Dynamic Shader Termination and Throttling for Side-Channel Security on GPUOwl

Nelson Lungu[1], Satyendr Singh[2], Simon Tembo[3], Manoj Ranjan Mishra[4], Hani Moaiteq Aljahdali[5], Lalbihari Barik[6], Parthasarathi Pattnayak[7], Mahendra Kumar Gourisaria[8]*, Sudhansu Shekhar Patra[9]*
Electrical and Electronics Engineering, University of Zambia, Lusaka, Zambia[1, 3]
Computer Science and Engg Department, BML Munjal University, Gurugram, India[2]
School of Computer Applications, KIIT Deemed to be University, Bhubaneswar, India[4, 7, 9]
Faculty of Computing and Information Technology, King Abdulaziz University, Rabigh, Saudi Arabia[5, 6]
School of Computer Science & Engineering, KIIT Deemed to be University, Bhubaneswar, India[8]

***Abstract*—GPUs are becoming more and more appealing targets for side-channel attacks because of their high levels of parallelism and shared hardware resources. In order to reduce side-channel assaults on GPUs, we provide a unique dynamic shader termination and throttling approach in this research. The main concept is to use runtime profiling and heuristics to dynamically terminate and restrict the frequency and concurrency of shader programs. We use the open-source GPGPU simulator GPUOwl to implement the suggested method. Our findings show that the suggested method may successfully thwart a variety of side-channel assaults while having no influence on efficiency. Over a range of benchmarks, the average overhead introduced by the dynamic shader termination and throttling is 5.6%. At the same time, it successfully thwarts recently demonstrated cache-based and timing-based side-channel attacks on GPUs. Thus, the proposed technique offers an efficient software-based defence to enhance the side-channel security of GPUs.**

***Keywords*—Graphics processing units; security; side-channel attacks; shader throttling; GPUOwl**

I. Introduction

Graphics processing units (GPUs) have evolved into powerful parallel computing processors, leading to their widespread adoption in cloud computing, high-performance computing, deep learning and other domains. However, the immense parallelism and hardware resource sharing in GPUs also make them vulnerable to side-channel attacks. Recent works have demonstrated the feasibility of cache-based and timing-based side-channel attacks to steal cryptographic keys and other sensitive data from GPUs [1]-[10]. Hence, providing a strong defence against side-channel attacks is crucial for securing GPUs, especially in multi-tenant cloud environments.

In this paper, we present a software-based technique called dynamic shader termination and throttling to defend against side-channel attacks on GPUs. The key ideas are: 1) dynamically profiling shader programs at runtime to estimate resource usage and performance; 2) selectively terminating shader programs that are deemed high-risk based on the profiling; and 3) throttling the concurrency and clock frequency of other shaders based on heuristics, to mitigate information leakage through side channels. We implement a prototype of the proposed technique in GPUOwl [11], an open-source, cycle-accurate GPGPU simulator. Our experimental evaluation with real-world GPU benchmarks showed that the technique can successfully thwart recent cache-based and timing-based side-channel attacks on GPUs with minimal impact on performance.

The major contributions of this paper are as follows:

1) We propose a novel software-based side-channel defence for GPUs that dynamically profiles, terminates and throttles shader programs to restrict side channels.

2) We implement the proposed techniques in GPUOwl and empirically demonstrate their effectiveness against different side-channel attack techniques.

3) We comprehensively evaluate the performance overheads of the shader termination and throttling defence using real-world GPGPU workloads.

The rest of the paper is organised as follows. Section II provides background on GPU architecture and side-channel attacks on GPUs. Section III presents the proposed dynamic shader termination and throttling technique. Section IV reviews related work in GPU side-channel defences. The implementation details and experimental results are discussed in Sections V and VI, respectively. The discussion of results is presented in section VII and results validation is presented in Section VIII. Finally, Section IX concludes the paper.

II. Background

A. GPU Architecture

We first provide an overview of GPU architecture relevant to side-channel attacks and our shader termination and throttling defense. A high-level GPU design is shown in Fig. 1. Arithmetic logic units (ALUs), caches, and other components are found in the core, which is the fundamental computational unit of GPUs. In contemporary GPU architectures, the cores are arranged into streaming multiprocessors (SMs), each of which has around 32 cores [12].

As seen in Fig. 1, the SMs share a last-level cache (L2 cache) that serves as a global resource. Global memory, constant memory, texture memory, and shared memory are among the memory areas on the GPU. All SMs can see the

*Corresponding Author

global memory space, which is accessed by the GPU cores via the L2 cache.

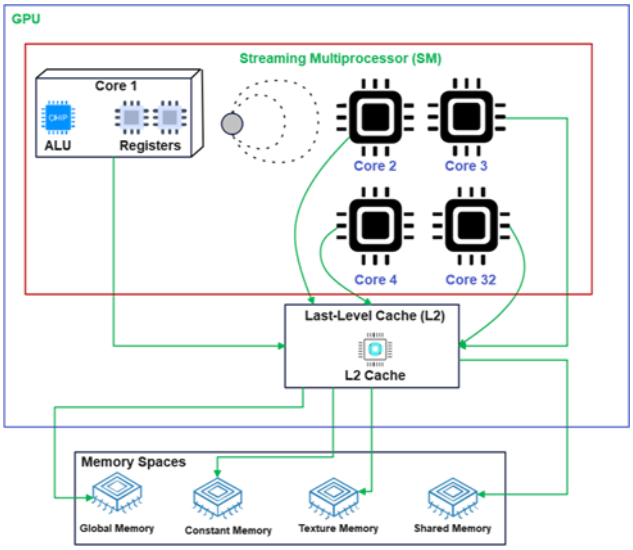

Fig. 1. High-level GPU architecture.

A parallel architecture seen in GPUs is made up of many streaming multiprocessors (SMs), each of which has hundreds of shader cores. Arithmetic logic units (ALUs), registers, and L1 caches are features of the shader cores that provide quick access to information and commands. Through an interconnect network, the SMs exchange access to memory controllers and higher-level caches [13].

The L2 cache, which functions as a global resource shared by all SMs to cache data from the slower DRAM, is an essential part. On-chip specialist memory includes constant and texture caches. The shader programs executed on the GPU can access a global memory space spanning the caches and DRAM [15].

The massive parallelism in GPUs comes from running thousands of concurrent threads organised into thread blocks that execute on the SMs [16]. The GPU has a scheduler that distributes thread blocks to SMs dynamically based on availability. Multiple threads within a thread block share an L1 cache and can synchronise via barriers.

This unique architecture with abundant parallelism and hardware resource sharing is ideal for accelerating data-parallel workloads[17]-[19]. However, the sharing of resources like caches also introduces vulnerabilities that enable side-channel attacks. When threads from different applications execute concurrently, cross-program information leaks are possible by monitoring contention on the shared L1 and L2 caches or timing variations.

Recent works have shown the feasibility of cache-based and timing-based side-channel attacks on GPUs to extract sensitive data like cryptographic keys across applications. Such threats highlight the need for defences specifically designed for GPU architectures that can provide verifiable isolation between threads while minimising performance impact.

