UNIVERSITY OF CALIFORNIA
RIVERSIDE

Security of Graphics Processing Units (GPUs) in Heterogeneous Systems

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Hoda Naghibijouybari

September 2020

Dissertation Committee:

Professor Nael Abu-Ghazaleh, Chairperson
Professor Srikanth Krishnamurthy
Professor Zhiyun Qian
Professor Daniel Wong

The Dissertation of Hoda Naghibijouybari is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

There is a large number of people that I would like to thank individually for making my Ph.D. graduation happens.

First of all, I would like to express my deepest appreciation to my advisor, Prof. Nael Abu-Ghazaleh for his consistent support, invaluable guidance and encouragement in my professional and personal life, during the PhD. His motivation, immense knowledge, and great insight into research inspired me to overcome all difficulties in the past five years. I will forever be thankful to him for his helpful career advice. I really appreciate the effort and time you invested in helping me to achieve my full potential in life, Prof. Nael. It is my great pleasure and honor to have known you and worked with you.

I am also thankful to my dissertation committee members; professors Srikanth Krishnamurthy, Zhiyun Qian and Daniel Wong for providing me invaluable feedback on my research. In addition to professor Rajiv Gupta whom was always generous in giving guidance regarding my professional development. I also want to thank my collaborators, Khaled Khasawneh, Sankha Dutta, Ajaya Neupane, Qiumin Xu, Shibo Wang and Prof. Murali Annavaram.

The past five years at Riverside have been some of the most wonderful time in my life, working with the talented and friendly colleagues in our research lab and spending time with my supportive friends. I would like to thank my colleagues: Khaled Khasawneh, Fatemah Alharbi, Sankha Dutta, Shafiur Rahman, Hodjat Asghari-Esfeden, Esmaeil Mohammadian Koruyeh, Ahmed Abdo, Sakib Md Bin Malek, Jason Zellmer, Abdulrahman Bin Rabiah, and specially Shirin Haji Amin Shirazi.

I would like to specially thank my dear partner, Sajjad Bahrami for his endless love and support, and my friends, Valeh Ebrahimi, Tina Mirzaie, Marzie Jafari, Anna Pohl, Sepideh Azarnoosh, Shaghayegh Gharghabi, Arezoo Etesamirad and Alireza Ramezani. Thank you for being there whenever I have needed you. Without you, I would not experience this fascinating PhD life.

Finally, and also most importantly, the completion of my dissertation would not have been possible without the support and nurturing of my parents, Hossein Naghibijouybari and Sherafat Fadaie. I feel so honored and blessed to have you as my parents. Thank you for instilling me with a strong passion for learning and for doing everything possible to put me on the path to greatness. I am also extremely grateful to my sisters, Hadis (and her family), Maryam and Fatemeh for their significant support and encouragement, throughout my life. No matter how far you are and how long I was not able to visit you all, I feel you next to me every single day.

Dedicated to my dear parents, for giving me love and strength to chase my dreams.

ABSTRACT OF THE DISSERTATION

Security of Graphics Processing Units (GPUs) in Heterogeneous Systems

by

Hoda Naghibijouybari

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2020
Professor Nael Abu-Ghazaleh, Chairperson

Modern computing platforms are becoming increasingly heterogeneous, combining a main processor with accelerators/co-processors to perform data-intensive computations. As the most common accelerator, Graphics Processing Units (GPUs) are widely integrated in all computing devices to enhance the performance of both graphics and computational workloads. GPUs as new components in heterogeneous systems introduce potential vulnerabilities and other security problems.

This dissertation studies the security of modern GPUs in terms of micro-architectural covert and side channels attacks and defenses. In micro-architectural attacks, information leakage is measured through processes interactions through the shared hardware resources on a processor.

The first contribution of my dissertation is a study of covert channel attacks on General Purpose GPUs (GPGPUs). I first reverse engineer the hardware scheduler to create co-residency between two malicious applications. I study contention and construct operational channels on different resources including caches, functional units and memory on three different Nvidia GPGPUs, obtaining error-free bandwidth of over 4 Mbps.

Next, I explore side channel attacks; a dangerous threat vector on GPUs where a malicious spy application can interleave execution with a victim application to extract sensitive information. I build three practical end-to-end attacks in both the graphics and the computational stacks on GPUs: 1) Website fingerprinting attack that identifies user browsing websites, 2) tracking user activity on web browsers that captures keystroke timing, and 3) Neural Network model extraction to reconstruct the internal structure of a neural network with high accuracy.

