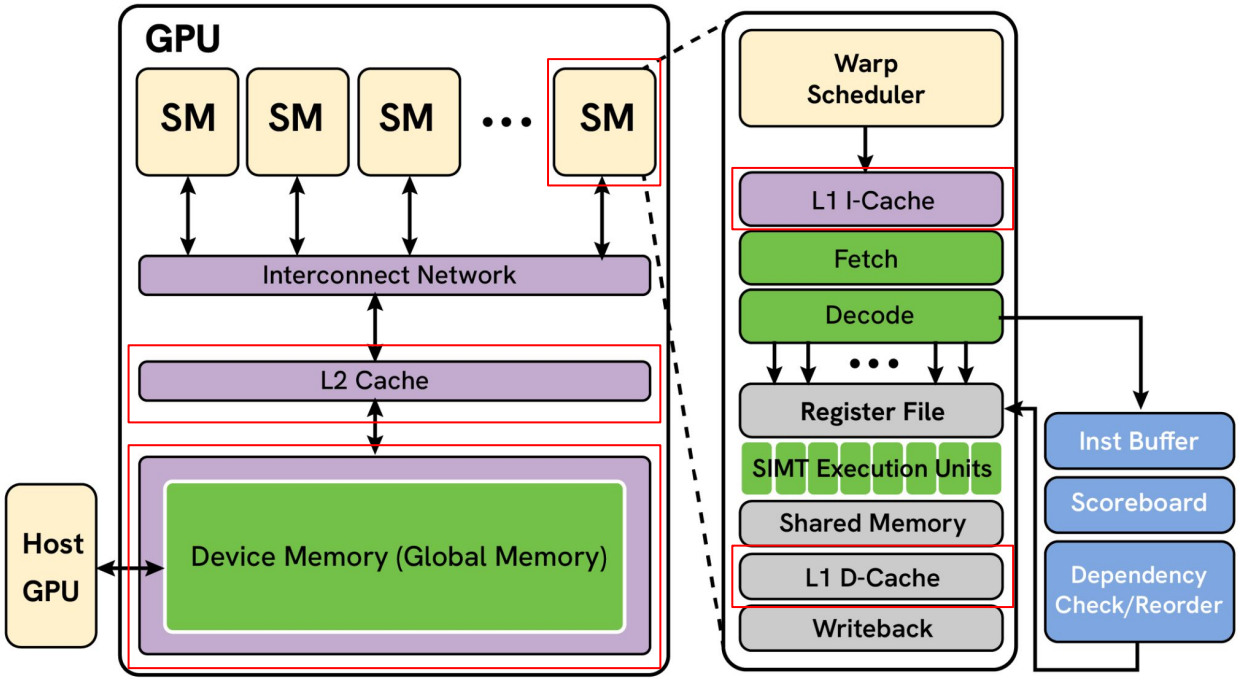
NVIDIA has established itself as the dominant player in the GPU market.

However, despite the widespread adoption of these GPUs, their potential security issues **from a hardware perspective** have not been as thoroughly studied as those of CPUs.

