ShadowScope: GPU Monitoring and Validation via Composable Side Channel Signals

Ghadeer Almusaddar*
galmusa1@binghamton.edu
Binghamton University
Binghamton, NY, USA

Yicheng Zhang*
yzhan846@ucr.edu
University of California, Riverside
Riverside, CA, USA

Saber Ganjisaffar
sganj003@ucr.edu
University of California, Riverside
Riverside, CA, USA

Barry Williams
bwilli33@binghamton.edu
Binghamton University
Binghamton, NY, USA

Yu David Liu
davidl@binghamton.edu
Binghamton University
Binghamton, NY, USA

Dmitry Ponomarev
dponomar@binghamton.edu
Binghamton University
Binghamton, NY, USA

Nael Abu-Ghazaleh
naelag@ucr.edu
University of California, Riverside
Riverside, CA, USA

Abstract

As modern systems increasingly rely on GPUs for computationally intensive tasks such as machine learning acceleration, ensuring the integrity of GPU computation has become critically important. Recent studies have shown that GPU kernels are vulnerable to both traditional memory-safety issues (*e.g.*, buffer overflow attacks) and emerging microarchitectural threats (*e.g.*, Rowhammer attacks), many of which manifest as anomalous execution behaviors observable through side-channel signals. However, existing golden model–based validation approaches that rely on such signals are fragile, highly sensitive to interference, and do not scale well across GPU workloads with diverse scheduling behaviors. To address these challenges, we propose *ShadowScope*, a monitoring and validation framework that leverages a *composable golden model*. Instead of building a single monolithic reference, *ShadowScope* decomposes trusted kernel execution into modular, repeatable functions that encode key behavioral features. This composable design captures execution patterns at finer granularity, enabling robust validation that is resilient to noise, workload variation, and interference across GPU workloads. To further reduce reliance on noisy software-only monitoring, we introduce *ShadowScope*$^+$, a hardware-assisted validation mechanism that integrates lightweight on-chip checks into the GPU pipeline. *ShadowScope*$^+$ achieves high validation accuracy with an average runtime overhead of just 4.6%, while incurring minimal hardware and design complexity. Together, these contributions demonstrate that side-channel observability can be systematically repurposed into a practical defense for GPU kernel integrity.

1 Introduction

Graphics Processing Units (GPUs) are ubiquitous in modern computing systems, powering everything from mobile devices to large-scale data centers [1, 3]. They accelerate data-intensive and compute-heavy workloads, ranging from multimedia applications to large language models (LLMs) such as ChatGPT [53] and LLaMA [35]. To support these demands, GPUs adopt a highly parallel execution model that launches massive numbers of threads [1, 7, 46]. Given the sensitive nature of many GPU workloads, ensuring the integrity of kernel execution has become increasingly critical. However, recent studies have shown that GPU kernel execution is vulnerable to both traditional memory-safety issues [11, 13, 15, 36, 55, 59, 71] and emerging microarchitectural attacks [14, 31, 39, 40, 43, 72, 73]. To protect GPU kernels, researchers have proposed a range of defenses, including both software- and hardware-based approaches [20, 21, 27, 64]. However, these solutions often suffer from high performance overhead and are limited in scope, addressing only a subset of attacks.

As an alternative, researchers have explored *golden model-based validation*, which compares a kernel's runtime behavior against a trusted reference derived from known good executions. Prior work has repurposed side channel signals, including performance counters and timing characteristics, and, in controlled laboratory settings, power or EM signals, to build such references [8, 32, 44, 58, 62]. These signals expose execution structure without intrusive software changes. Traditional golden model-based validation has so far been evaluated primarily on simple CPUs or small embedded systems in controlled environments, and for programs with invariant execution patterns. Translating these methods to GPUs is nontrivial. SIMT execution variability, the presence

*Both authors contributed equally to this research.

of many SMs, dynamic scheduling, and other sources of execution variability introduce distorts to traces, making it difficult to capture a single golden model.

Research challenges. Leveraging side-channel signals for kernel validation on GPUs introduces several challenges due to the complexity of GPU hardware. (1) Side-channel signals are fragile and noisy, and interference from concurrent processes can overwhelm the useful signal. Prior studies on small devices even report SNR values below 0.001 for power and EM measurements [28, 34], and GPU parallelism further complicates matters by overlapping thousands of threads and mixing their behaviors. This makes signal extraction on GPUs especially challenging. (2) Aligning observations with specific kernel executions is unreliable. Resource contention, kernel scheduling, and driver-level optimizations can shift or stretch execution phases, leading to misaligned traces [66, 68]. Such misalignment inflates false positives in golden model validation. (3) Execution patterns vary naturally. Differences across inputs and configurations can change memory access patterns or warp scheduling, which undermines the assumption of stable reference traces. This variability makes it hard to define accurate golden models that generalize across real workloads.

Our approach. To overcome these challenges, we propose *ShadowScope*, a GPU monitoring and validation framework based on **composable golden reference models**. In this design, the full GPU program execution is represented as a sequence of modular segments (*e.g.*, segmented along kernel invocations, CPU-GPU memory transfers, or intra-kernel phases). Each segment is validated independently rather than as part of a single monolithic trace. This segmentation reduces the impact of scheduling noise and input variability, making golden models more robust in highly parallel GPU environments. It also allows for each segment to be configured to allow for some variability in execution. *ShadowScope* builds on two key innovations. (1) Compositional validation. The golden reference is decomposed into segments defined by composable functions marking kernel boundaries or phases. Each segment captures a repeatable execution pattern and is validated independently, isolating variability and absorbing scheduling noise. An application is deemed valid only if all segments conform to their expected behavior. (2) Auxiliary instrumentation. The application is instrumented to emit lightweight markers as side-channel signals (more accurately, covert channel signals). These markers indicate segment boundaries and can encode contextual parameters such as kernel and/or input configurations. When these markers are received by a verifier, they enable it to align the observed traces to improve robustness under concurrency with minimal overhead. Communicating configuration information as part of the marker can also allow the verifier to dynamically match the appropriate reference golden model to the communicated kernel configuration parameters. We call this idea composable golden reference models.

We first implemented *ShadowScope* in software and evaluated it on two NVIDIA GPUs: Tesla V100 and GeForce RTX 4060. Using the NVIDIA CUPTI profiler API [47], we collect performance data such as timing, instruction counts, and memory usage during kernel execution to construct the golden reference model. We decomposed execution into segments corresponding to kernel invocations and inserted functions that communicate markers through the side channels at the kernel boundaries. To assess effectiveness, we tested *ShadowScope* against four representative GPU attacks: buffer overflows [15, 36], the mind-control attack [55], Rowhammer [38], and slowdown/DoS attacks [43, 63]. Across these cases, *ShadowScope* successfully detected anomalous execution with high accuracy, achieving up to a 100% true positive rate and as low as 0% false positives under controlled conditions. Moreover, our evaluation shows that the method is robust to noise and interference from other GPU workloads.

When implementing *ShadowScope* in software on NVIDIA GPUs, we identified several limitations in the existing Performance Monitoring Units (PMUs) that hinder effective attack detection. These include (1) low sampling rates, which reduce visibility into short-lived kernel behavior, (2) high profiling overhead and restrictions on event groupings imposed by current GPU profiling tools, and (3) Side channels from accessible performance counters: enabling access to performance counters is known to enable side-channel leakage [40]: for this reason, software access to performance counters is often disabled on GPUs. To address these gaps, we introduce *ShadowScope*$^+$, a lightweight hardware-assisted mechanism for performance monitoring and on-chip kernel validation. *ShadowScope*$^+$ supports higher sampling rates and isolated profiling events, while eliminating the need for CPU and driver intervention required in software-based validation. These enhancements enable *ShadowScope*$^+$ to effectively detect two primary classes of GPU attacks: kernel deviation and mind-control attacks (Section 5.4.1). We show that the performance and hardware complexity of *ShadowScope*$^+$ are small, making it a practical and efficient solution for securing GPU application execution (Sections 5.4.2 and 5.4.3).

In summary, the paper makes the following contributions:

- We present the **first** defense framework that leverages GPU side-channel signals to validate execution.
- We propose a *composable* golden modeling approach that enables robust validation by segmenting execution into independently verifiable units. These units of execution is demarcated via markers that enables the verifier to synchronize with the execution to tolerate variability in GPU parallel execution. The markers are communicated through the side-channel.
- We implement *ShadowScope* in software and show on NVIDIA GPUs that it can detect the presence of four representative attacks with high accuracy.

- To improve monitoring quality and lower its overhead, we design a hardware implementation that extends the PMU to implement the monitoring. We show that *ShadowScope*$^+$ is able to validate correct execution and identify two major classes of attacks.

2 Background

In this section, we first review the GPU execution model and then introduce the emerging control flow attacks on the GPU. After that, we describe the GPU Performance Monitoring Units (PMUs) that are accessible to developers and form the fundamental structure of *ShadowScope*.

2.1 GPU Execution Security

Unlike CPU processes, GPU kernels operate using the single instruction, multiple threads (SIMT) model, which enables the execution of thousands of threads in parallel. This improves computation efficiency for workloads such as 3D graphics rendering and deep learning. In this work, we focus specifically on NVIDIA GPUs.