The third contribution of the dissertation is to study architectural mitigations to protect GPU-based systems against these attacks. I propose GPUGuard, a decision tree based detection and a hierarchical defense framework which isolates contending applications into separate security domains at different hierarchy levels to maximize sharing when it is safe, but to reliably close contention based channels when there is a possibility of such a channel.

The final contribution of my dissertation is an exploration of cross-component covert and side channel attacks in integrated CPU-GPU environments, exploiting the sharing of common resources among them. These attacks demonstrate the vital need to secure heterogeneous systems and components, and not just the CPUs, against microarchitectural attacks.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over last several years, covert and side channel attacks have posed a substantial threat to modern computing systems. They have particularly gained attention due to their potential to reveal sensitive data to untrusted parties as recently demonstrated with the Meltdown [140] and Spectre [124] attacks and their many subsequent variants [128].

Side channels are information leakage channels where an adversary can extract victim's secret data through monitoring the computing activity via some effects such as timing, power or electromagnetic analysis. In covert channels, in contrast, a malicious insider, or trojan, intentionally colludes with the adversary to exfiltrate secrets.

Unlike electromagnetic or power-based channels, microarchitectural side and covert channel do not require physical access to the target device. Instead, only malicious or cooperating spy applications need to be co-located on the same machine as the victim and the leakage is measured through processes interactions through the shared resources on the processor, including caches and other structures.

Most microarchitectural covert and side channel attacks have focused on the main processor and its structures, with substantial research considering variants of attacks and defenses for primary structures such as the caches, branch prediction units, and others. However, modern computing systems are increasingly made up of a federation of the CPU with other specialized accelerators and processors to gain performance, and it is essential for hardware security researchers to understand how covert and side channel attacks manifest within such complex environments and how to secure the systems against potential attacks.

Graphics Processing Units (GPUs) are integrated in modern computing platforms in every domain of computing, from high-end servers and high-performance computing machines all the way down to low-power hand-held devices including mobile phones and tablets to accelerate a wide range of applications including security, computer vision, computer graphics, computational finance, bio-informatics and many others [152].

It is not surprising that to date most of the covert and side channel attacks have been demonstrated on CPUs. GPUs have mostly been spared from such attacks primarily because until recently they had limited (or even no) support for concurrent kernel execution which leads to resource contention. But that limitation is rapidly being relaxed. For example, AMD multiuser GPUs [16] and Nvidia vGPUs [15] both enable up to 16 concurrent clients to share a GPU. Nvidia GPUs support asynchronous compute to concurrently run graphics and general purpose workloads since Maxwell [12]. The Volta multi-process service [20] features spatially sharing a GPU among multiple applications. Intra-SM concurrent kernel execution [176, 230, 219, 220, 70] further enables finer-grain sharing within a single SM to improve overall GPU utilization. Furthermore, GPUs are now being deployed in a virtualized cloud environment such as the Google Cloud [89], Amazon AWS cloud [51] and Microsoft Azure [153]. A malicious VM can now spy on other applications that share a GPU (a side channel attack) or collude with another to covertly communicate sensitive information (a covert channel) to bypass information isolation boundaries.

GPUs often process sensitive data both with respect to graphics workloads (which render the screen and can expose user information and activity) and computational workloads (which may include applications with sensitive data or algorithms). Covert and side channel attacks on GPUs represent a novel and dangerous threat vector.

In this dissertation, I focus on microarchitectural covert and side channel attacks and defenses between two applications co-executing on a GPU. At the core of these attacks an adversary/spy exploits contention in hardware resources to indirectly infer information. GPUs have a substantially different execution model with massive parallelism, internal hardware schedulers that impact colocation and contention, as well as several unique (micro)architectural structures such as constant cache, which provide a varied range of paths for contention based channel formation. I also explore security of GPUs in integrated heterogeneous systems, where attacks can spill over from the GPU to compromise the security of whole system.

1.1 Contributions of the Dissertation

This dissertation studies the security of discrete and integrated GPUs and consists of four main parts (as shown in Figure 1.1): Covert channel and side channel attacks on discrete GPUs and architectural mitigation against these attacks, as well as microarchitectural attacks in integrated CPU-GPU systems.