Background – GPU Architecture

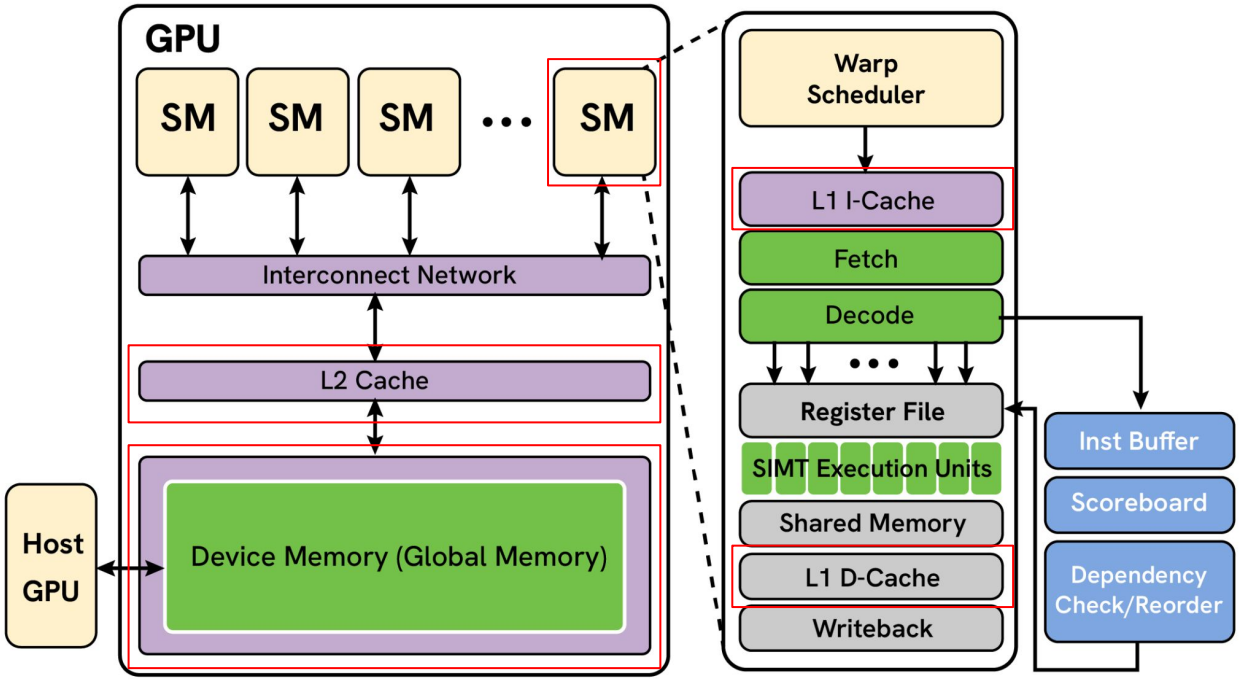


Background – GPU Architecture



www.cantech.in

Background – GPU Architecture



Streaming Multiprocessor (SM): An array containing multiple processing cores.

L1i / L1d Cache: Similar to CPU L1, but shared among all cores within an SM.

L2 Cache: Comparable to a CPU's L3 cache; acts as the **Last-Level Cache (LLC)** for the entire GPU.

Device Memory: The GPU's main memory, analogous to system DRAM.

Background – NVIDIA GPU Programming

CUDA: NVIDIA's parallel computing platform and programming model.

Compilation Flow: Programs compile into **PTX** (Parallel Thread Execution) Intermediate Representation (IR), which is then translated into GPU-specific Assembly (SASS).

The discard Instruction: An **unprivileged** key PTX instruction, similar to **unprivileged clflush** in x86 architectures.

- **clflush**: Flushes a cache line from all levels; if "dirty," it writes data back to main memory.
- **discard**: Invalidates data in the L2 cache **without** writing it back to GPU memory.

Background – NVIDIA GPU Programming

Syntax Example: `discard.L2 [addr], 128;`

- *Invalidates the 128-byte range starting at [addr] in the L2 cache.*

Differences between discard & clflush:

1. clflush removes the target cache line from the entire cache hierarchy, while **discard only removes the cache line from L2 (as specified).**
2. 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.

Background – Dimensions of GPU Cache Behavior

Inclusion Policy (Location): Determines if data in a higher-level cache (L1) must also exist in a lower-level cache (L2).

- *Key question: Where is the data allowed to reside?*

Write Policy (Timing): Dictates when a data update is propagated to the next level (e.g., L1 to L2).

- *Key question: When do we notify the next level that data has changed?*

Write Allocation Policy: Defines how the system handles a "write miss."

- *Key question: Do we load data into the cache before modifying it?*

Background – Inclusion Policies

Inclusive: Every block in L1 **must** also reside in L2. (Simplifies consistency).

Exclusive: A block in L1 **cannot** be in L2. (Optimizes total capacity).

Non-Inclusive: A "middle ground" where a block in L1 **may or may not** be in L2.

Industry Standards:

- **L3 vs. L2/L1:** Typically Inclusive or Non-Inclusive.
- **L2 vs. L1:** Typically Exclusive or Non-Inclusive.

Background – Cache Write & Allocation Policies

Write-Through: Data is written to the current cache and **immediately** updated in the next level (e.g., L1 → L2).

Write-Back: Data is updated **only** in the current cache. The next level is updated only when the line is **evicted**.