GPU execution. The execution flow within the GPU begins with the CPU launching a *kernel* on the GPU, specifying the grid and block dimensions. A grid, consisting of multiple blocks, is divided into groups of threads. These blocks are then distributed across the streaming multiprocessors (SMs).Within each block, threads are organized into warps (32 threads) that execute the same instruction simultaneously. The warp scheduler in each SM selects which warps to execute, ensuring efficient parallel processing.

GPU kernel execution integrity. Ensuring GPU kernel execution integrity is vital for maintaining the correctness and security of GPU computations. This involves preserving the integrity of both the kernel code and the execution flow. It guarantees that the GPU performs intended operations without malicious interference.

Emerging attacks on GPU kernels. Recent memory corruption attacks, such as buffer overflow attacks, have been extensively demonstrated on GPUs [11, 36, 55]. These attacks target the crucial part of the GPU execution unit – the kernel. For instance, Miele et al. [36] and Di et al. [13] demonstrated heap and stack-based buffer overflow attacks on GPU kernels to corrupt data and manipulate the execution flow. Park et al. [55] corrupted GPU kernel executions in machine learning models, significantly reducing the accuracy of model predictions.

2.2 PMU on GPUs

GPU vendors like NVIDIA and AMD have introduced performance monitoring units similar to those on CPUs to help developers optimize application performance. On NVIDIA GPUs, these performance counters are accessed through the CUDA Profiling Tools Interface (CUPTI) [47]. On AMD GPUs, these counters can be analyzed via the GPU Performance API (GPUPerfAPI) [5].

PMU events and metrics. NVIDIA GPUs provide two kinds of counters to track the performance of CUDA applications: events and metrics. An event in CUDA CUPTI is a measurable activity that occurs during the execution of a CUDA application [50]. These events gather detailed performance metrics, helping to understand and optimize CUDA application performance. Events and metrics relate to various aspects of GPU architecture resources, including SM, L1/L2 cache, dynamic random access memory (DRAM), GPU interconnects (*e.g.*, NVLink and PCIe), and so on. Different events and metrics have varying sampling rates, and some can be profiled together while others cannot.

3 System Overview

In this section, we first describe our threat model and then elaborate on the design of our approach, utilizing side channel signals in GPUs to validate untested kernels.

3.1 Threat Model and Design Goals

We consider a victim and attacker who can launch kernels to run on GPUs. Victim kernels can be part of common applications that can be accelerated on GPUs, such as ML model training and inference [15, 18, 56] or HPC benchmarks [9, 12]. The attacker is an unprivileged remote user who can exploit a memory corruption vulnerability to change kernel control flow (e.g., buffer overflow) within the GPU kernels used by the victim application [11, 13, 15, 36]. The attacker may also attempt microarchitectural attacks such as side channel attacks [40, 43], rowhammer attacks [69], DoS attacks [70], which originate outside a victim application but try to compromise it by affecting its execution behavior.

Attacker's capabilities. We assume the attacker operates entirely within the GPU. The attacker can exploit existing memory vulnerabilities to alter the intended execution behavior of the victim's GPU kernels. The attacker does not need to tamper with the program executing on the host CPU. Instead, the attacker can launch malicious kernels on the GPU that run concurrently with the victim's kernels and attempt microarchitectural attacks to compromise them. We assume that the GPU's performance monitoring unit (PMU) is trusted and that all commands and data related to the PMU are protected from tampering. In other words, the attacker cannot modify the data collection process or the collected side-channel data, regardless of how many kernels they are able to launch. We monitor the periodic execution of victim kernels using the CUPTI Event API. Notably, CUPTI can only profile kernels that execute within the same context as the profiler [48].

Design goals. The goal of the validator of *ShadowScope* is to validate the execution behavior of victim GPU kernels that are vulnerable to existing GPU control flow attacks [11, 15,

36, 55, 59]. We collect side-channel leakages emitted during the execution of untested kernels and compare them against pre-generated golden reference traces. By analyzing deviations between the collected and golden traces, we check whether the execution behavior of the untested kernel has deviated from its execution path.

3.2 Overview of *ShadowScope*

Fig. 1 overviews the idea of composable golden models, the validation component of *ShadowScope* which involves four key components: untested kernels instrumented with composable functions f, a side-channel data collector through PMU, a set of pre-generated golden reference traces of kernels, and a validator.

We use composable functions to enhance the communication of side-channel information to the validator, supporting more effective golden model validation. This information helps target security-critical kernels more precisely and improves detection accuracy. In addition, it reduces the size of the golden model by focusing only on relevant execution segments. Composable functions can also convey important input parameters, which guide the selection of the appropriate golden model for a given execution context.

The workflow begins when an untested CUDA kernel is executed on the GPU. During execution ❶, the kernel, instrumented with composable functions, generates side-channel footprints that are captured for validation. The composable function f is used to mark the start and end boundaries of the untested kernel. Next, the validator's data collector captures the resulting side-channel signals from the PMU, including the footprints generated by the kernel instrumented with the composable function ❷. Based on the metadata associated with the execution, the validator then selects the corresponding golden reference trace from a pre-collected dataset ❸. Finally, statistical comparison algorithms are applied to compare the collected trace with the corresponding golden trace and determine whether the kernel's execution integrity has been compromised ❹. If the execution fails validation, it is flagged as anomalous and prevented from continuing on the GPU.

```
__global__ composable_function() {
  int old_val, new_val;
  old_val = atomicCounter;
  new_val = old_val + 1;
  __syncthreads();
  atomicCAS(&atomicCounter, old_val, new_val);
  __syncthreads(); }
void function() {
  int blocks(256,1,1), threads(32,1,1);
  // blocks and threads to execute composable_function
  int compsBlocks(40,1,1), compsThreads(4,1,1);
  // composable function integration
  composable_function<<<cBlocks,cThreads>>>();
  kernel_to_be_validated<<<blocks,threads>>>(arg1, arg2);
  // composable function integration
  composable_function<<<cBlocks,cThreads>>>(); }
```

Listing 1. Composable function design.

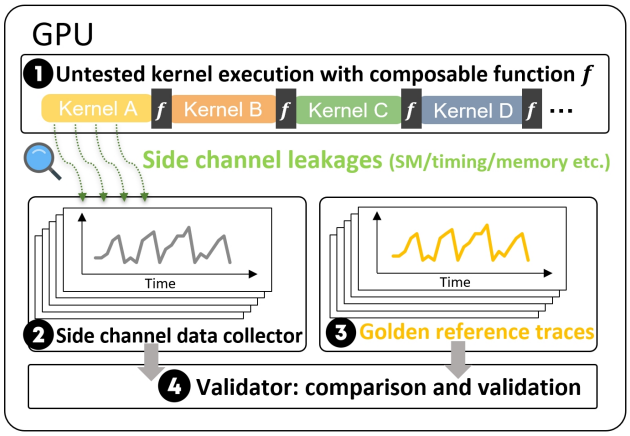

Figure 1. Overview of *ShadowScope* framework.

3.3 Instrumenting GPU Kernels with Composable Functions

Why insert composable functions? We insert composable functions f at the beginning and end of each GPU kernel to enhance the robustness of our golden model validation. This design choice is motivated by several reasons: First, these functions help the validator better align untested traces with their corresponding golden traces. Second, given that GPUs execute hundreds of kernels, precisely identifying the start and end of each kernel enables us to localize potential attacks more effectively. Furthermore, the composable functions facilitate the transmission of important kernel metadata—such as input size, block size, and grid size—through a covert channel, allowing the validator to select the appropriate golden reference trace for comparison. Finally, these functions must remain lightweight to avoid significantly altering the kernel's primary side-channel leakage characteristics.

Design of composable functions on NVIDIA GPUs. We base these composable functions on atomic operations because they are rarely used in common GPU benchmark kernels and reliably trigger correlated side-channel readings. To identify these functions better, we execute them with a prespecified number of threads and thread blocks. Composable function design can be adjusted based on the targeted kernel to be distinguishable from kernel execution. For example, we utilize the PMU event `global_atom_cas`, which counts the number of global atomic Compare-And-Swap operations performed on GPU memory [50]. This approach enables accurate tracking and monitoring of each kernel's execution boundaries. We explain the integration of composable functions in targeted trusted software in Listing 1.

Splitting side-channel traces based on composable functions. In addition to events used to validate the kernel, we collect events to detect composable functions. This set of events represents the execution behavior of trusted kernels. For instance, we collect one event group consisting of four

events: `instruction_executed`, `global_store`, `global_load`, and `global_atomic_cas`. The first event correlates with the SM level, representing the number of executed instructions in each SM. The `global_load` and `global_store` events identify the amount of data read from the GPU's global memory by CUDA threads during kernel execution. The last event tracks the number of times an atomic Compare-And-Swap (CAS) operation is executed in global memory. This atomic event helps identify the region of interest based on composable functions within each kernel.

As shown in Figure 2, the readings of `global_atom_cas` help identify the start and end of each layer or kernel execution in AlexNet. For simplicity, we only display the `instruction_executed` readings, which represent the total number of instructions executed across all SMs per sample.