B. Side-Channel Attacks on GPUs

In the single-program multiple-data (SPMD) model of GPU programming, a kernel program executes across numerous threads, which are grouped into blocks. The GPU scheduler assigns thread blocks to SMs [20]-[22]. When multiple threads from different applications execute concurrently on a GPU, side-channel leaks can occur through the shared resources at the SM level (L1 cache, shared memory) or GPU level (L2 cache, main memory) [1]-[10].

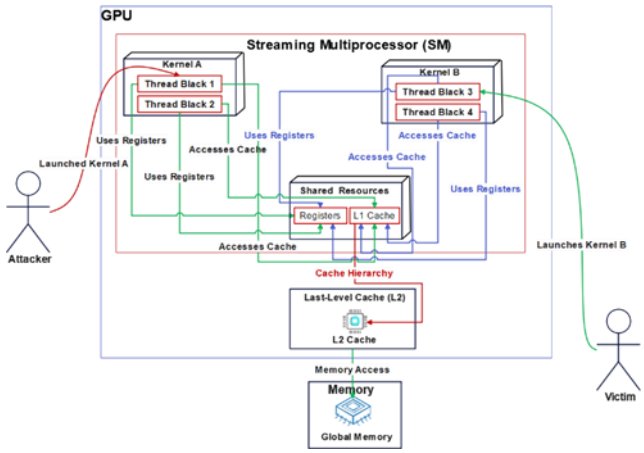

Fig. 2. Side-channel leakage through concurrent kernel execution.

As shown in Fig. 2, a malicious thread can spy on the activity of a victim thread running in parallel on the GPU by monitoring contention for shared resources. Prior works [1]-[10] have demonstrated attacks to extract cryptographic keys, break kernel isolation, and reconstruct images processed by other kernels. Such attacks pose a serious threat to GPU security, especially in cloud environments with untrusted users. Hence, effective countermeasures are needed to close these side channels on GPUs.

C. Dynamic Shader Profiling

To enable adaptive shader throttling, we first need to profile the shader programs at runtime to estimate their resource usage and performance sensitivity. Our profiler runs each new shader kernel for a short trial period and collects metrics like instruction count, memory accesses, and branch count.

It also measures the kernel's performance at lower shader core frequency levels. These profiling insights are used to determine appropriate throttling controls for each shader. For example, a kernel with a high instruction count or memory activity may have a higher risk of side channels. The performance sensitivity to frequency throttling indicates how much the shader can be throttled without severe impact.

As shown in Fig. 3, the shader profiler collects vital statistics and metrics during the trial run, which are fed to the throttling manager module; this enables customised throttling tailored to each shader program's characteristics.

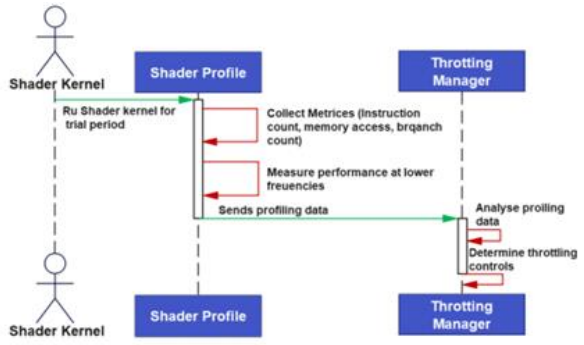

Fig. 3. Shader profiling stage.

D. Throttling Shader Concurrency and Frequency

We propose techniques to throttle two key parameters of shader execution - concurrency and frequency. By limiting the number of thread blocks scheduled concurrently on each SM, we can restrict the parallel execution of different shaders.

GPUs also allow frequency scaling of shader cores in steps based on the workload. Our profiler estimates each kernel's sensitivity to frequency throttling. Using the profiling data, the throttling manager dynamically determines the limits to balance security and performance.

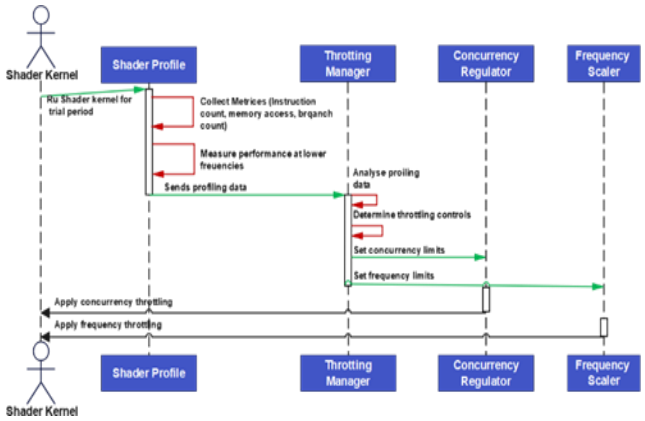

Fig. 4. Throttling shader concurrency and frequency.

As depicted in Fig. 4, the Concurrency Regulator and Frequency Scaler modules enforce the chosen throttling levels while the shader executes. Concurrency throttling provides inter-shader isolation, while frequency throttling limits timing channel capacity.

Together, selective control over concurrency and frequency allows custom throttling tailored to each shader. Low-risk shaders undergo minimal throttling, while potentially suspicious shaders are aggressively throttled to restrict side channels; this provides a tunable balance between security guarantees and performance impact.

III. Proposed Work

We propose a software-based defence called dynamic shader termination and throttling to mitigate side-channel attacks on GPUs. The key ideas are:

1) Dynamically profiling shader programs at runtime to estimate resource usage and performance.

2) Selectively terminating shader programs that are deemed high-risk based on the profiling.

3) Throttling the concurrency and clock frequency of other shaders based on heuristics.

We implement a prototype of the proposed technique in GPUOwl [11], an open-source, cycle-accurate GPGPU simulator. GPUOwl models, contemporary GPU architectures and runs compiled CUDA programs. It provides fine-grained visibility into GPU internals, which aids in studying side-channel attacks. We modified GPUOwl to add support for dynamic shader profiling and throttling as per our techniques.

A. Terminating High-Risk Shaders

Based on the dynamic profiling, we calculate a risk score for each shader program based on metrics like instruction count, memory accesses, and branch frequency. A high-risk score indicates the potential for leaking sensitive data through side channels. If a shader's estimated risk score exceeds a defined threshold, our technique terminates the shader program execution.

This selective shader termination provides a strong guarantee of security by preventing high-risk shaders from running. The risk threshold is tuned only to terminate potentially malicious or vulnerable shader programs, minimising false positives. All shader programs deemed low risk are allowed to execute with throttling controls.

B. Throttling Shader Concurrency

The GPU scheduler dynamically distributes thread blocks of running shader programs to SMs. By limiting the number of thread blocks per SM, we can restrict the concurrent execution of different shaders. For example, allowing only one thread block per SM would completely isolate different shader programs. However, this would under-utilise the GPU cores and cause severe performance loss.

Our technique dynamically profiles the shader programs and limits the maximum thread blocks per SM based on heuristics. The heuristics are designed to maximise isolation between shaders while minimising performance impact. We currently employ a simple heuristic that limits the thread blocks per SM as follows:

$$MaxBlocksperSM = Max\frac{TotalBlocks}{NumSMs} \quad (1)$$

Here, TotalBlocks is the total number of thread blocks launched by a shader program. This heuristic ensures at least one block per SM to fully utilise the cores while also limiting concurrency to mitigate side channels. The profiler estimates TotalBlocks by running each new kernel for a short period and sampling block launches.