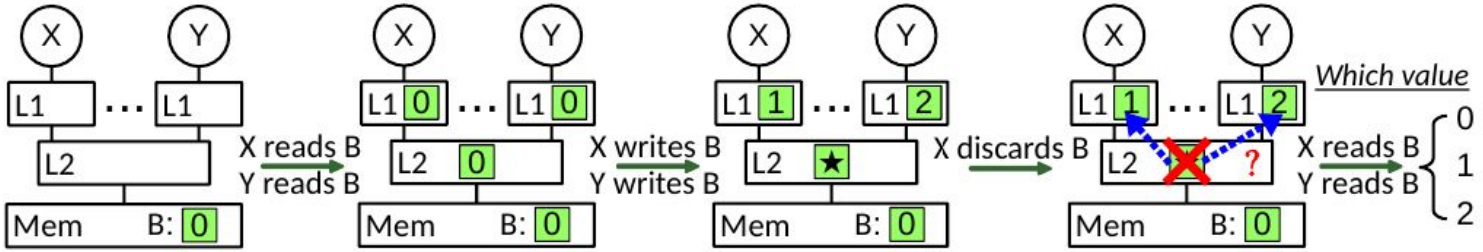
Write-Allocate: On a write-miss, the block is **fetched** from memory into cache *before* the write occurs.

No-Write-Allocate: On a write-miss, data is written **directly** to the next level, bypassing the current cache.

Experiments & Observations – Targets

Grade	GPU Model	Generation	SM Count	Memory Size	L2 Size	Release
Consumer	RTX 3060	Ampere	28	12GB	3MB	Jan 2021
	RTX 3080	Ampere	64	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

Experiment 1 – Inclusion Policy

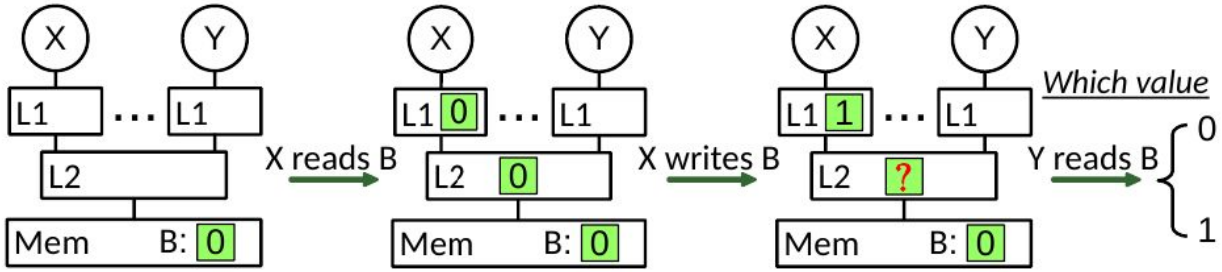


Result: X retrieves 0 & Y retrieves 2

Observation: 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.

It is clear that if an SM has a data block in its L1 cache, it will prioritize and read that value regardless of whether the data is outdated or has been modified elsewhere.

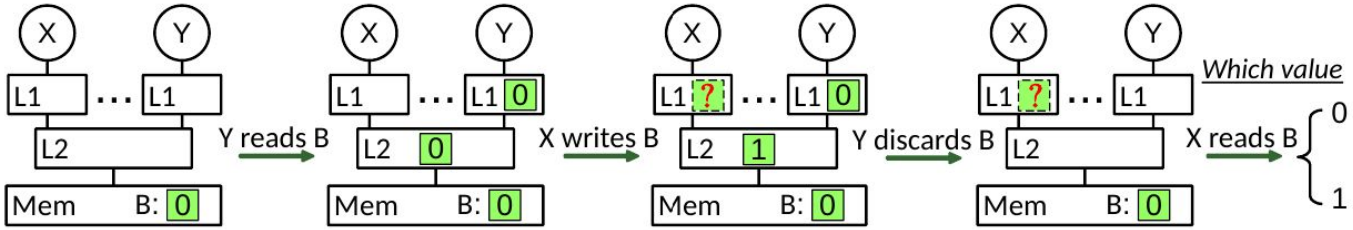
Experiment 2 – Write Policy



Result: Y always retrieves 1 regardless of the GPU and selected SMs.