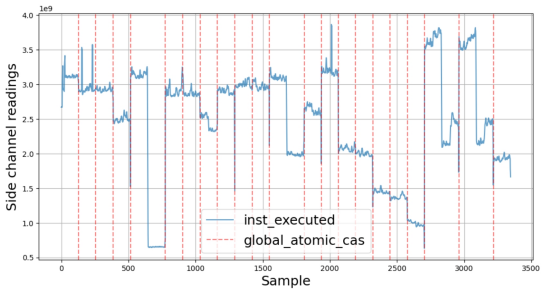

Figure 2. Splitting side-channel traces (AlexNet) based on `global_atom_cas` readings. Each split segment represents a single kernel/layer.

3.4 Side-channel Data Collector

Design of data collector on NVIDIA GPUs. We use the CUPTI API to collect data from performance counters in NVIDIA GPUs. Figure 3 explains the data collection approach using the CUPTI API. Both the targeted kernel and CUPTI profiler need to be in the same context to count the PMU data of the targeted kernel. To count PMU events using the CUPTI profiler, we set data collection mode to continuous using `cuptiSetEventCollectionMode`. A set of counters needs to be enabled and set to count specific events via `cuptiEventGroupCreate`, `cuptiEventGroupSetAttribute`, and `cuptiEventGroupEnable`. Then, all these events should be added to the **same events group** for event counts to be read together using `cuptiEventGroupAddEvent`. An event group is a collection of events that can be counted together. Not all supported events by NVIDIA GPU can be added to the same events group based on their type [48].

The profiler is set based on the targeted event group and data collection mode. During the execution of the targeted kernel, a CPU thread is used to read events counts per each sample from GPU for collection using `cuptiEventGroupReadAllEvents`. It is important to note that the sampling rate is affected by the type of the event being collected as we will show later.

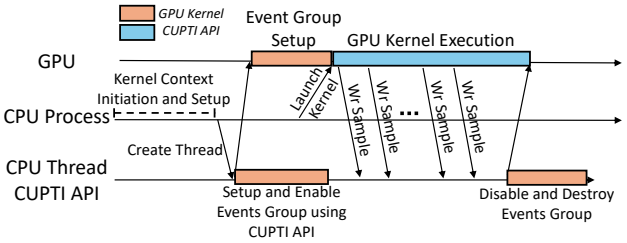

Figure 3. PMU data collection process using CUPTI API.

3.5 Golden Reference Model of GPU Kernels

The golden model is a conventional verification technique widely used throughout hardware development life cycles [16, 22, 37]. It verifies whether an IC design satisfies the required specifications. If the design fails to meet these specifications, it must be revised and re-verified until it passes all verification tests. In this work, we investigate whether golden model verification can be extended to GPU execution flows. Unlike traditional hardware systems, GPUs exhibit fundamentally different architectures and highly parallel execution patterns. To address these differences, we propose a novel golden model specifically tailored for GPU execution, focusing on the kernel, the smallest execution unit on a GPU.

The golden reference model is generated during the secure execution of all trusted kernels and serves as the baseline for validation. Importantly, these golden models only need to be constructed once based on trusted execution behavior.

3.6 Untested Kernel Validation

When collecting the execution behavior of an untested kernel using the side-channel data collector based on PMU, the resulting traces are compared against their corresponding golden model. With the help of composable functions embedded within the kernels, we can accurately determine the start and end of each kernel execution. This enables us to extract side-channel readings that precisely correspond to the regions of interest.

We use cross-correlation as the similarity metric to assess whether the kernel execution flow deviates from the expected behavior. This validation method has been successfully applied in prior work [62]. We choose cross-correlation for two main reasons: (1) it enables pattern matching even when traces are delayed or temporally misaligned, a common occurrence during side-channel data collection via the CUPTI API, because it inherently searches for the optimal lag alignment; and (2) it allows comparisons between traces of unequal lengths, which is critical in GPU environments where concurrent workloads can introduce timing variations and affect the sampling rate.

A signal segment is considered matched if the correlation coefficient exceeds 0.8. To detect attacks, we apply a rejection threshold based on consecutive mismatches. Specifically, *ShadowScope* flags a kernel as compromised only when four or more rejections occur in succession, allowing it to tolerate up to three consecutive rejections without raising a false alarm.

4 Software Instantiation of *ShadowScope* on NVIDIA GPUs

In this section, we first present the software implementation of *ShadowScope* on NVIDIA GPUs, including side-channel data collection, PMU event selection, and composable golden model generation. We then evaluate its security by implementing four GPU kernel attacks and show that *ShadowScope* successfully detects all attacks and identifies the compromised kernels.

4.1 Events Selection and Grouping

Unlike CPU workloads, GPU kernel execution involves multiple Streaming Multiprocessors (SMs), and different scheduling approaches. Some performance events are collected at the level of individual SMs, others capture the aggregate behavior of shared resources across several SMs. Another challenge with the NVIDIA GPU PMU, particularly when using the CUPTI API, is that not all hardware events can be grouped together [47, 50]. Based on our findings, events that are collected over the same number of instances (such as per SM or per group of SMs) can be grouped and collected together. For instance, on a Tesla V100 GPU with 80 SMs, the event `inst_executed` yields 80 readings, one per SM. In contrast, the event `fb_subp0_read_sectors` produces only 16 readings per sample, as each reading represents a group of 5 SMs. The NVIDIA Volta microarchitecture supports profiling 82 events. We categorize these events in Table 1.

Targeted GPU kernels. To evaluate our golden model approach *ShadowScope*, we targeted benchmarks from different suites. These benchmarks are written in the CUDA programming language. We categorize 18 benchmarks from Rodinia [9], CUDA-SDK [51], GraphBig [41] and four famous DNN models, summarized in Table 2.

4.2 Targeted Attacks

In this work, we evaluate *ShadowScope* against four types of attacks targeting GPU kernel execution and demonstrate its effectiveness in identifying abnormal or modified kernel execution flows. These attacks fall into the following categories.

Attack 1: buffer overflow attack. Buffer overflow attacks occur when an attacker overwrites the call stack's return address, redirecting the original program execution to malicious code. This type of attack was first introduced in prior work [15, 36]. It involves several steps. (1) The attacker exploits stack-based buffer overflow vulnerabilities in the CUDA kernel by providing input that exceeds the size of a fixed buffer, causing it to overwrite adjacent memory on the stack, such as function pointers. (2) The attacker crafts the input to overwrite these pointers with addresses of malicious code. (3) The payload is triggered by invoking the vulnerable kernel function. As a result, when the kernel executes, it uses the overwritten function pointers, which now point to the attacker's code. This causes the malicious payload to run instead of the intended function, allowing the attacker to take control of the CUDA application's execution flow.

Attack 2: Mind control attack. The mind control attacks [55] involve exploiting GPU memory vulnerabilities to undermine deep learning model performance. The attack process includes several key steps. (1) Setup: Attackers gain arbitrary code execution on the GPU by exploiting memory vulnerabilities in GPU kernels. This often involves hijacking the control flow of the GPU kernel using techniques like buffer overflow to overwrite function pointers. (2) Skipping GPU kernel execution: Attackers overwrite the identified GPU kernels, effectively converting them into `no-ops`. This manipulation degrades the deep learning model's accuracy, causing predictions to become no different from random guessing.

Attack 3: Rowhammer attack. Rowhammer is a hardware-level attack that rapidly accesses DRAM rows to induce bit flips in adjacent rows [38]. These bit flips can corrupt data or be exploited to bypass security mechanisms. Prior work [31, 69] has shown that such flips in DNN model weights can significantly degrade model performance. In this work, we aim to protect GPU memory against Rowhammer attacks. We launch such attacks on the GPU in two main steps: (1) evicting the L2 cache using the `discard` instruction [49] (available only on NVIDIA's Ampere and Ada architectures), and (2) repeatedly accessing the same DRAM bank on the GPU thousands of times to trigger potential bit flips.

Attack 4: DoS/Slow-down attack. Modern DRAM chips (e.g., DDR5 and GDDR5) have introduced a Refresh Management (RFM) interface to help mitigate Rowhammer attacks. However, recent work [42, 43, 63] has shown that RFM features can be exploited by attackers to intentionally trigger refresh activity and block DRAM banks, thereby slowing down co-located applications. These attacks are both effective, causing up to a 4.8× slowdown, and stealthy, as the attacker can activate only a single address in different RFM sub-banks to evade detection. Similar to traditional Rowhammer attacks, we simulate this attack on the GPU in two main steps: (1) flushing the L2 cache, and (2) randomly accessing DRAM addresses in RFM sub-banks to trigger slowdowns in victim applications.

Table 1. CUPTI Events in NVIDIA Volta.

Category	CUPTI Events
SM	active/elapsed_cycles_pm/sys/warps/sm, inst_executed/issued1/issued0, thread_inst_executed, sm_cta/warps_launched
Memory	fb_subp0/subp1_read/write_sectors
FP Unit	inst_executed_fma/fp16_pipe_s0/s1/s2/s3
L2 Cache	l2_subp0/subp1_read/write/total_sector_misses/queries, l2_subp0/subp1_write_sysmem_sector_queries
Tensor Cores	tensor_pipe_active_cycles_s0/s1/s2/s3
Trigger Unit	prof_trigger_00/01/02/03/04/05/06/07
Atomic Ops	atom_count, global_atom_cas, shared_atom, shared_atom_cas
Global Memory	global_load/store
Local Memory	local_load/store
Shared Memory	shared_load/store, shared_ld/st_bank_conflict, shared_ld/st_transactions
Texture	l2_subp0/subp1_read/write_tex_hit_sectors, l2_subp0/subp1_read/write_tex_sector_queries
PCIe	pcie_rx/tx_active_pulse
misc	generic_load/store

Table 2. Evaluated benchmarks.