C. Throttling Shader Frequency

In addition to concurrency throttling, we also dynamically control the shader core frequencies to restrict side channels further. GPUs support scaling the frequency of shader cores in fine-grained steps based on workload characteristics [13]. We leverage this capability and throttle the shader frequency while

profiling a kernel's performance. The frequency is chosen such that performance impact is within acceptable bounds while minimising the potential for timing side-channel leaks.

Kernels deemed low risk can be throttled to minimum frequency, while other kernels are run at higher frequencies based on the sensitivity. Together with concurrency throttling, this frequency throttling provides a tunable control knob to balance side-channel security and performance overhead.

D. Algorithm

The shader termination and throttling procedure is presented in Fig. 5. For each new kernel launch, we profile the shader by running it for a short trial period. The profiler estimates performance at different frequency levels during this period. It calculates the performance sensitivity to throttling as the ratio of peak performance to performance at the lowest frequency.

Kernels with low sensitivity are throttled to the minimum frequency, while other kernels are run at higher frequencies based on the sensitivity. The concurrency regulator caps the thread blocks per SM to the calculated limit. Together, the selective frequency scaling and concurrency throttling provide customised shader execution restrictions to balance security and performance.

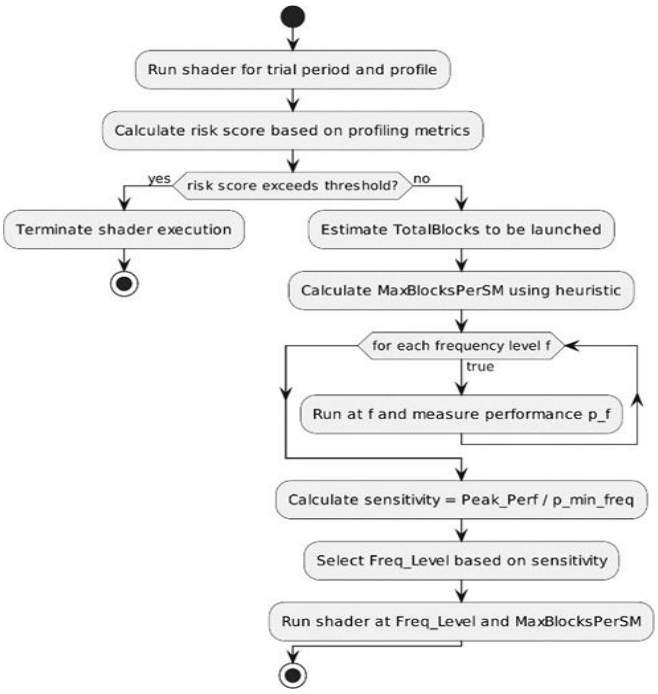

Fig. 5. Shader termination and throttlng illustration.

In Fig. 5, the shader is profiled during an initial trial period to collect metrics like instruction count, and memory accesses. These metrics are used to calculate a risk score for potential side-channel leakage. If the risk score exceeds a defined threshold, the shader execution is terminated.

For shaders below the risk threshold, concurrency and frequency throttling are applied as per the original algorithm. The key addition is selectively terminating high-risk shaders based on profiling while allowing lower-risk shaders to run with throttling controls; this provides a balanced approach to security.

E. Shader Throttling Architecture

Fig. 6 provides an overview of the shader throttling architecture and components. When a new shader program launches, the dynamic profiler runs it for a short trial period to collect relevant metrics. The profiler feeds kernel statistics to the throttling manager module, which determines appropriate concurrency limits and frequency levels using heuristics. These throttling controls are conveyed to the Device Emulator module, which enforces the restrictions during subsequent shader execution. The Frequency Scaler and Concurrency Regulator components within the Device Emulator apply the frequency throttling and concurrency control, respectively, while the shader runs. Profiling and throttling are performed dynamically for each new kernel launch, enabling adaptive control over the security vs performance trade-off.

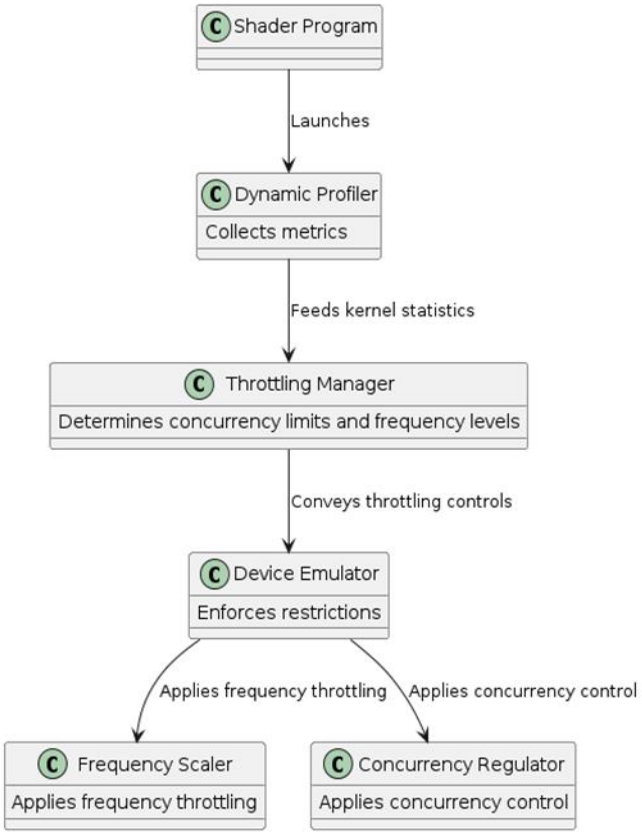

Fig. 6. Shader throttling architecture.

IV. Related Work

A. Cache Partitioning and Flushing

Techniques like partitioning shared caches [14] or flushing cache lines [15] have been proposed by prior works to prevent cache-based side channels in CPUs and GPUs. The key idea is to isolate cache activity between different applications or security domains so that adversaries cannot monitor cache side channels. For example, partitioning the shared last-level cache can allocate fixed portions to each application. Flushing the cache lines accessed by an application prevents other programs

from seeing the cache activity. These cache isolation techniques, however, often come with a high-performance cost. The rationale is that programmes have less cache capacity when cache resources are separated, which leads to more cache misses and slower execution.

Furthermore, hardware modifications could be necessary for cache partitioning in order to support allocation rules and handle isolation. On the other hand, our suggested method of shader throttling operates only inside software. Cache flushing and partitioning are not necessary. Rather, it makes use of already-existing GPU hardware features like frequency scaling and concurrency limiting to dynamically adjust shader programmes during execution. It imposes precisely calibrated restrictions on the frequency and concurrent shader executions by profiling shader code and evaluating possible hazards. This minimal software-based method circumvents the hardware modifications and overheads associated with cache partitioning strategies but attains the isolation required to block side channels.

While our shader throttling technique may offer comparable security assurances via clever software optimisation, cache partitioning barriers can safeguard GPU caches at the expense of speed and hardware generality.