Observation: the write policy employed by GPU L1 caches is write-through. Data is written to both L1 and L2 immediately, but not to GPU memory (write-through at L1, write-back at L2).

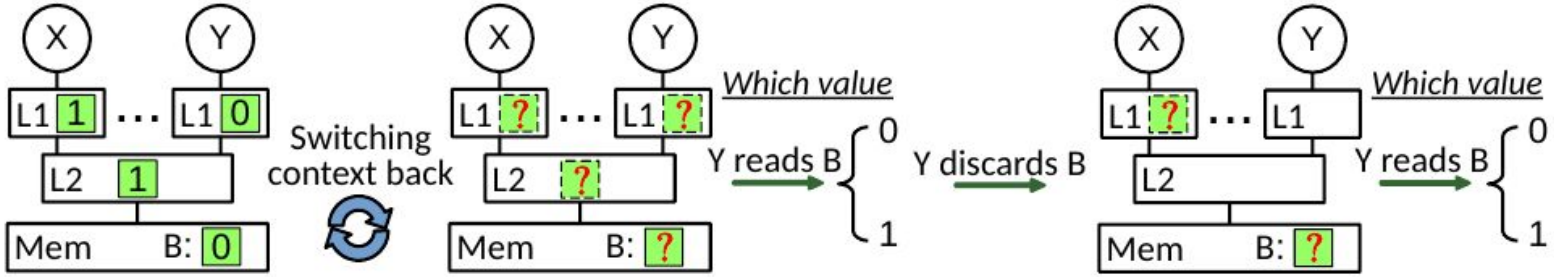
Experiment 3 – Write Allocation Policy



Result: X always reads 1.

Observation: 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.

Experiment 4 – L1 Cache Auto-Flushing



Result:

- Y before discard: 1
- Y after discard: 0

Observation: L2 is not flushed; otherwise, Y should have consistently seen the same value no matter if discarding B was performed or not.

Experiment 5 – Associativity and Replacement

Experiment Setup

- **Goal:** Determine L2 associativity and replacement dynamics.
- **Method:** Map congruent addresses $\{A_0, A_1, \ldots\}$ to a single L2 cache set.
- **Metric:** Test set capacity using **Store (st)** vs. **Load (ld)** operations.

Key Findings

- **Store-only:** Max 7 blocks retained (indicates a **"Dirty Line" limit**).
- **Load-only:** Max 16 blocks retained (confirms **16-way associativity**).
- **Observation:** NVIDIA restricts the dirty-to-clean ratio to < 50% in L2.
- **Replacement Policy:** Complex; primarily **LRU-based**, but evicts both the *Least Recently Used* (dirty) and *Least Recently Read* (clean) lines simultaneously.

Experiment 5 – Associativity and Replacement

The Challenge: 16-way associativity is not managed uniformly.

Priming the Set: A combination of **7 stores + 9 loads** effectively populates a cache set.

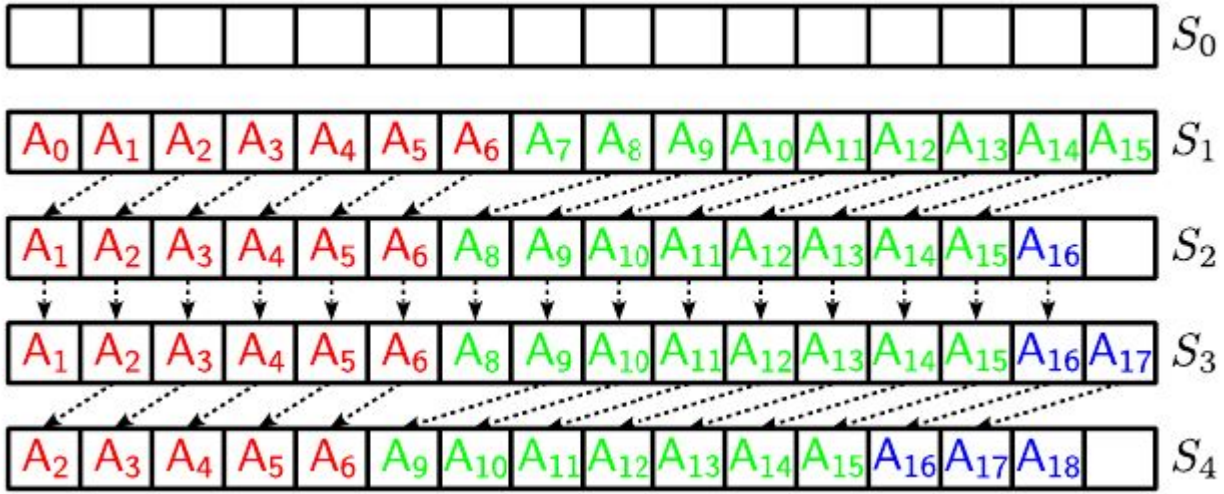
Dual Eviction Phenomenon:

- Accessing a new address (A_{16}) evicts **two** lines instead of one.
- It evicts the *Least Recently Used* (LRU) line **AND** the *Least Recently Read* (LRR) line.

Key Condition: This "Dual Eviction" only triggers if the LRU line is **dirty**.

Takeaway: L2 management is deterministic, allowing for a reliable "Eviction Set" for the attack.

Experiment 5 – Associativity and Replacement



Threat Model

Attacker Goal: Spy on a victim application's memory access patterns to leak sensitive information (e.g., website URLs, keystrokes).

Co-location: Attacker and victim execute on the same GPU, sharing the **Last-Level Cache (L2)** and **Main Memory**.

Capabilities:

- The attacker runs a non-privileged process (no "Root" or "Admin" required).
- Access to standard **CUDA/PTX** instructions, including discard.

Attack – Algorithm

1. Write new values to addresses $A_0,...,A_6$.
2. Read the values at addresses $A_7,...,A_{15}$.
3. Wait for the context to be switched away and back.
4. Execute the discard instruction on A_0.
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.

Attack – Algorithm

1. Write new values to addresses $A_0,...,A_6$.
2. Read the values at addresses $A_7,...,A_{15}$.
3. **Wait for the context to be switched away and back.**
4. Execute the discard instruction on A_0.
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.

Question: how do we detect a context switch?

Attack – Context Switch Detection (First Approach)

```
prev = 0;
start = clock64();
do {
    delta = clock64() - start;
    if (delta - prev > T)
        break;
    prev = delta;
} while (1);
```

Attack – Context Switch Detection (First Approach)

```
prev = 0;
start = clock64();
do {
    delta = clock64() - start;
    if (delta - prev > T)
        break;
    prev = delta;
} while (1);
```

It works well for Ampere GPUs, **but not for Ada Lovelace GPUs.**

Attack – Context Switch Detection (First Approach)

```
prev = 0;
start = clock64();
do {
    delta = clock64() - start;
    if (delta - prev > T)
        break;
    prev = delta;
} while (1);
```

It works well for Ampere GPUs, **but not for Ada Lovelace GPUs.**

Issue in Ada Lovelace: %clock64 **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.

Let's make it **better**

Attack – Context Switch Detection (Second Approach)

Setup: Two Streaming Multiprocessors (SMs) work in tandem:

- **Spy SM:** Executes the attack steps and monitors the victim.
- **Incrementer SM:** Continuously increments a flag variable in L2.

The "L1 Blindness" Effect:

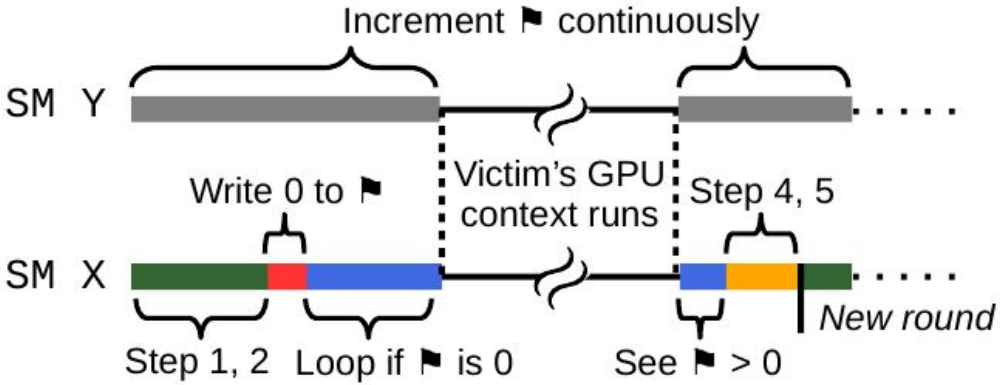
- The Spy SM writes 0 to the flag and enters a while(flag == 0) loop.
- Due to **L1 Incoherence**, the Spy reads the "stale" 0 from its local L1, even as the Incrementer updates the "true" value in L2.
- Key idea: context switch → **L1 cache is flushed**