Suite	Benchmarks
Rodinia	gaussian, heartwall, huffman, lud, myocyte, particlefilter, srad
CUDA SDK	matrixMul, vectorAdd, convolutionSeperable, histogram, sortingNetworks, fp16ScalarProduct
GraphBig	BetweennessCentr
DNN Models	CifarNet, AlexNet, SqueezeNet, ResNet-50

4.3 Evaluation Results

Experiment setup and data collection. We conduct Attacks 1 and 2 on an NVIDIA Tesla V100 GPU with driver version 535.183.01 and CUDA version 12.2. Attacks 3 and 4 are performed on a GeForce RTX 4060 with Samsung GDDR6 memory, using driver version 545.29.06 and CUDA version 12.3. To evaluate our defense method, we collect three datasets for each benchmark: golden, normal, and attack. Each dataset contains 100 traces. The golden and normal datasets are collected under benign conditions using only GPU benchmarks. In contrast, the attack dataset is recorded during active attack execution. We use the golden dataset to extract reference traces. Then, we evaluate detection performance using the normal and attack datasets. The golden model comparison algorithm is implemented in Python. True Positive Rate (TPR) and False Positive Rate (FPR) are used as evaluation metrics.

4.3.1 Evaluation of Buffer Overflow Attack (Attack 1). In this attack, the adversary exploits a buffer overrun to overwrite function pointers and redirect execution flow. Each call to a targeted function can be hijacked to invoke attacker-controlled code. To monitor the attack, we utilize four CUPTI events per SM: `global_load`, `global_store`, `inst_executed`, and `global_atom_cas`. Since the Tesla V100 has 80 SMs, each reading yields 320 samples. We use `global_atom_cas` as a marker to identify the start and end of each kernel.

Figure 4 presents the performance of *ShadowScope* in identifying buffer overflow attacks across selected benchmarks from the CUDA SDK and GraphBig suites. As shown, *ShadowScope* consistently achieves high TPR, with 3 out of 7 benchmarks reaching 100%, and the lowest still maintaining a strong 87% (*sortingNetworks*). The False Positive Rate (FPR) remains low overall, with 4 benchmarks recording 0%, and the highest FPR observed being 25% for *sortingNetworks*. The average TPR and FPR are 96% and 9%, respectively. The relatively high FPR observed in some benchmarks is caused by the limited sampling rate of the GPU's PMU. In some fast-executing kernels, such as those from *sortingNetworks* or *convolutionSeparable*, the PMU collects fewer than 20 readings per kernel. This low resolution makes it difficult to compare traces against the golden reference accurately. As a result, false positives increase. In Section 5, we propose a new GPU PMU design to address this limitation.

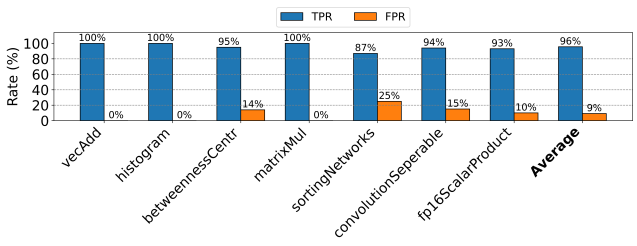

Figure 4. Performance of *ShadowScope* in monitoring buffer overflow attacks.

4.3.2 Evaluation of Mind Control Attack (Attack 2). In this attack, critical DNN layers or GPU kernels are either

skipped or replaced with no-op operations. To detect such behavior, we use the same four CUPTI events described in Section 4.3.1 to monitor GPU kernel executions.

Figures 5 and 6 compare the CUPTI event traces of normal and attack executions for CifarNet. In the normal trace (Figure 5), eight distinct kernel executions are observed. Each is clearly separated by spikes in the `global_atom_cas` event and exhibits consistent `inst_executed` activity. In contrast, the attack trace in Figure 6 deviates from this pattern. Although the `global_atom_cas` event still marks the kernel boundaries, the second kernel is missing. This layer-skipping attack shifts the execution sequence and introduces irregular fluctuations in `inst_executed` values as well. *ShadowScope* detects such anomalies by capturing these structural and behavioral inconsistencies.

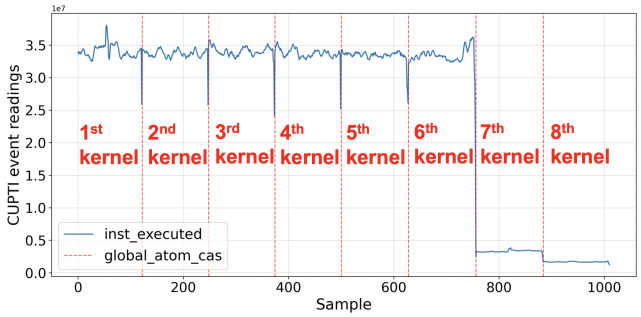

Figure 5. Side-channel signal of normal *CifarNet* execution.

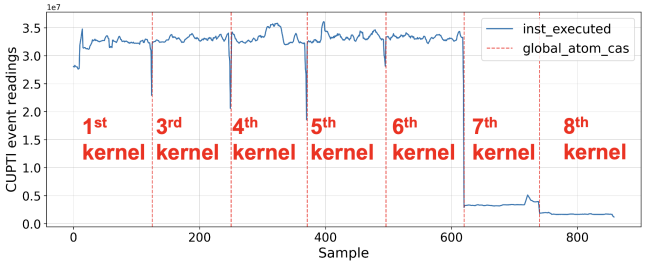

Figure 6. Side-channel signal of *CifarNet* under attack. **The second kernel is skipped.**

Figure 7 shows the detection performance of *ShadowScope* across four representative DNN architectures: CifarNet, AlexNet, SqueezeNet, and ResNet-50. *ShadowScope* achieves perfect detection (100% TPR) on both AlexNet and SqueezeNet, with zero or near-zero false positives (less than 1%). Detection on CifarNet and ResNet remains strong, reaching 92% and 89% TPR, respectively. However, both exhibit slightly higher FPRs of 4%. Overall, the system delivers consistent results, with an average TPR of 95% and an average FPR of only 2%.

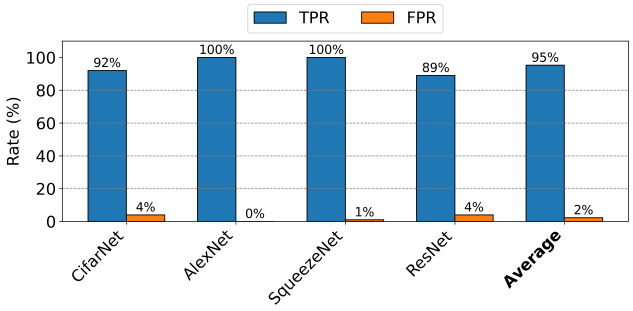

Figure 7. Performance of *ShadowScope* in monitoring mind control attacks.

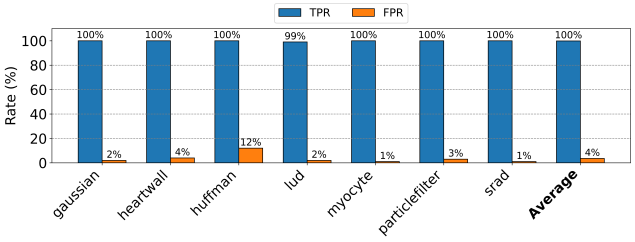

Figure 8. Performance of *ShadowScope* in monitoring rowhammer attacks.

4.3.3 Evaluation of Rowhammer Attack (Attack 3). In this attack, we demonstrate how *ShadowScope* can detect abnormal memory behavior, such as Rowhammer attacks, within the GPU's GDDR memory. To monitor memory activity, we use four CUPTI events: `fb_subp0_read_sectors`, `fb_subp1_read_sectors`, `l2_subp0_total_read_sector_queries`, and `l2_subp0_total_write_sector_queries`. These events help track GPU memory usage patterns at both the DRAM and L2 cache levels. Rowhammer attacks rely on frequent row access and L2 cache flushing. As a result, they produce distinct memory access patterns that *ShadowScope* can effectively capture.

Figure 8 shows the detection performance of *ShadowScope* across seven benchmarks from the Rodinia suite. *ShadowScope* achieves perfect detection (100% TPR) on six out of seven benchmarks, with *lud* slightly lower at 99% TPR. Most workloads exhibit low false positive rates, ranging from 1% to 4%, except for *huffman*, which records a higher FPR of 12%. We observe that when the sampling rate falls below 50 samples per kernel, detection accuracy decreases, which leads to more false positives in *huffman*. Overall, *ShadowScope* demonstrates strong and consistent performance across diverse workloads, achieving an average TPR of 100% and an average FPR of only 4%.