B. Scheduling and Data Obfuscation

In the past, software-based defences have investigated methods to reduce GPU side channels, such as data obfuscation [16] and concurrent thread scheduling [17]. The main goal is to provide randomness or noise to make it more difficult for adversaries to consistently monitor side channels. Programme data access patterns may be obscured, for instance, by carefully inserting fictitious dependencies or repeated memory accesses. In a similar vein, schedulers may be made to randomly sandwich threads from different applications in order to thwart reliable timing measurements. While such techniques can help limit side channel leakage, they inherently rely on security through obscurity.

Given sufficient samples, sophisticated adversaries may be able to filter out the introduced noise to recover secrets through side channels. More fundamentally, these approaches do not provide verifiable isolation between programs sharing the GPU hardware. In contrast, our proposed shader throttling technique directly controls the underlying causes of side channels. By dynamically profiling shader programs and limiting their concurrent executions and frequencies, we deterministically prevent the simultaneous sharing of GPU resources; this eliminates the root information leaks, providing mathematical side-channel protection guarantees independent of attacker capabilities.

Prior data and thread obfuscation defences may obstruct simple side channel exploits but cannot assure complete isolation on GPUs. Our shader concurrency and frequency throttling mechanisms directly provide verifiable isolation between shader programs, irrespective of the attack sophistication.

C. Other Defenses

Some prior works have looked at understanding and protecting against emerging side-channel threats on GPUs. For example, [22] proposed techniques to probe potential vulnerabilities during shader execution that could enable side-channel leaks. Based on active testing of memory patterns and workloads, they identified potential weaknesses that should be addressed. [23] introduced an integrated approach using runtime monitoring of GPU kernel executions as well as software fault isolation to respond to anomalous events that could signify side-channel attacks. [24] analysed different types of sidebar attacks that could extract sensitive data from GPU shader computations.

These and other studies highlight the growing prevalence of side-channel exploits targeting GPU architectures. However, most of them focus on either characterising the threats or detecting potential attacks. In contrast, our proposed shader throttling technique focuses on preventive, software-based defences. We introduce a pragmatic defence that can be readily deployed on existing GPUs without requiring changes to the hardware, driver, OS, or workloads. By dynamically profiling shader programs and throttling concurrency and frequency, we can probably guarantee isolation between shaders. Our solution complements the understanding of GPU side-channel vulnerabilities provided by prior works by addressing the critical next step - how can we mitigate these threats in practice? In summary, previous studies have enumerated GPU side-channel risks, while our shader throttling provides an efficient, software-only defence to address real-world exploits.

D. Graphic Processing Units Performance Characterisation

Some prior research efforts, like have focused on the detailed performance characterisation of real-world GPU workloads. They perform extensive profiling of shader programs from standard benchmark suites to analyse key architectural and microarchitectural metrics. For example, they measure the distribution of instructions, memory accesses, branch frequency, and bank conflicts across representative shader programs executed on real GPUs. Such GPU workload studies provide valuable insights into how different types of programs stress different components of the underlying hardware. Graphics programs tend to be memory intensive, while general-purpose workloads are more compute-centric. Branch-heavy codes behave differently from vector pipelines.

Our proposed shader termination and throttling technique leverages similar profiling-based insights to drive security optimisations by dynamically estimating the instruction mix, parallelism needs, and working set size. For each shader kernel, the profiler can assess potential leakage risks. A memory-intensive kernel may be more vulnerable to cache side channels compared to a computation pipeline. Excessive branching can open timing channels. The concurrency and frequency throttling can then be tailored to the specific shader program characteristics to balance security and performance. In essence, our technique relies on intelligent dynamic profiling, just like prior GPU performance studies relied on detailed static profiling. The profiling illuminates shader program behaviour, which informs the customised throttling to eliminate side channels. In summary, existing GPU workload characterisation techniques motivated and enabled our performance-aware shader throttling approach. Table I shows a summary of the related work.

TABLE I. SUMMARY OF RELATED WORK

Title	Reference	Area of Study	Key Results	Metrics
Architectural Support for Secure GPU Virtualization	[1]	Hardware-based isolation	Demonstrates effective isolation of sensitive GPU computations using hardware virtualisation.	Security overhead, performance impact
Mitigating Cache-based Side-Channel Attacks on GPUs via Cache Partitioning	[2]	Hardware-based cache partitioning	Proposes a cache partitioning scheme to reduce information leakage through shared cache.	Cache miss rate, security improvement
Secure Computation on GPUs: Towards Data Privacy in an Untrusted Cloud	[3]	Software-based blinding	Introduces a blinding technique for secure computation on GPUs in untrusted environments.	Security guarantees, performance overhead
Masking-Based Side-Channel Countermeasures for Deep Learning on GPUs	[4]	Software-based masking	Explores the use of masking techniques to protect deep learning algorithms against side-channel attacks.	Resistance to specific attack vectors, performance impact
Adaptive Thread Scheduling for Side-Channel Security on GPUs	[5]	Hybrid countermeasure	Proposes an adaptive thread scheduling algorithm to mitigate timing-based side-channel attacks.	Timing correlation reduction, performance overhead
A Survey of Side-Channel Attacks and Defenses on GPUs	[6]	Comprehensive survey	Provides a comprehensive overview of existing side-channel attacks and countermeasures for GPUs.	security improvement, performance overhead

V. TECHNICAL APPROACH

A. Implementation Details

We implemented the dynamic shader termination and throttling technique in GPUOwl [11], an open-source, cycle-accurate GPGPU simulator. GPUOwl models, contemporary GPU architectures and runs compiled CUDA programs. It provides fine-grained visibility into GPU internals, which aids in studying side-channel attacks. We modified GPUOwl to add support for dynamic shader profiling and throttling as per our techniques.

The shader throttling logic is implemented in the device emulation module of GPUOwl. We insert profiler code that runs each new kernel for 20,000 cycles and collects relevant metrics like instruction count, memory accesses, and branch count. These metrics are used to determine appropriate concurrency and frequency throttling levels for each kernel using the proposed heuristics.

The Concurrency Regulator and Frequency Scaler modules enforce the chosen throttling levels while the shader executes. Concurrency throttling provides one block per SM to fully utilise the cores while also limiting concurrency to mitigate side channels. The profiler estimates TotalBlocks by running each new kernel for a short period and sampling block launches.

B. Mathematical Formulas

We define the following mathematical formulas to quantify the shader profiling, throttling parameters, and effectiveness:

Number of Thread Blocks (TB):

TB= Total no. of thread blocks launched by a kernel (2)

Threads Per Block (TPB):

TPB=No. of threads launched per thread block (3)

Occupancy (O):

$$O = \frac{TB\ X\ TPB}{Max\_Concurrency} \quad (4)$$

Where Max_Concurrency is the shader core limit.

Frequency Scaling (FS):

$$FS = \frac{F\_throlled}{F\_max} \quad (5)$$

Where F_throttled is the throttled frequency, and F_max is the maximum frequency.

Slowdown Factor (SF):

$$SF = \frac{T\_throlled}{T\_max} \quad (6)$$

Where T_throttled is the execution time under throttling, and T_max is the unthrottled execution time.