Attack – Context Switch Detection (Second Approach)

The Trigger:

- A **Context Switch** (CS) triggers an automatic **L1 Flush**.
- This clears the stale 0 from the Spy's L1, forcing it to fetch the updated value (>0) from the L2 cache.

Execution: Once the loop breaks, the Spy immediately proceeds to the **Invalidate (Step 4)** and **Compare (Step 5)** phases.



Question: Is the integrity of cache sets guaranteed?
Spoiler alert: no.

Attack – Cache Integrity Problem

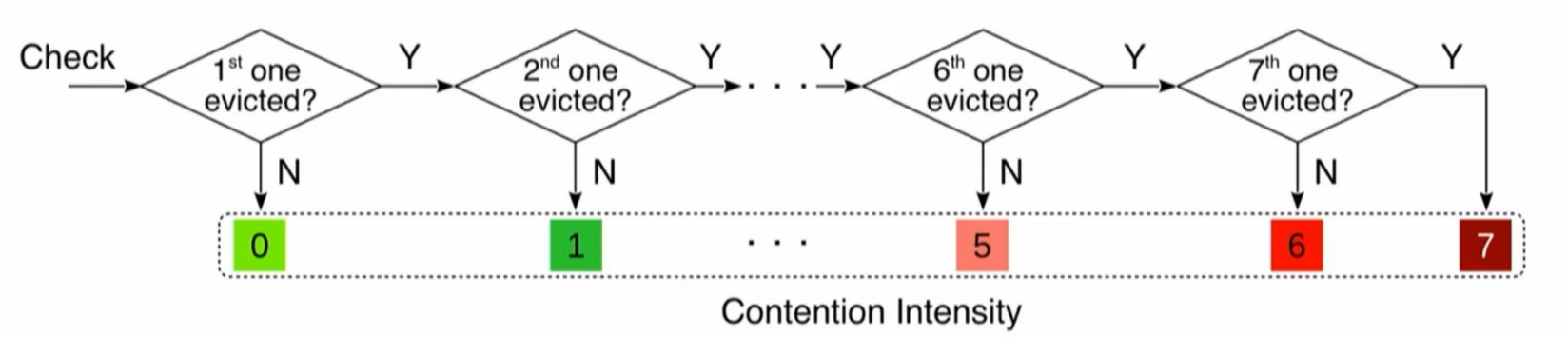
Noise (Spatial Overhead): Even a "dummy" program with an infinite loop causes some L2 activity during a context switch. This "noise" makes the cache set look "accessed" even if the victim didn't do anything.

Long Time Slices: GPU context switches don't happen every millisecond. The victim runs for a long time, meaning it will eventually touch almost every cache set.

Attack – Algorithm (Remake)

1. Write new values to addresses $A_0,...,A_6$.
2. Read the values at addresses $A_7,...,A_{15}$.
3. Wait for the context to be switched away and back.
4. **Execute the discard instruction on $A_0,...,A_6$.**
5. **Read the current value at $A_0,...,A_6$ 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.**