4.3.4 Evaluation of DoS/Slow-down Attack (Attack 4). In this attack, we use the same four memory-related CUPTI events described in Section 4.3.3 to detect abnormal GDDR memory usage by the adversary. These events help

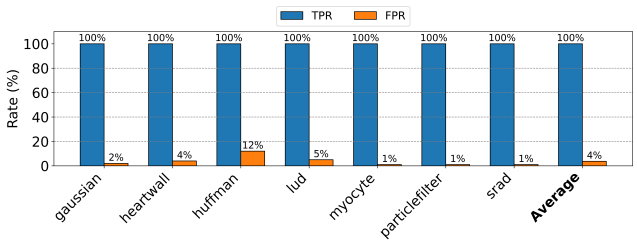

Figure 9. Performance of *ShadowScope* in monitoring DoS/Slow-down attacks.

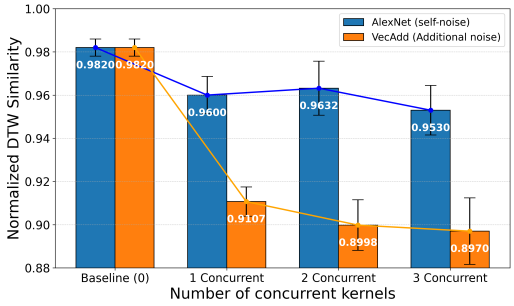

Figure 10. Impact of concurrent kernel noise (*ShadowScope*).

monitor low-level memory access behavior. Recent denial-of-service (DoS) or slowdown attacks [43, 63] on DDR memory exploit spoofed RFM to block access to DRAM banks. This behavior can severely degrade the performance of co-located applications. To succeed, the attack must both flush the L2 cache and randomly access DRAM addresses within RFM sub-banks.

Figure 9 presents the results of *ShadowScope* on seven benchmarks from the Rodinia suite. *ShadowScope* achieves perfect detection (100% TPR) across all benchmarks, highlighting its strong sensitivity to attack behavior. Most benchmarks show low FPRs, typically between 1% and 5%. However, *huffman* exhibits a higher FPR of 12%. Even so, the system delivers excellent overall performance, achieving an average TPR of 100% and an average FPR of just 4%.

4.4 Robustness to Noise

Since GPUs are designed for high parallelism and can run multiple programs concurrently, we evaluate how *ShadowScope* performs under kernel interference. To assess the impact of noise, we first collect 20 traces with only AlexNet running on the GPU. We then introduce two noise scenarios separately: concurrent execution of additional AlexNet models (self-noise), and concurrent execution of other benchmarks such as VecAdd (external noise). For each condition, we collect another 20 traces. We use a normalized DTW similarity score [61], where values closer to 1 indicate higher similarity, to compare noisy traces against the baseline.

Figure 10 shows the effect of concurrent kernel execution on normalized DTW similarity as measured by *ShadowScope*. We compare two scenarios: (1) AlexNet as self-noise, where the same model runs alongside itself, and (2) VecAdd as external noise, where a lightweight benchmark runs concurrently with AlexNet. In the baseline setting without concurrent kernels, the similarity score averages 0.9820. As the number of concurrent AlexNet kernels increases from one to three, similarity gradually declines. The drop is more pronounced in the VecAdd scenario, where the score falls to 0.8970. Although this value still indicates strong similarity, it may lead to a higher false positive rate. In comparison, the score under self-noise remains higher at 0.9530.

4.5 Limitation of Existing PMU

Even though *ShadowScope* performs well in most GPU benchmark scenarios, it depends heavily on NVIDIA's existing GPU Performance Monitoring Unit (PMU). However, the current PMU design presents several limitations that may hinder accurate kernel validation and anomaly detection. In this section, we summarize the key limitations of the existing GPU PMU.

Sampling rate. The sampling rate of NVIDIA CUPTI is relatively low, which can lead to insufficient data for fast-executing kernels. As a result, the golden model may fail to capture enough information to validate kernel execution accurately.

Events grouping. CUPTI enforces fixed event groupings, preventing the simultaneous collection of certain event combinations. This limits the ability to fully observe kernel behavior during execution.

Profiling overhead. CUPTI-based profiling introduces measurable runtime overhead. This can distort the behavior of lightweight kernels and reduce the fidelity of the collected traces.

Interference from concurrent kernels. When multiple kernels run concurrently, event counters may overlap or interfere with one another, making it difficult to isolate and attribute events to specific kernels.

To address these limitations, we propose *ShadowScope*$^+$, a hardware-assisted framework for ***in-GPU kernel validation***, as described in Section 5.

5 GPU-Centric *ShadowScope*

To address the challenges mentioned in Section 4.5, we propose *ShadowScope*$^+$, a hardware-assisted framework for ***in-GPU kernel validation*** that operates independently of the CPU. *ShadowScope*$^+$ leverages the GPU's existing hardware PMUs in conjunction with a dedicated on-chip validator to perform real-time validation of kernel execution. By localizing validation logic entirely within the GPU, *ShadowScope*$^+$ minimizes performance overhead and improves parallelism. Unlike CUPTI-based solutions, it also avoids the need for

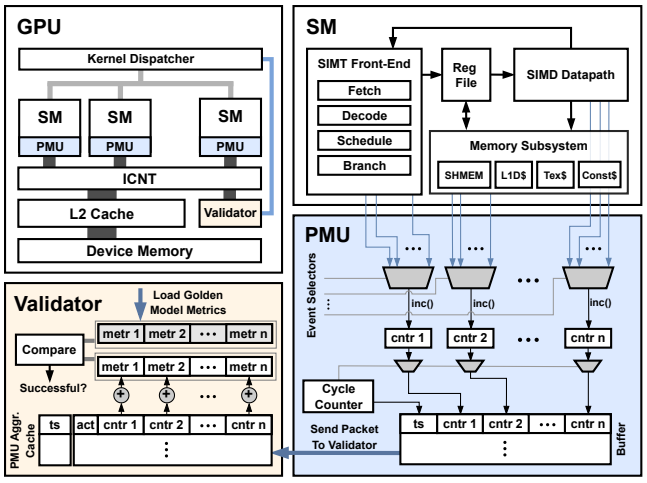

Figure 11. *ShadowScope*$^+$ high-level architecture.

privileged software components or driver-level modifications, thereby reducing the trusted computing base and easing deployment.

5.1 *ShadowScope*$^+$ Design

We equip each SM with a local PMU capable of independently collecting performance data, as illustrated in Figure 11. This is similar to *Streaming Multiprocessor Performance Counter (SMPC)* model in NVIDIA GPUs [52, 60]. In addition, we introduce an on-chip Validator module integrated into the GPU's Interconnection Network (*ICNT*) alongside other SMs and memory partitions (including L2 cache slices). This Validator is responsible for analyzing and validating the performance samples collected by the local PMUs.

5.1.1 Performance Monitoring Unit (PMU). As illustrated in Figure 11, each SM in the proposed design integrates a local PMU responsible for capturing microarchitectural events during kernel execution. The PMU receives a set of one-bit, cycle-wide signals from various SM components, each signaling the occurrence of specific events, such as instruction type counts, L1 data cache hits or misses, warp issues, or idle scheduler slots. These signals are routed to a set of configurable multiplexers, each of which selects one event signal to monitor. The selection logic is controlled via firmware-accessible registers, as commonly used by tools like NVIDIA Nsight [52] and CUPTI [47, 50]. Each multiplexer output is connected to a dedicated 32-bit up-counter that increments on every cycle the selected event signal is asserted. This straightforward yet effective design enables fine-grained, per-SM event tracking with low overhead and minimal disruption to kernel execution.

At the end of each sampling window, determined either by a firmware-configurable cycle counter register or by the end of the kernel execution, the PMU freezes all active counters and stores their values in a buffer entry, along with the corresponding timestamp or cycle count (*ts*), and resets the counters to zero in preparation for the next sampling window. In standard profiling mode (i.e., when kernel validation is disabled), a lightweight dedicated DMA engine transfers these buffer entries into a ring buffer located in the GPU's global device memory. This data can subsequently be used by existing profiling tools for performance analysis and reporting, or leveraged by our framework to construct a golden model representing the performance profile of a benign kernel execution. When kernel validation is enabled, however, *ShadowScope*$^+$ redirects the buffer entries to the Validator module over the GPU interconnection network, utilizing idle or underutilized bandwidth to minimize interference with the primary workload.

5.1.2 Validator Module. As performance samples are received from active PMUs, the Validator first consults a small PMU aggregation cache, which tracks the accumulation of performance counter values for each sampling window (identified by a timestamp) across active PMUs. Upon receiving a packet, the Validator uses the timestamp as a cache tag to look up the corresponding entry. If no entry exists, a new one is created, and the active PMU count (*act*) field is initialized based on the number of active PMUs for the currently executing kernel, as specified by the kernel dispatcher. If an entry is found, the incoming metric values are added to the existing values in the cache entry and the act count is decremented to reflect the arrival of data from one of the active PMUs. Once the *act* count reaches zero, indicating that all expected PMU samples for that window have been received, the Validator reads the accumulated values and updates a set of internal registers representing the total monitored metrics for that sampling window. These aggregated metrics are then compared against the preloaded golden model values by computing the distance between corresponding metric pairs, as illustrated in Figure 11. If the distance for these metrics exceeds a pre-defined threshold, tuned according to the sensitivity and semantics of the selected metrics, the kernel is flagged as potentially malicious. In such cases, the Validator signals the kernel dispatcher to stop the kernel execution and notifies the CPU through the GPU driver to initiate appropriate mitigation steps. If the comparison passes, indicating no malicious behavior, the Validator proceeds to monitor subsequent sampling windows.