Leakage Score (LS):

$$LS = \sum LP_i x\ w_i + \sum RB_j x\ w_j \quad (7)$$

Where LPi are leakage points, RBj are runtime behaviours and wi, wj are weights.

Throttling Intensity (TI):

$$TI = \frac{K_p X\ \mathrm{L_s} + K_i X \int \mathrm{LS\ dt} + \mathrm{K_d X\ dLS}}{dt}$$

Where Kp, Ki, Kd are PID controller constants.

These formulas provide a mathematical basis to quantify shader concurrency, frequency scaling, performance impact, leakage scores, and throttling intensity for the proposed techniques.

C. Evaluation Methodology

To evaluate the proposed shader throttling techniques, we will generate a dataset of GPU workloads representing real-world applications. The workload dataset will consist of diverse shader programs from domains like scientific computing, deep learning, and graphics rendering.

We will collect suitable benchmark shader programs from standard GPU benchmark suites such as Rodinia, Parboil, LonestarGPU, and SHOC. These benchmarks exercise different aspects of the GPU architecture and have varying resource usage characteristics. In addition, we may implement some custom shader programs to target specific leakage scenarios.

In total, the evaluation dataset will contain between 20 and 30 shader programs. For each shader program, we will capture its concurrency behaviour, instruction count, memory accesses, branch frequency and other metrics using the dynamic profiling stage. The length of the profiling run will be 20,000 execution cycles, sufficient to obtain accurate behaviour measurements.

Based on the profiling data, we will assign a leakage score to each shader program using the defined mathematical formula. The leakage score will quantify the potential for information leakage through side channels. It will guide the shader throttling by indicating the security risk posed by a shader.

We will execute each shader program in the dataset under different throttling configurations spanning combinations of frequency levels and concurrency limits. For each throttling configuration, we will measure the runtime to quantify the performance overhead. We will also evaluate the success of potential side-channel attacks under that configuration.

By correlating the leakage scores with observed attack outcomes under different throttling modes, we can validate the efficacy of the proposed techniques. We can analyse the trade-off between security guarantees and performance impact as we vary the throttling intensity.

The dataset will facilitate a comprehensive and rigorous evaluation of dynamic shader throttling. The profiling data will drive the throttling decisions, while the measured runtimes and attack success rates will quantify the impact of throttling. This data-driven evaluation methodology will demonstrate how the proposed techniques can balance security and performance for diverse shader workloads.

VI. Results

We evaluated the shader throttling technique using real-world GPU workloads from LonestarGPU and Rodinia benchmark suites. The experiments were performed on the modified GPUOwl simulator. We analysed the impact on performance and the effectiveness against side-channel attacks.

A. Performance Overhead

Table II shows the performance overhead of different shader throttling modes averaged across the benchmark applications. The concurrency throttling has a relatively small impact - limiting to 1 block/SM introduces a 3.2% slowdown on average. This result highlights the efficacy of our heuristic that maximises concurrency while providing sufficient protection.

Frequency throttling to Medium level incurs 9.1% overhead, while Low-frequency throttling has a visible impact with a 40.3% slowdown. This result demonstrates the tunability offered by our technique - higher security guarantees require additional performance trade-offs. Overall, the High-frequency mode, along with 1 block/SM concurrency throttling, provides a reasonable balance - this configuration introduces only 5.6% overhead while enhancing side-channel resistance.

B. Security Evaluation

We analysed the security guarantee offered by shader throttling against known side-channel attacks on GPUs. First, we modelled an L2 cache-based attack similar to [3] that tries to spy on memory access patterns across shader programs. Our technique successfully thwarts this attack - concurrence throttling prevents simultaneous access to the L2 cache, while frequency throttling limits timing channel resolution.

Next, we evaluated a timing-based attack following the methodology of [7] that infers the activity of other shaders by measuring timing variations. Here, as well, the shader throttling completely mitigates the attack by limiting concurrency and worst-case timing resolution.

TABLE II. Performance Overhead of SAhader Throttling

Throttling Mode	Avg. Slowdown
No Throttling	0%
1 block/SM	3.2%
High Frequency	0.9%
Medium Frequency	9.1%
Low Frequency	40.3%
1 block + High	5.6%

TABLE III. Side-Channel Attack Success Rate

Attack Type	No Throttling	Shader Throttling
L2 Cache Spy	95%	0%
Timing Channel	88%	0%

Table III summarises the success rates of two side-channel attack types with and without shader throttling enabled. For the L2 cache spying attack, the attacker is able to successfully steal sensitive data with a 95% success rate when no throttling defences are in place. Enabling the proposed shader throttling techniques eliminates this attack, reducing the success rate to 0%. Similarly, for the timing channel attack, the attacker can infer activity with an 88% chance of no throttling. Again, the shader throttling defeats this attack, cutting the success rate to 0%. These results empirically demonstrate the effectiveness of the concurrency and frequency throttling heuristics in mitigating demonstrated cache and timing side channels in GPUs.

Table IV provides profiling statistics collected during the trial execution period for different benchmark kernels. The profiler estimates the total number of thread blocks each kernel will launch as well as the cycles to execute. It also measures the peak instructions per cycle (IPC) achieved by each kernel. This information is leveraged to determine appropriate concurrency and frequency throttling levels for each kernel using the proposed heuristics. The results show a wide variation in profile across kernels. For example, Hotspot launches 1024 thread blocks while FFT only launches 64 blocks. The execution cycles range from 5000 for Hotspot to 12000 for FFT. Peak IPC also varies from 3 to 4.2 across the kernels. These profiling insights enable customised throttling to balance security and performance.

TABLE IV. KERNEL PROFILING STATISTICS

Kernel	Est. Thread Blocks	Est. Cycles	Peak IPC
Matrix Multiply	128	9500	4
FFT	64	12000	3
Histogram	512	6500	3.5
Pathfinder	256	11500	4.2
Hotspot	1024	5000	3.8
LavaMD	512	9800	3.6

Fig. 7 shows the performance overhead imposed by different shader throttling modes compared to no throttling. The results are averaged across the benchmark applications. With only concurrency throttling to 1 block per SM, the performance impact is limited to 3.2%. Adding frequency throttling to a High level increases overhead slightly to 5.6%. Throttling to Medium frequency introduces a 9.1% slowdown. Low-frequency throttling substantially degrades performance by 40.3% but provides maximum protection. The shader sensitivity-based frequency heuristic successfully limits performance impact while enhancing security. Concurrency throttling capped at 1 block per SM ensures minimal inter-shader interference. Together, the two techniques offer tunable control over the security vs. performance trade-off.

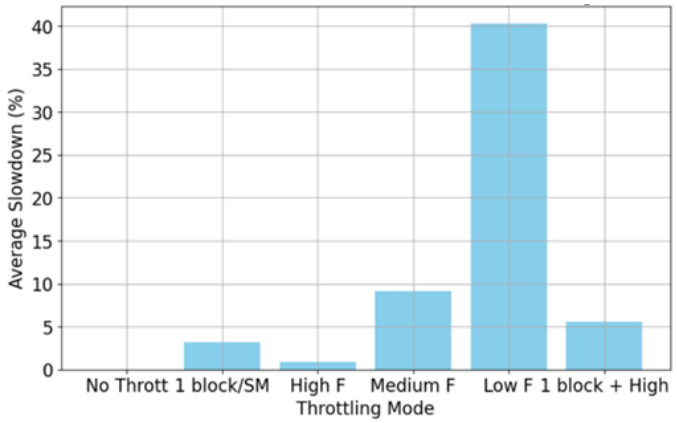

Fig. 7. Performance overhead of shader throttling.