1.1.1 GPU Covert Channel Attacks

For the first time, we demonstrate that with multiprogramming available on GPGPUs, covert channel attacks between two applications running concurrently on a GPU become possible [160, 161]. The attack offers a number of advantages that may make them an attractive target for attackers compared to CPU covert channels including: (1) With GPU-accelerated

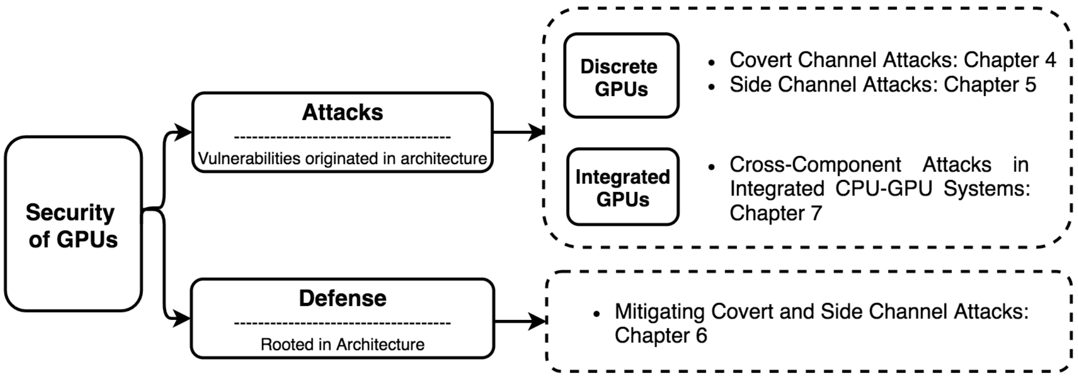

Security of GPUs
Attacks
Vulnerabilities originated in architecture
Defense
Rooted in Architecture
Discrete GPUs
Covert Channel Attacks: Chapter 4
Side Channel Attacks: Chapter 5
Integrated GPUs
Cross-Component Attacks in Integrated CPU-GPU Systems: Chapter 7
Mitigating Covert and Side Channel Attacks: Chapter 6

Figure 1.1: Dissertation Overview

computing available on major cloud platforms such as Google Cloud Platform, IBM cloud, and Amazon web service [166], this threat is substantial. The model of sharing GPUs on the cloud is evolving but allowing sharing of remote GPUs is possibility [75, 186, 58, 180]. Therefore, GPU covert channels may provide the attackers with additional opportunities to co-locate, which is a pre-requisite for these types of attacks [187]; (2) GPGPUs operate as an accelerator with separate resources that do not benefit from protections offered by an Operating system. In fact, due to this property they have been proposed for use as a secure processor [205]; and (3) GPGPU channels can be of high quality (low noise) and bandwidth due to the inherent parallelism and, as we demonstrate, the ability to control noise.

Through reverse engineering, we explore how the communicating applications can leverage two-level scheduler to achieve co-residency. Based on their co-residency options (on the same Streaming Multiprocessor (SM) or across SMs), the SM local resources or inter-SM shared resources can be used for contention. We characterize the contention behavior to exploit it to construct covert channels and demonstrate attacks on constant caches, functional units and global memory on three generations of Nvidia GPUs. We explore different optimizations to increase the bandwidth using synchronization and available parallelism on the GPGPU. We also demonstrate preventing interference from other workloads on covert communication. Our experiments on constructing covert channels through different shared hardware resources show that high bandwidth (over 4 Mbps) and error free covert communication is feasible on GPGPUs, which is the fastest known microarchitectural covert channel under realistic conditions.

1.1.2 GPU Side Channel Attacks

The presence of a covert channel can also forecast the possibility of a side-channel attack. This dissertation demonstrates that the inevitable trend towards sharing on GPUs, indeed leads to side channel exposure in both user devices (through graphics workloads) and computational/cloud computing environments. We introduce a completely new attack vector for side channels that can expose secret information [162, 159].

We first reverse engineer how applications share of the GPU for different threat scenarios and also identify different ways to measure leakage. We demonstrate a series of end-to-end GPU side channel attacks covering the different threat scenarios on both graphics and computational stacks, as well as across them. The first attack implements website fingerprinting through GPU memory utilization API or GPU performance counters. We extend this attack to track user activities as they interact with a website or type characters on a keyboard. We can accurately track re-rendering events on GPU and measure the timing of keystrokes as they type characters in a textbox (e.g., a password box), making it possible to carry out keystroke timing analysis to infer the characters being typed by the user. A second attack uses a CUDA spy to infer the internal structure of a neural network application, demonstrating that these attacks are also dangerous on the cloud. We believe that this class of attacks represents a substantial new threat targeting sensitive GPU-accelerated computational (e.g. deep neural networks) and graphics (e.g. web browsers) workloads.

We propose a mitigation based on limiting the rate of access to the APIs that leak the side channel information. Alternatively (or in combination), we can reduce the precision of this information. We show that such defenses substantially reduce the effectiveness of the attack, to the point where the attacks are no longer effective.