5.2 *ShadowScope*$^+$ Mechanism

Figure 12 illustrates the operational flow of the proposed monitoring and validation mechanism in *ShadowScope*$^+$. At the time of kernel invocation, the golden model corresponding to Kernel 1 (K1) is loaded into the GPU's global device memory, where it is later accessed by the Validator for runtime comparison. Kernel execution begins when the kernel dispatcher assigns kernel to a set of active SMs. Upon dispatch, the dispatcher also notifies the Validator of the kernel launch and the associated active SMs. Each of these SMs

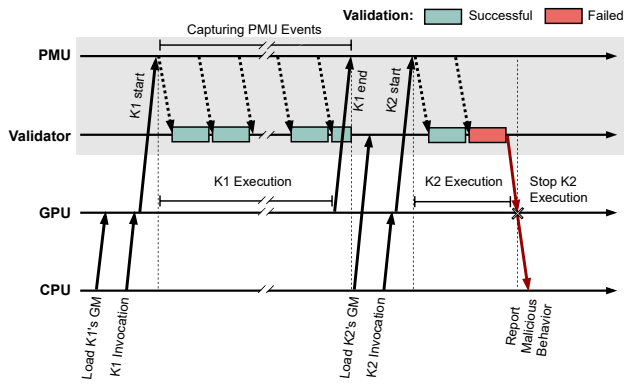

Figure 12. An overview of the *ShadowScope*$^+$ mechanism for in-GPU kernel validation.

activates its local PMU, which begins collecting microarchitectural performance events specific to the executing kernel.

During execution, each local PMU accumulates event samples and transmits them to the Validator at the end of every predefined sampling window, or upon kernel completion. The Validator then performs runtime validation by aggregating the metrics from all active PMUs and comparing them against the expected values from the golden model for the corresponding sampling window (as illustrated by the colored boxes on the Validator timeline in Figure 12). If the collected samples align with the golden model within acceptable thresholds, the kernel is classified as benign, and execution continues without interruption (as indicated by the green boxes in the timeline).

In contrast, if discrepancies are detected, such as deviations in control flow or memory access behavior, as in the case of Kernel 2 (K2), the Validator flags the violation by identifying significant deviations (as indicated by the red box in the timeline). It then signals the GPU to halt execution of the kernel and reports the anomaly and the offending kernel (K2) to the CPU via the GPU driver.

5.3 Simulation Methodology

To implement and evaluate our proposed design, we extend GPGPU-Sim v4.0 [24], a cycle-accurate simulator for NVIDIA GPU architectures. Our proof-of-concept is modeled on a small GPU configuration based on the NVIDIA Fermi architecture and the GTX480 platform. Detailed simulation parameters are provided in Table 3. For evaluation, we use four CUDA Samples benchmarks, *vectorAdd*, *matrixMul*, *histogram*, and *bitonicSort*, along with two DNN models, *AlexNet* and *CifarNet*. These workloads are used in Sections 5.4.1 and 5.4.2 to assess the effectiveness of the validation and detection mechanisms, as well as to measure performance overhead.

Table 3. Simulator configuration parameters.

Parameter	Specification
Number of SMs	15
SM Configuration	Warps/SM = 48, Schedulers/SM = 2, Warps/Scheduler = 16, Register File/SM = 32 KB
Execution Units	2 SPs, 1 SFUs
Shared Memory	48 kB, latency = 26 cycles
L1 Data Cache	16 KB, 8-sets, 16-ways
L2 Cache	786 KB, 64-sets, 16-ways, 6-banks
Min Access Latency	L1 = 35 cycles, L2 = 120 cycles
Memory Model	GDDR5, latency = 220 cycles
Frequency	SM,ICNT,L2:700 MHz, MEM:924 MHz

5.4 Evaluation Results

5.4.1 Security Evaluation. We evaluate the detection effectiveness of *ShadowScope*$^+$ in the presence of two known GPU attacks: buffer overflow and mind-control attacks. To quantify deviations in kernel execution, we employ normalized *Dynamic Time Warping* (DTW) scores to measure the similarity between benign and attack execution traces of the same kernel, captured using custom hardware PMUs and analyzed by the Validator proposed in *ShadowScope*$^+$.

Kernel Deviation Attacks. Table 4 presents the average and standard deviation of normalized DTW similarity scores between benign and attack kernel executions across four benchmarks. Lower scores indicate greater divergence from expected behavior; typically, a score below 0.1 suggests low similarity. Among the benchmarks, *matMul* shows the highest similarity score (0.0876), indicating minimal deviation, while *bitonicSort* and *vecAdd* exhibit the lowest scores (0.0178 and 0.0181, respectively), reflecting more substantial deviations caused by the attacks. Notably, *vecAdd* yields a standard deviation of 0, as it contains only a single executed kernel. These results demonstrate the effectiveness of our approach in identifying anomalous executions.

Table 4. Normalized DTW similarity scores between benign and attack traces.

Benchmarks	Similarity score	
	average	std
bitonicSort	0.0178	0.0151
histogram	0.0427	0.0315
matMul	0.0876	0.0088
vecAdd	0.0181	-

Mind Control Attacks. In this attack, we evaluate *AlexNet* and *CifarNet*. For AlexNet, using *ShadowScope*$^+$, we collect 10 segments of traces during benign execution, with each

segment corresponding to the execution of one layer or kernel. In the attack scenario, one layer is skipped, resulting in only 9 segments being collected. This allows us to detect that the second layer was skipped. Similarly, for CifarNet, we also detect that the second layer is missing, indicating a successful identification of the mind control attack.

5.4.2 Performance Evaluation. Figure 13 presents the performance overhead introduced by *ShadowScope*$^+$ compared to the baseline GPU architecture across the evaluated benchmarks. Since both PMU event monitoring and kernel validation occur off the critical path of kernel execution, their impact on performance is minimal. The primary source of overhead stems from the transfer of collected samples via the GPU's shared interconnection network by the local SM PMUs, and the loading of golden model samples by the Validator through shared memory channels into its local fetch buffer. This leads to contention on the interconnect bandwidth and memory channels, particularly in applications that heavily utilize these shared resources, such as memory-intensive workloads like DNN models. The results show that *ShadowScope*$^+$ incurs a geometric mean performance overhead of 4.6% in terms of normalized IPC compared to the baseline, with the lowest overhead observed in bitonicSort at 0.4%, and the highest in CifarNet at 9.2%.

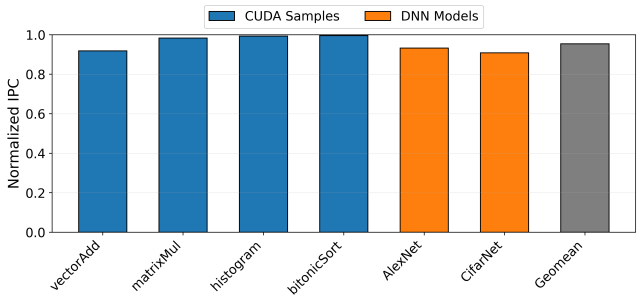

Figure 13. Performance normalized to baseline GPU.

5.4.3 Hardware Complexity Evaluation. For our hardware cost evaluation, each PMU is modeled with eight 32-bit counter registers and eight 8-to-1 multiplexers to monitor different microarchitectural events per counter. A 32-bit cycle counter generates timestamps for sampling windows. Each PMU includes an output buffer with eight 36-byte entries (288 bytes total) to store timestamps and associated metric values. The Validator's fetch buffer mirrors this structure but at half the size, four entries totaling 144 bytes, to hold golden model metrics loaded from the global device memory of the GPU. The PMU aggregation buffer is implemented as a simple cache with 33-byte blocks containing eight metrics, an 8-bit active-SM counter value, and a 32-bit timestamp as the tag. Aggregation is performed using eight 32-bit adders, while the Validator's comparison logic consists of eight subtractors, eight magnitude comparators, and a single threshold register to compute deviations from the golden model.

Table 5. Area and power overhead (percentage increase from baseline) of integrating PMUs into each SM (reported per SM and per GPU) and the Validator (reported relative to the entire GPU).

Component	Metric	Per SM	Per GPU
PMU	Area Overhead (%)	0.03	0.02
	Static Power (%)	0.07	0.13
	Dynamic Power (%)	0.28	0.26
Validator	Area Overhead (%)	-	0.004
	Static Power (%)	-	0.01
	Dynamic Power (%)	-	0.02
	Total Area Overhead (%)		0.03
	Total Static Power (%)		0.13
	Total Dynamic Power (%)		0.28

We evaluated SRAM-based structures using CACTI [30] using 40nm technology, the same technology used by AccelWattch [23], for modeling the NVIDIA GTX480 GPU. Key logic components, including counters, multiplexers, and comparators, were implemented in Verilog and synthesized using Synopsys Design Compiler 2017.09. Table 5 reports the overhead of a single PMU relative to an SM, the cumulative overhead of all PMUs relative to the full GPU, and the Validator's cost as a global module relative to the full GPU.

6 Related Work

Golden model validation. A variety of approaches have been proposed for hardware trojan detection using golden models, which are constructed from trusted hardware side-channel data collected through physical sources such as power, leakage current, and temperature [17, 26, 33, 57].