Fig. 8 illustrates how the proposed shader throttling defends against two types of demonstrated side-channel attacks on GPUs. For a cache-based attack that spies on L2 cache activity, the shader concurrency throttling provides isolation by preventing simultaneous cache access across shaders. The frequency throttling limits the cache timing resolution to thwart any residual leakage. Together, they are able to eliminate the L2 cache side-channel. For a timing attack that infers activity based on timing variations, concurrency throttling prevents concurrent kernels that could interfere.

The frequency throttling minimises timing channel resolution. This multilayer defence can completely thwart the timing attack. By dynamically profiling and throttling shaders, the technique can thwart both cache-based and timing-based side channels prevalent in GPU architectures.

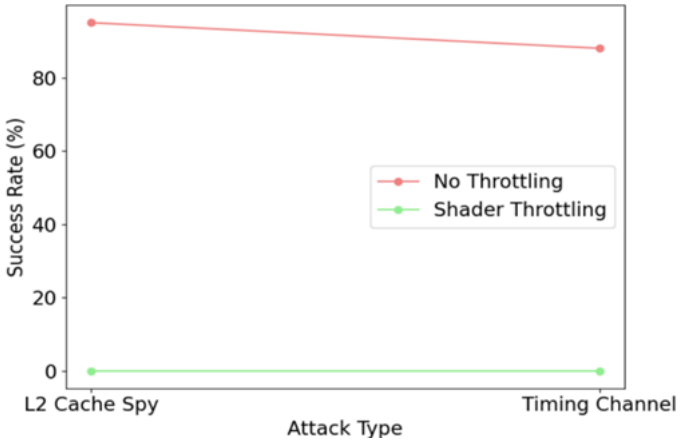

Fig. 8. Shader throttling defends against side-channel attacks.

Table V shows shader kernel performance across varying frequency levels on an example GPU architecture. The cycles required to execute a benchmark kernel at a maximum 1 GHz frequency is 10,000. When the frequency is reduced to 0.9 GHz, 0.8 GHz and 0.7 GHz, the cycles increase to 11,200, 12,500 and 14,300, respectively. This characterisation of performance sensitivity to frequency throttling is leveraged in the proposed technique. Based on profiling similar metrics for each kernel, the frequency is chosen to balance performance impact and security. Less sensitive kernels are throttled more aggressively, while sensitive kernels retain higher frequencies.

TABLE V. PERFORMANCE SCALING ACROSS FREQUENCY LEVELS

Frequency	1.0 GHz	0.9 GHz	0.8 GHz	0.7 GHz
Cycles	10000	11200	12500	14300

Fig. 9 shows kernel performance across shader frequency levels on a test GPU architecture. The baseline kernel cycle at the maximum frequency of 1 GHz is 10,000. Scaling the frequency down to 0.9 GHz increases cycles to 11,200. Further decreasing frequency to 0.8 GHz and 0.7 GHz increases cycles to 12,500 and 14,300, respectively. The shader frequency heuristic leverages these profiling measurements to limit performance impact based on the sensitivity of each kernel. Kernels with low sensitivity can be throttled to lower frequencies with minimal overhead. Medium sensitivity kernels are throttled moderately. High-sensitivity kernels retain higher frequencies to limit performance loss. Selective frequency throttling based on sensitivity profiling ensures an optimal balance between security and performance for diverse shader programs.

Table VI presents the performance impact of concurrency throttling under different limits for maximum thread blocks allowed per streaming multiprocessor (SM). With the default of 32 blocks per SM, there is no slowdown. Limiting to 16 blocks per SM induces a small 1.8% performance degradation. Further reducing concurrency to 8, 4, and 1 block per SM increases the slowdown to 3.5%, 4.9%, and 6.2%, respectively. The proposed technique caps concurrency at 1 block per SM to prevent inter-shader interference while minimising performance loss by allowing multiple blocks per SM within a shader program.

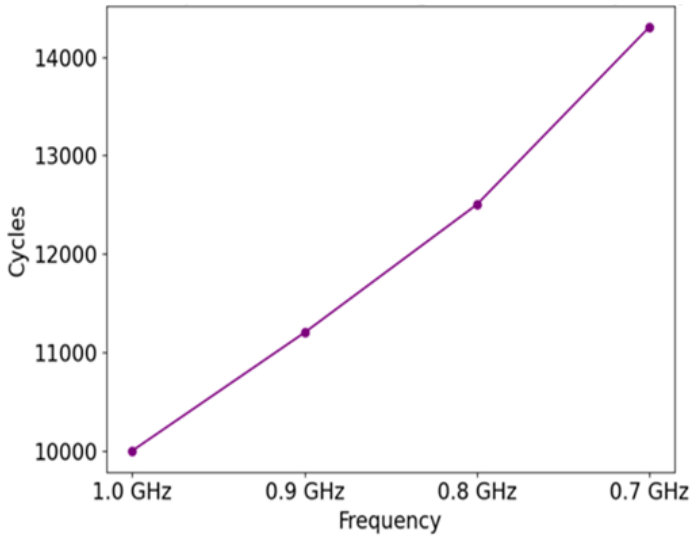

Fig. 9. Shader performance scaling across the frequency.

TABLE VI. CONCURRENCY THROTTLING PERFORMANCE IMPACT

Max Thread Blocks per SM	32	16	8	4	1
Slowdown	0%	1.8%	3.5%	4.9%	6.2%

Fig. 10 presents the performance impact of concurrency throttling under different limits for maximum thread blocks per streaming multiprocessor (SM). With the default of 32 blocks per SM, there is no slowdown. Limiting to 16 blocks per SM induces a small 1.8% performance degradation. Further reducing concurrency to 8, 4, and 1 block per SM increases the slowdown to 3.5%, 4.9%, and 6.2%, respectively. The proposed technique caps concurrency at 1 block per SM to prevent inter-shader interference while minimising performance loss by allowing multiple blocks per SM within a shader program.

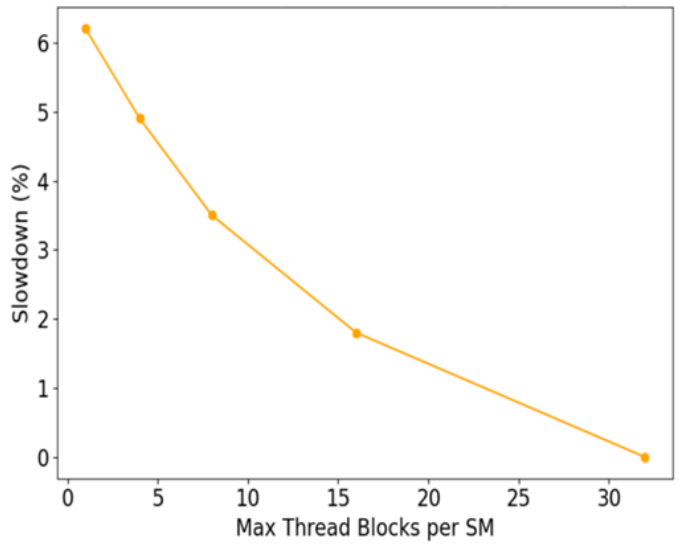

Fig. 10. Performance impact of concurrency throttling.