Different attack vectors have been proposed for website fingerprinting and keystroke timing at the level of CPU hardware, OS or web browser and there are several proposed mitigations that completely close these leakage vectors. The graphics end-to-end attacks implemented in this dissertation demonstrate that as GPUs are being involved in the rendering process of every computing device as hardware accelerator to improve the performance and efficiency of rendering and freeing up the CPU for other tasks, it opens up a new avenue for the attacker to bypass all already existed mitigations and spy on users activities in critical applications like web browsers. Using GPUs for web browsers rendering process are being widely supported and improved by introducing WebGL that enables websites and browsers to use GPUs for whole rendering pipeline, making our attacks substantial threat for user privacy. Based on WebGL statistics [40] 98% of visitors to a series of websites used WebGL enabled browsers that can be compromised by GPU side channels presented in this dissertation.

In the future, as more applications supports GPU accelerated rendering, our graphics based side channels such as keystroke timing attack on GPUs can spy on a wide range of applications including text editors and expose users' private data.

1.1.3 Mitigation: Secure GPUs

With the assessment of threats on GPUs, our next goal is to provide a comprehensive solution to mitigate contention based covert and side channel attacks between two kernels co-executing on a GPU. While solutions for such attacks in CPUs have been proposed, GPUs have a substantially different execution model with massive parallelism and our solution uses GPU-appropriate forms of partitioning triggered by detection of covert communication or side channel.

We propose GPUGuard, a dynamic detection and defense mechanism against covert channel attacks [231]. Our attack detection relies on the increased resource contention that is exhibited when a GPU is facing a covert or side channel attack. In the context of CPUs, Hunger et al. have already shown that resource contention is one of the most quantifiable impact of an attack [106]. As such, GPUGuard non-intrusively monitors resource contention across kernels through a set of well defined features and resource utilization metrics. It then uses these features and metrics to classify kernel interaction behaviors using a decision tree based design, and to identify covert and side-channel formation. The detection algorithm feeds the classification results to Tangram, a GPU-specific covert channel elimination scheme. Tangram uses a combination of warp folding, pipeline slicing, and cache remapping mechanisms to close the channels with 8%-23% performance overhead when there are active attacks, and 15% for normal benchmarks categorized as attacks with only a small (8.5%) false positive rate. In all other cases, GPUGuard pays nearly zero performance overhead.

1.1.4 Microarchitecural Attacks in Integrated CPU-GPU Systems

GPUs also come in integrated form in which they are tightly integrated in the same die as the CPUs, sharing some architectural resources such as last level cache and memory subsystem. This sharing opportunity provides the attackers with the opportunity to launch the attacks that are originated from one component and spy on the other, compromising security of the whole system. In this dissertation, we explore the feasibility of microarcitetural covert and side channel attacks between two asymmetric components in integrated systems. First, we build two covert channels: LLC-based PRIME+PROBE and contention based on memory bus across CPU and integrated GPU. Then we investigate the feasibility of more dangerous remote side channel attack in JavaScript, which is launched on integrated GPUs and uses the last level cache to spy on applications running on the CPU.

Although we demonstrate two instances of such cross-component attacks (specifically, on integrated GPU and CPU) in heterogeneous systems, the threat model can be extended to include any other accelerator or components, sharing resources with CPUs. Having experience with these channels improves our understanding of the threats posed of microarchitectural attacks beyond a single component which is a threat model increasing in importance as we move increasingly towards heterogeneous computing platforms.

In summary, the high-level contributions of this dissertation are:

- We reverse engineer several components and characterize the contention on different resources in discrete and integrated GPUs, including memory hierarchy, functional units, hardware schedulers and colocation of both graphics and computational applications.
- We build high quality and bandwidth covert channel attacks on a variety of shared resources on GPGPUs: caches, different types of functional units, and global memory.
- We implement a series of end-to-end side channel attacks on graphics and computational stacks of GPUs, and also across them: website fingerprinting attack, inter-keystroke timing attack to track user activity on the web browsers, and neural network model extraction attack.
- We propose an architectural defense framework to dynamically detect and mitigate covert and side channel attacks on GPUs.
- We present a new class of attacks that span different components within a heterogeneous systems: microarchitectural attacks in integrated CPU-GPU systems. We illustrate these attacks by building covert channels in native code and investigating the possibility of remote side channel in JavaScript.

1.2 Outline

We present a general background including GPU architecture and programming models in Chapter 2. Chapter 3 gives an overview of related work and discusses state-of-the-art attacks and defences on different components in heterogeneous systems. Chapter 4 and 5 present the covert and side channel attacks on discrete GPUs and our proposed mitigation is discussed in Chapter 6. In Chapter 7, we explore the possibility of microarchitectural attacks in integrated CPU-GPU systems. Finally, Chapter 8 concludes before highlighting potential future work.