Attacks detection and software validation using hardware performance counter (HPC). HPCs collected from the PMU have been widely used to detect malware and side-channel attacks by identifying deviations in software behavior. Demme at al. [10] explored the feasibility of using HPC for malware and side channel detection in both ARM and Intel CPUs. Lee et al. [29] proposed using HPC to detect Spectre attacks by randomly selecting detectors, feature sets, and sampling periods to improve robustness against sophisticated threats. Khasawneh et al. [25] focused on detecting evasive malware by designing a system with multiple detectors and randomly selecting among them, making it difficult for attackers to evade detection. Most prior work using HPCs has focused on CPU-based systems and malware detection, with limited emphasis on software validation and little attention to attacks targeting GPU kernels.

Control flow attacks targeting GPU kernels. Park et al. [55] proposed reducing the prediction accuracy of DNN models by accessing arbitrary memory locations used by targeted ML models. The attack is based on a buffer overflow attack which is used to inject random code to lower

the accuracy of ML models. Recently, Guo et al. [15] investigated buffer overflow vulnerabilities in NVIDIA GPUs and demonstrated traditional code injection attacks and ROP-style attacks are possible on GPUs. Roels et al. [59] adapted CPU-style code reuse attacks to NVIDIA GPUs, enabling the discovery of new ROP gadgets and the construction of Turing-complete ROP attacks on GPUs.

Exploiting PMU as a side channel in GPUs. By collecting data from GPU PMU during victim kernel execution, attackers can leak secret information about the victim. Naghibijouybari et al. [40] exploited performance counters reflecting shared resource usage between victim and attacker to perform a series of side-channel attacks. Wei et al. [67] exploited data collected from GPU PMU to monitor context switches during DNN model training to learn targeted model layers and parameters. Wang et al. [65] also targeted DNN to perform model extraction attacks and mainly targeted events related to a unified memory management system which they found to have higher effectiveness.

Software and hardware defenses in GPUs. Hardware and software-based approaches propose the implementation of TEEs in GPUs. Volos et al. [64] to secure GPU kernels with no modification to CPU. Their approach only changes the command processor within GPU for TEE support. Jang et al. [21] also proposed TEE support for GPU. Their design requires modifications to the I/O interconnect and changes to the GPU driver to be included within CPU TEE. NVIDIA H100 includes hardware support for confidential computing (CC) targeting virtualized environment [45]. To isolate a virtual machine (VM), H100 CC requires support for TEE at the CPU side as well. TEE at the CPU side can be achieved through Intel TDX [19], AMD SEV-SNP [4], or ARM CCA [6].

Software attestation using side channels. Existing work also proposed using side channels such as power and electromagnetic signals for good use such as software attestation. Side channel data of program execution based on electromagnetic signals are used to detect deviations in program execution [44, 62]. This research targeted embedded systems and assumes predictable execution; ShadowScope extends the range of such approaches using the idea of composable verification. A line of research uses performance counters on CPUs to classify programs as benign or malicious [2, 25, 54]; however, these approaches provide a weaker protection since they do not verify the correct execution of a program.

7 Concluding Remarks

We present *ShadowScope*, a robust monitoring and validation framework for GPU kernel execution based on composable golden models derived from side-channel signals. By capturing modular and repeatable execution patterns, *ShadowScope* effectively detects both anomalous behavior within trusted kernels and signs of kernel compromise. It detects four types of attacks with up to 100% true positive rate and low false positive rates. To further reduce noise sensitivity and performance overhead, we introduce *ShadowScope*$^{+}$, a lightweight hardware-assisted mechanism for on-chip validation, achieving accurate detection with only 4.6% performance overhead.

References

[1] Tor M. Aamodt, Wilson Wai Lun Fung, and Timothy G. Rogers. General-purpose graphics processor architectures, 2018.

[2] Samira Mirbagher Ajorpaz, Daniel Moghimi, Jeffrey Neal Collins, Gilles Pokam, Nael Abu-Ghazaleh, and Dean Tullsen. Evax: Towards a practical, pro-active & adaptive architecture for high performance & security. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1218–1236. IEEE, 2022.

[3] Amazon. What is a GPU? https://aws.amazon.com/what-is/gpu/. Last accessed on: 07/31/2024.

[4] AMD. AMD Secure Encrypted Virtualization (SEV). https://www.amd.com/en/developer/sev.html. Last accessed on: 05/25/2025.

[5] AMD. GPUPerfAPI. https://gpuopen.com/gpuperfapi/. Last accessed on: 07/27/2024.

[6] arm. Arm Confidential Compute Architecture. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture. Last accessed on: 05/25/2025.

[7] Caroline Collange. GPU architecture: Revisiting the SIMT execution model. http://www.irisa.fr/alf/downloads/collange/cours/hpca2020_gpu_0.pdf. Last accessed on: 07/31/2024.

[8] Alexander Cathis, Mulong Luo, Mohit Tiwari, and Andreas Gerstlauer. Lapd: Lifecycle-aware power-based malware detection. *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2025.

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009. doi:10.1109/IISWC.2009.5306797.

[10] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, page 559–570, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2485922.2485970.

[11] Bang Di, Jianhua Sun, and Hao Chen. A study of overflow vulnerabilities on gpus. In *Network and Parallel Computing: 13th IFIP WG 10.3 International Conference, NPC 2016, Xi'an, China, October 28-29, 2016, Proceedings*, page 103–115, Berlin, Heidelberg, 2016. Springer-Verlag. doi:10.1007/978-3-319-47099-3_9.

[12] Peter Eastman, Jason Swails, John D Chodera, Robert T McGibbon, Yutong Zhao, Kyle A Beauchamp, Lee-Ping Wang, Andrew C Simmonett, Matthew P Harrigan, Chaya D Stern, et al. Openmm 7: Rapid development of high performance algorithms for molecular dynamics. *PLoS computational biology*, 13(7):e1005659, 2017.

[13] Christopher Erb, Mike Collins, and Joseph L. Greathouse. Dynamic buffer overflow detection for GPGPUs. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 61–73, 2017. doi:10.1109/CGO.2017.7863729.

[14] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand pwning unit: Accelerating microarchitectural attacks with the gpu. In *2018 ieee symposium on security and privacy (sp)*, pages 195–210. IEEE, 2018.

[15] Yanan Guo, Zhenkai Zhang, and Jun Yang. GPU Memory Exploitation for Fun and Profit. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4033–4050, 2024.

[16] Sunny L He, Natalie H Roe, Evan Wood, Noel M Nachtigal, and Jovana Helms. Model of the product development lifecycle. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2015.

[17] Kangqiao Hu, Abdullah Nazma Nowroz, Sherief Reda, and Farinaz Koushanfar. High-sensitivity hardware trojan detection using multimodal characterization. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1271–1276, 2013. doi:10.7873/DATE.2013.263.

[18] Xing Hu, Ling Liang, Shuangchen Li, Lei Deng, Pengfei Zuo, Yu Ji, Xinfeng Xie, Yufei Ding, Chang Liu, Timothy Sherwood, et al. Deepsniffer: A dnn model extraction framework based on learning architectural hints. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 385–399, 2020.

[19] intel. Intel® Trust Domain Extensions. https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html. Last accessed on: 05/25/2025.

[20] Andrei Ivanov, Benjamin Rothenberger, Arnaud Dethise, Marco Canini, Torsten Hoefler, and Adrian Perrig. {SAGE}: Software-based attestation for {GPU} execution. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 485–499, 2023.

[21] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. Heterogeneous isolated execution for commodity gpus. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 455–468, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3297858.3304021.

[22] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. {TheHuzz}: Instruction fuzzing of processors using {Golden-Reference} models for finding {Software-Exploitable} vulnerabilities. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3219–3236, 2022.

[23] Vijay Kandiah, Scott Peverelle, Mahmoud Khairy, Junrui Pan, Amogh Manjunath, Timothy G Rogers, Tor M Aamodt, and Nikos Hardavellas. Accelwattch: A power modeling framework for modern gpus. In *MICRO-54: 54th Annual IEEE/ACM International symposium on microarchitecture*, pages 738–753, 2021.

[24] Mahmoud Khairy, Zhesheng Shen, Tor M Aamodt, and Timothy G Rogers. Accel-sim: An extensible simulation framework for validated gpu modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 473–486. IEEE, 2020.

[25] Khaled N. Khasawneh, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Lei Yu. Rhmd: Evasion-resilient hardware malware detectors. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 315–327, 2017.

[26] Charles Lamech, James Aarestad, Jim Plusquellic, Reza Rad, and Kanak Agarwal. Rebel and tdc: Two embedded test structures for on-chip measurements of within-die path delay variations. In *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 170–177, 2011. doi:10.1109/ICCAD.2011.6105322.

[27] Jaewon Lee, Yonghae Kim, Jiashen Cao, Euna Kim, Jaekyu Lee, and Hyesoon Kim. Securing gpu via region-based bounds checking. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 27–41, 2022.

[28] Itamar Levi, Davide Bellizia, David Bol, and François-Xavier Standaert. Ask less, get more: Side-channel signal hiding, revisited. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(12):4904–4917, 2020.

[29] Congmiao Li and Jean-Luc Gaudiot. Detecting spectre attacks using hardware performance counters. *IEEE Transactions on Computers*, 71(6):1320–1331, 2022. doi:10.1109/TC.2021.3082471.