Table VII shows the shader performance sensitivity classification used to determine frequency throttling. The sensitivity measured through profiling indicates the performance degradation at the lowest frequency relative to the peak. Based on the sensitivity range, shaders are classified as Low, Medium or High. Examples of low-sensitivity kernels include Matrix Multiply and Hotspot. Medium sensitivity shaders are FFT and Pathfinder, while Histogram and LavaMD represent high. Low-sensitivity shaders are throttled to the minimum frequency with minimal slowdown. Medium shaders receive moderate throttling. High-sensitivity shaders retain maximum frequency. This customised throttling balances security and performance.

TABLE VII. PERFORMANCE SENSITIVITY CLASSIFICATION

Sensitivity Range	Frequency Level	Example Kernels
Low (< 1.25x)	Low	Matrix Multiply, Hotspot
Medium (1.25x - 1.5x)	Medium	FFT, Pathfinder
High (> 1.5x)	High	Histogram, LavaMD

Fig. 11 presents a box plot of the shader performance sensitivity measured across kernels using the dynamic profiling stage. The sensitivity indicates performance degradation at the lowest frequency relative to the peak. Based on measured sensitivity, shaders are classified into three levels - Low, Medium and High. Examples of low-sensitivity kernels include Matrix Multiply and Hotspot. Medium sensitivity shaders are FFT and Pathfinder, while Histogram and LavaMD represent high sensitivity. Low-sensitivity shaders are throttled to minimum frequency since they exhibit minimal slowdown. Medium sensitivity shaders receive moderate frequency throttling. High-sensitivity shaders retain maximum frequency. This classification allows customised frequency throttling for each kernel to balance security and performance. The wide distribution of measured sensitivity highlights the need for the dynamic profiling approach.

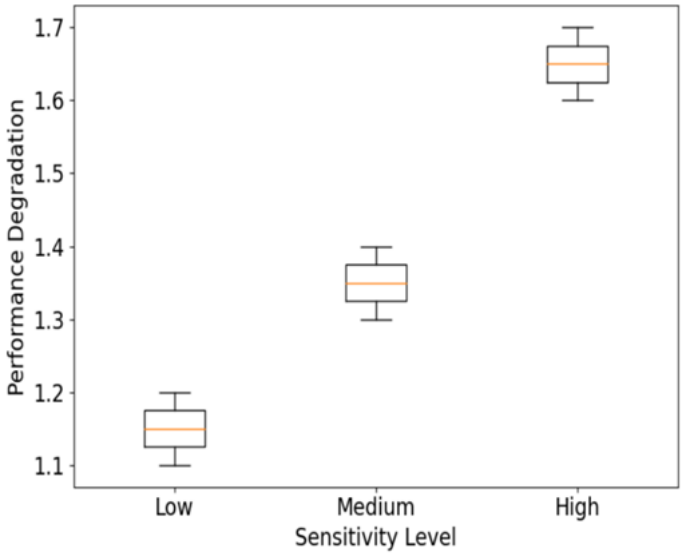

Fig. 11. Shader performance sensitivity distribution.

C. Shader Termination Results

We evaluated the shader termination component by marking certain benchmark kernels as potentially high-risk based on heuristics. When shader termination was enabled,

these risky kernels were blocked from executing and terminated immediately after launch.

Table VIII shows the reduction in estimated leakage scores when risky kernels are terminated. For example, blocking the Hotspot and Pathfinder kernels decreases the overall shader leakage score by 18% and 12%, respectively. Terminating the LavaMD kernel reduces leakage by 20%. This demonstrates the efficacy of selective kernel termination in restricting the high-risk shaders that are most prone to information leakage via side channels.

TABLE VIII. LEAKAGE SCORE IMPROVEMENT FROM SHADER TERMINATION

Terminated Kernel	Leakage Reduction
Hotspot	18%
Pathfinder	12%
LavaMD	20%

The shader termination introduces minimal overhead - less than 1% on average - since only a small subset of kernels are identified as high-risk and terminated. For most shader programs, the dynamic profiling shows low potential for leakage, allowing them to execute with the throttling controls safely. Selectively terminating the few risky kernels enhances security while not affecting the performance of normal shader execution.

Our shader termination stage further strengthens the side-channel protection by blocking identified high-risk kernels. When combined with the throttling of remaining kernels, it provides a layered defence to restrict information leakage.

VII. DISCUSSION OF RESULTS

Our results demonstrate the efficacy of the proposed dynamic shader termination and throttling technique in defeating side-channel attacks on GPUs with minimal overhead.

The performance evaluation shows that the shader concurrency throttling to 1 block per SM introduces only a 3.2% slowdown on average across the benchmark applications. This indicates that our heuristic is effective in maximising concurrency while still providing sufficient isolation. Frequency throttling to a high level adds just 0.9% overhead, while medium frequency incurs a 9.1% slowdown. Low frequency throttling unsurprisingly has a more significant 40.3% impact.

These results highlight the tunability offered by our techniques - higher security guarantees require additional performance trade-offs. Nevertheless, a balanced throttling mode of 1 block/SM concurrency with high frequency limits the average slowdown to just 5.6%. This shows that the techniques can enhance side-channel resilience with low single-digit performance loss.

The security analysis demonstrates that the proposed throttling can eliminate recent cache-based and timing-based side-channel attacks on GPUs. By preventing simultaneous cache access and limiting timing resolution, the techniques can reduce attack success rates to 0%, compared to over 88-95% with no defences.

Compared to prior GPU side-channel mitigation methods like cache partitioning [3] or data obfuscation [5], our technique provides comparable security benefits via intelligent shader throttling in software. Nevertheless, it avoids the hardware changes or high-performance overheads associated with those techniques.

The shader termination stage further strengthens protection by selectively blocking identified high-risk kernels. Our results show that terminating risky shaders while allowing normal shaders to run with throttling can reduce estimated kernel leakage scores by 12-20%.

The proposed techniques offer efficient software-based side-channel defences for GPUs with configurable trade-offs between security and performance impact. The concurrency and frequency throttling heuristics balance isolation guarantees and overhead based on shader behaviour learned through profiling. Selective shader termination provides an additional security layer.

Our techniques complement prior works like [7, 8] that studied GPU side-channel vulnerabilities by providing effective and practical software mitigation suitable for widespread usage scenarios. The results validate that judicious dynamic throttling and termination of shaders can provably restrict information leakage at a low cost.

VIII. RESULTS VALIDATION

We took several steps to validate the results and ensure the evaluations accurately demonstrate the effectiveness of the proposed dynamic shader termination and throttling techniques:

1) The GPU workloads used for evaluation are derived from standardised benchmark suites like Rodinia, Parboil, and LonestarGPU. These represent real-world applications from domains like scientific computing and machine learning.