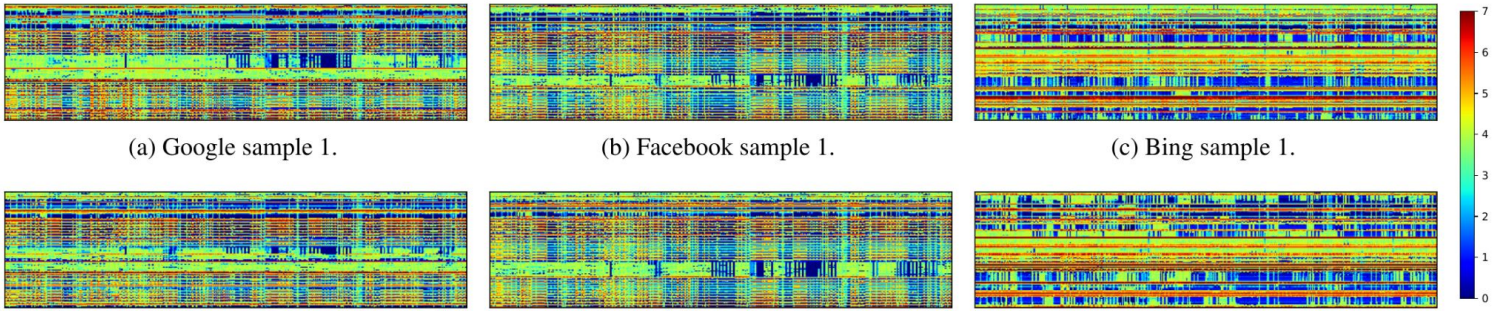
Attack – Algorithm (Remake)



The higher the value, **the higher volume of data** being loaded by the victim.

Case Study 1 – Website Fingerprint

Browsers use GPUs for **rendering web pages**. As a result, **GPU cache data comes from web pages.**



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

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

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

Case Study 1 – Website Fingerprint Evaluation

Using neural network models, they try to map the memorygrams into websites.

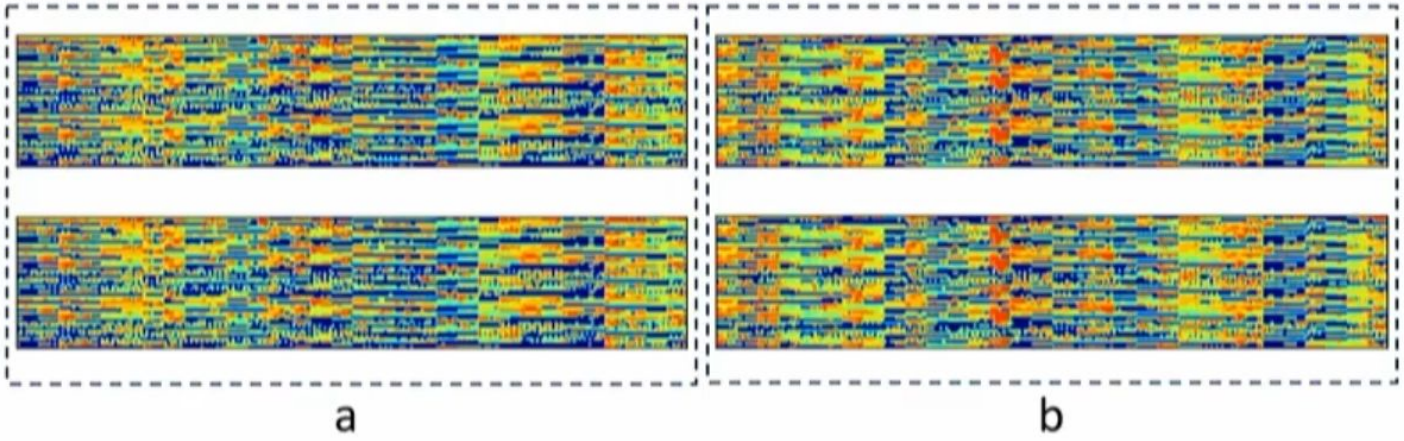
For each of 50 websites, they gather 100 memorygrams.

- They used DenseNet-121 model + 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%

Case Study 2 – Virtual Keystroke Extraction

When a key is pressed in virtual keyboard, it exhibits some local visual effects like shading the surroundings of the key. So pressing different keys, **leads to accessing different GPU memory addresses.**



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.

Case Study 2 – Virtual Keystroke Extraction Evaluation

They train a DenseNet-121 model with 300 samples for each character.

- Infer Ubuntu system login passwords

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

Conclusion

The authors unveiled previously unknown characteristics of NVIDIA GPU caches for the first time.

They have introduced a new cache attack primitive that enables spying on GPU cache activities without relying on timers.

They have showcased the use of this primitive with two case studies on NVIDIA's Ampere and Ada Lovelace GPUs

Thank you!
Questions?