[30] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B Brockman, and Norman P Jouppi. Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques. In *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 694–701. IEEE, 2011.

[31] Chris S Lin, Joyce Qu, and Gururaj Saileshwar. Gpuhammer: Rowhammer attacks on gpu memories are practical. *arXiv preprint arXiv:2507.08166*, 2025.

[32] Yannan Liu, Lingxiao Wei, Zhe Zhou, Kehuan Zhang, Wenyuan Xu, and Qiang Xu. On code execution tracking via power side-channel. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1019–1031, 2016.

[33] Yu Liu, Yier Jin, and Yiorgos Makris. Hardware trojans in wireless cryptographic ics: Silicon demonstration & detection method evaluation. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 399–404, 2013. doi:10.1109/ICCAD.2013.6691149.

[34] Stefan Mangard. Hardware countermeasures against dpa–a statistical analysis of their effectiveness. In *Topics in Cryptology–CT-RSA 2004: The Cryptographers' Track at the RSA Conference 2004, San Francisco, CA, USA, February 23-27, 2004, Proceedings*, pages 222–235. Springer, 2004.

[35] Meta. Llama. https://llama.meta.com. Last accessed on: 07/31/2024.

[36] Andrea Miele. Buffer overflow vulnerabilities in cuda: a preliminary analysis. *Journal of Computer Virology and Hacking Techniques*, 12:113–120, 2016.

[37] A Molina and Oswaldo Cadenas. Functional verification: Approaches and challenges. *Latin American applied research*, 37(1):65–69, 2007.

[38] Onur Mutlu and Jeremie S Kim. Rowhammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(8):1555–1571, 2019.

[39] Hoda Naghibijouybari, Khaled N Khasawneh, and Nael Abu-Ghazaleh. Constructing and characterizing covert channels on gpgpus. In *Proceedings of the 50th annual IEEE/ACM international symposium on microarchitecture*, pages 354–366, 2017.

[40] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: Gpu side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 2139–2153, 2018.

[41] Lifeng Nai, Yinglong Xia, Ilie G Tanase, Hyesoon Kim, and Ching-Yung Lin. Graphbig: understanding graph computing in the context of industrial solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[42] Ravan Nazaraliyev, Saber Ganjisaffar, Nurlan Nazaraliyev, and Nael Abu-Ghazaleh. Practical: Subarray-level counter update and bank-level recovery isolation for efficient prac rowhammer mitigation. *arXiv preprint arXiv:2507.18581*, 2025.

[43] Ravan Nazaraliyev, Yicheng Zhang, Sankha Baran Dutta, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. Not so refreshing: Attacking gpus using rfm rowhammer mitigation. In *34th USENIX Security Symposium (USENIX Security 25)*, 2025.

[44] Alireza Nazari, Nader Sehatbakhsh, Monjur Alam, Alenka Zajic, and Milos Prvulovic. Eddie: Em-based detection of deviations in program execution. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 333–346, 2017.

[45] NVIDIA. Confidential Compute on NVIDIA Hopper H100. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/HCC-Whitepaper-v1.0.pdf. Last accessed on: 05/25/2025.

[46] NVIDIA. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Last accessed on: 07/31/2024.

[47] NVIDIA. CUPTI. https://docs.nvidia.com/cupti/. Last accessed on: 07/16/2024.

[48] NVIDIA. CUPTI Event API. https://docs.nvidia.com/cupti/api/group__CUPTI__EVENT__API.html. Last accessed on: 06/04/2025.

[49] NVIDIA. Parallel Thread Execution. https://docs.nvidia.com/cuda/parallel-thread-execution/.

[50] NVIDIA. CUPTI: User Guide. https://docs.nvidia.com/cuda/archive/11.0_GA/cupti/pdf/Cupti.pdf, 2022. Last accessed on: 07/18/2024.

[51] NVIDIA. NVIDIA CUDA samples. https://github.com/NVIDIA/cuda-samples, 2024.

[52] NVIDIA Corporation. Nsight Visual Studio Edition 4.6 User Guide: Performance Counters. https://docs.nvidia.com/nsight-visual-studio-edition/4.6/Content/Analysis/Report/CudaExperiments/KernelLevel/PerformanceCounters.htm, 2024. Accessed on 05/29/2025.

[53] OpenAI. Introducing ChatGPT. https://openai.com/index/chatgpt/. Last accessed on: 07/31/2024.

[54] Meltem Ozsoy, Khaled N Khasawneh, Caleb Donovick, Iakov Gorelik, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Hardware-based malware detection using low-level architectural features. *IEEE Transactions on Computers*, 65(11):3332–3344, 2016.

[55] Sang-Ok Park, Ohmin Kwon, Yonggon Kim, Sang Kil Cha, and Hyunsoo Yoon. Mind control attack: Undermining deep learning with gpu memory exploitation. *Computers & Security*, 102:102115, 2021. URL: https://www.sciencedirect.com/science/article/pii/S0167404820303886, doi:10.1016/j.cose.2020.102115.

[56] Kartik Patwari, Syed Mahbub Hafiz, Han Wang, Houman Homayoun, Zubair Shafiq, and Chen-Nee Chuah. Dnn model architecture fingerprinting attack on cpu-gpu edge devices. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 337–355. IEEE, 2022.

[57] Reza Rad, Jim Plusquellic, and Mohammad Tehranipoor. Sensitivity analysis to hardware trojans using power supply transient signals. In *2008 IEEE International Workshop on Hardware-Oriented Security and Trust*, pages 3–7, 2008. doi:10.1109/HST.2008.4559037.

[58] Reza M Rad, Xiaoxiao Wang, Mohammad Tehranipoor, and Jim Plusquellic. Power supply signal calibration techniques for improving detection resolution to hardware trojans. In *2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 632–639. IEEE, 2008.

[59] Jonas Roels, Adriaan Jacobs, and Stijn Volckaert. Cuda, woulda, shoulda: Returning exploits in a sass-y world. In *Proceedings of the 18th European Workshop on Systems Security*, EuroSec'25, page 40–48, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3722041.3723099.

[60] Alvaro Saiz, Pablo Prieto, Pablo Abad, Jose Angel Gregorio, and Valentin Puente. Top-down performance profiling on nvidia's GPUs. In *2022 IEEE international parallel and distributed processing symposium (IPDPS)*, pages 179–189. IEEE, 2022.

[61] Stan Salvador and Philip Chan. Toward accurate dynamic time warping in linear time and space. *Intelligent data analysis*, 11(5):561–580, 2007.

[62] Nader Sehatbakhsh, Alireza Nazari, Haider Khan, Alenka Zajic, and Milos Prvulovic. Emma: Hardware/software attestation framework for embedded systems using electromagnetic signals. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 983–995, 2019.

[63] Hritvik Taneja and Moinuddin Qureshi. Roguerfm: Attacking refresh management for covert-channel and denial-of-service. *arXiv preprint arXiv:2501.06646*, 2025.

[64] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on GPUs. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 681–696, Carlsbad, CA, October 2018. USENIX Association. URL: https://www.usenix.org/conference/osdi18/presentation/volos.

[65] Zhendong Wang, Xiaoming Zeng, Xulong Tang, Danfeng Zhang, Xing Hu, and Yang Hu. Demystifying arch-hints for model extraction: An attack in unified memory system, 2022. URL: https://arxiv.org/abs/2208.13720, arXiv:2208.13720.

[66] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. Simultaneous multikernel: Fine-grained sharing of gpus. *IEEE Computer Architecture Letters*, 15(2):113–116, 2015.

[67] Junyi Wei, Yicheng Zhang, Zhe Zhou, Zhou Li, and Mohammad Abdullah Al Faruque. Leaky DNN: Stealing deep-learning model secret with gpu context-switching side-channel. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 125–137. IEEE, 2020.

[68] Qiumin Xu, Hyeran Jeon, Keunsoo Kim, Won Woo Ro, and Murali Annavaram. Warped-slicer: Efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming. *ACM SIGARCH Computer Architecture News*, 44(3):230–242, 2016.

[69] Fan Yao, Adnan Siraj Rakin, and Deliang Fan. {DeepHammer}: Depleting the intelligence of deep neural networks through targeted chain of bit flips. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1463–1480, 2020.

[70] Wei Zhang. Defend GPUs against DoS attacks . In *2013 IEEE 32nd International Performance Computing and Communications Conference (IPCCC)*, pages 1–2, Los Alamitos, CA, USA, December 2013. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/PCCC.2013.6742758, doi:10.1109/PCCC.2013.6742758.

[71] Yicheng Zhang, Ravan Nazaraliyev, Sankha Baran Dutta, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. Beyond the bridge: Contention-based covert and side channel attacks on multi-gpu interconnect. In *2024 International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 35–36. IEEE, 2024.

[72] Yicheng Zhang, Ravan Nazaraliyev, Sankha Baran Dutta, Andres Marquez, Kevin Barker, and Nael Abu-Ghazaleh. Nvbleed: Covert and side-channel attacks on nvidia multi-gpu interconnect. *arXiv preprint arXiv:2503.17847*, 2025.

[73] Zhenkai Zhang, Kunbei Cai, Yanan Guo, Fan Yao, and Xing Gao. {Invalidate+ Compare}: A {Timer-Free}{GPU} cache attack primitive. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 2101–2118, 2024.