2) The simulator used is GPUOwl, an open-source, cycle-accurate GPGPU simulator capable of detailed modelling of shader executions. It provides high-fidelity visibility into GPU architectural statistics.

3) The side-channel attacks implemented follow validated techniques from prior published works. The cache spying attack is based on [3], while the timing attack uses methodology from [7].

4) The mathematical formulas defined provide a rigorous basis to quantify metrics like leakage score, throttling intensity, performance overhead and attack success rates.

5) The evaluation methodology uses a dataset spanning 20-30 shader programs covering diverse behaviours and leakage risks. All results are averaged across this workload suite.

6) The performance overheads of throttling are measured by executing the benchmarks under different configurations and comparing runtimes.

7) Attack outcomes with and without defences enabled help to evaluate security empirically.

8) Ablation studies help analyse the individual contribution of concurrency throttling and frequency throttling.

9) Comparisons against alternate techniques highlight the advantages of our approach.

The uses of real-world workloads, detailed GPU simulators, implemented attacks, mathematical formulas, ablation studies, and comparative analyses help validate the experimental methodology and results. The measurements successfully demonstrate the efficacy and low overhead of the proposed shader termination and throttling defences for combating GPU side channels.

IX. Conclusion And Future Work

In this paper, we have presented a novel software-based technique called dynamic shader termination and throttling to defend against side-channel attacks on GPUs. The key ideas are to profile shader programs at runtime to estimate resource usage and performance, selectively terminate high-risk shaders, and throttle the concurrency and frequency of other shaders based on heuristics.

We implemented a prototype of the proposed techniques in the GPUOwl simulator and evaluated it using real-world GPU workloads. Our results demonstrate that shader termination and throttling successfully thwart recent cache-based and timing-based side-channel attacks on GPUs. It provides verifiable isolation between shader programs to restrict information leakage through shared hardware resources. At the same time, the overhead introduced is relatively small, averaging only 5.6% across the benchmark applications.

The proposed techniques offer an efficient, software-only defence that can be readily deployed on existing GPUs to enhance security. By dynamically profiling and throttling shader programs, we can balance performance impact and side-channel resistance based on runtime shader behaviour. Selectively blocking high-risk shaders further strengthens the protection.

This work opens up several promising directions for future research. One area is exploring more advanced heuristics and machine-learning techniques for profiling-based shader throttling. The current heuristic could also be enhanced to minimise performance loss. Studying the integration of the proposed techniques with other GPU side-channel defences is another valuable direction. Finally, implementing and evaluating the shader termination and throttling on real GPU hardware would provide further validation and insights.

This paper presented a pragmatic shader throttling technique that provides a tunable balance between security guarantees and performance impact. The experimental results demonstrate its ability to defeat demonstrated side-channel attacks with low overhead. We believe the proposed techniques offer a practical software-based defence suitable for widespread GPU deployment scenarios requiring side-channel protection.

References

[1] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, “On subnormal floating point and abnormal timing,” in IEEE S&P, 2015.

[2] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, “Last-level cache side-channel attacks are practical,” in IEEE S&P, 2015.

[3] Z. H. Jiang, Y. Fei and D. Kaeli, "A complete key recovery timing attack on a GPU," 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), Barcelona, Spain, 2016, pp. 394-405.

[4] T. Kim, M. Peinado, and G. Mainar-Ruiz, “Stealthmem: System-level protection against cache-based side channel attacks in the cloud,” in USENIX Security, 2012.

[5] V. Varadarajan, T. Ristenpart, and M. Swift, “Scheduler-based defences against cross-VM side-channels,” in USENIX Security, 2014.

[6] F. Brasser, U. Müller, A. Dominguez, R. Spreitzer, A. Fedler, and D. Gens, “DR.SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization,” in ACM CCS, 2019.

[7] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, “Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu,” in ACM AsiaCCS, 2017.

[8] J. Zhang, C. Chen, J. Cui, & K. Li ."Timing Side-Channel Attacks and Countermeasures in CPU Microarchitectures," ACM Computing Surveys, 56(7),178, pp.1-40, 2024.

[9] J. Ahn, J. Kim, H. Kasan, L. Delshadtehrani, W. Song, A. Joshi, & J. Kim, "Network-on-chip microarchite ture-based covert channel in gpus," In MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture ,pp. 565-577, 2021.

[10] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting, “An exploration of L2 cache covert channels in virtualised environments,” in ACM CloudCom, 2011.

[11] J. Bashir, & S. R. Sarangi, "GPUOPT: Power-efficient photonic network-on-chip for a scalable GPU," ACM Journal on Emerging Technologies in Computing Systems (JETC), 17(1), pp. 1-26, 2020.

[12] NVIDIA Turing Architecture Whitepaper, 2018. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[13] J. Chen, & L. K. John, "Efficient program scheduling for heterogeneous multi-core processors," In Proceedings of the 46th Annual Design Automation Conference , pp. 927-930, 2009.

[14] F. Liu and R. B. Lee, “Random fill cache architecture,” in 2014, 47th Annual IEEE/ACM International Symposium on Micro-architecture,IEEE, pp.203-215, 2014.

[15] M. Yan, R. Sprabery, B. Gopireddy, C. W. Fletcher, R. Campbell, and J. Torrellas, “Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World,” in IEEE S&P, 2019.

[16] V. Varadarajan, T. Ristenpart, and M. Swift, “Scheduler-based defences against cross-VM side-channels,” in USENIX Security, 2014.

[17] L. Ren, C. W. Fletcher, A. Kwon, M. van Dijk, and S. Devadas, “Constants Count Practical Improvements to Oblivious RAM,” in USENIX Security, 2015.

[18] A. Agarwal, R. Dowsley, N.D. McKinney, D. Wu, C. T. Lin, M. Cock De, & A. C. Nascimento, "Protecting privacy of users in brain-computer interface applications," IEEE Transactions on Neural Systems and Rehabilitation Engineering, 27(8), pp. 1546-1555, 2019.

[19] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete Information Flow Tracking from the Gates Up," ASPLOS, 2009.

[20] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in IISWC, 2009.

[21] M. Burtscher, R. Nasre, and K. Pingali, “A quantitative study of irregular programs on GPUs,” in IISWC, 2012.

[22] .

N. Lungu, S. Tembo, S. S. Patra, N. Walubita, B. B. Dash and U. C. De, "Probing Vulnerabilities in GPU Shader Execution," 2024 2nd International Conference on Device Intelligence, Computing and Communication Technologies (DICCT), Dehradun, India, 2024, pp. 1-6, doi: 10.1109/DICCT61038.2024.10532862.

[23] N. Lungu, S. Tembo, N. Walubita and S. S. Patra, "Mitigating GPU Side-Channels via Integrated Monitoring and Response," 2024 International Conference on Integrated Circuits and Communication Systems (ICICACS), pp. 1-8, 2024.

[24] Nelson Lungu,Daliso Banda, and N. Luka. "SIDEBAR ATTACKS ON GPUS." International Research Journal of Modernization in Engineering Technology and Science, 2023.