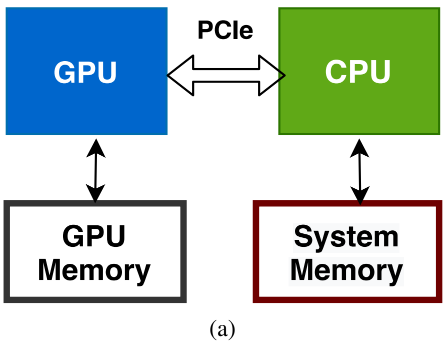
Our attacks demonstrate that covert and side channel vulnerabilities are not restricted to the CPU. Any shared component within a heterogeneous system can leak information as contention arises between applications that share a resource. Given the wide-spread use of GPUs in safety-critical applications, we believe that they are an especially important component to secure. Understanding of microarchitectural attacks and architecting security in heterogeneous systems including GPUs and other accelerators will become an essential design objective.

Chapter 2

GPU Overview

Graphics Processing Units (GPUs) are integral components to most modern computing devices including embedded systems, mobile phones, personal computers and workstations. They were primarily used to optimize the performance of graphics and multi-media heavy workloads. Their highly parallel structure makes them more efficient than central processing units (CPUs) to process large blocks of data in parallel. Therefore, General Purpose GPUs (GPGPUs) are also increasingly integrated on computing servers, computational clouds and clusters to accelerate a broad range of applications from domains including security, computer vision, computational finance, bio-informatics and many others [152].

GPUs come in two forms (1) discrete GPUs: a separate device, which is connected with the rest of the systems typically using a PCIe bus (or NVLink in new generations of Nvidia GPUs [168]), which addresses a separate physical memory; (2) Integrated GPUs (iGPUs): which are built on the same die as the main CPU and physically share the same system memory. Figure 2.1 demonstrates how discrete and integrated GPUs are connected to the the CPU.



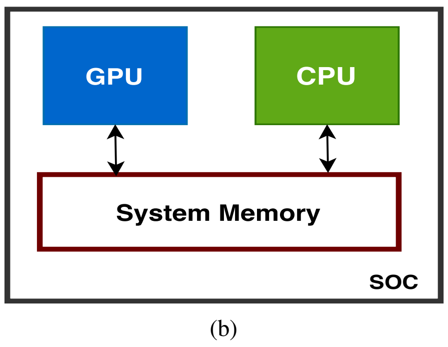


Figure 2.1: GPU in system (a) Discrete GPU; (b) Integrated GPU

This chapter overviews GPU programming interfaces, architecture, and multiprogramming support to provide an idea about how they are programmed, and how resources are shared and contention arises within them.

2.1 GPU Programming Interfaces

GPUs were originally designed to accelerate graphics and multimedia workloads. They are usually programmed using application programming interfaces such as OpenGL for 2D/3D graphics [35], or WebGL [38] which is usable within browsers. We call OpenGL/WebGL and similar interfaces the graphics stack of the GPU. OpenGL is accessible by any application on a desktop with user level privileges. On embedded systems like smartphones and tablet computers, a modified version of OpenGL is available called OpenGL-ES [43].

In the past few years, GPU manufacturers have also enabled general purpose programmability for GPUs, allowing them to be used to accelerate data intensive applications using programming interfaces such as CUDA [24], OpenCL [31], and Vulkan [37]. We call this alternative interface/software stack for accessing the GPU the computational stack.

Computational GPU programs are used widely on computational clusters, and cloud computing systems to accelerate data intensive applications [26]. These systems typically do not process graphics workloads at all since the machines are used as computational servers without direct graphical output. Nowadays, most non-cloud systems also support general purpose computing on GPUs [23] and are increasingly moving towards GPU concurrent multiprogramming.

On desktops or mobile devices, general purpose programmability of GPUs requires installation the CUDA software libraries and GPU driver. Nvidia estimates that over 500 Million installed devices support CUDA [24], and there are already thousands of applications available for it on desktops, and mobile devices.

2.2 GPU Architecture

Figure 2.2 presents an architecture overview of a discrete GPU. There are a number of resources that are shared between threads based on where they are mapped within the GPU. The GPU consists of a number of Graphical Processing Clusters (GPCs) which include some graphics units like raster engine and a number of Streaming Multiprocessor (SM) cores. Each SM has several L1 caches (for the instructions, global data, constant data and texture data). They are all shared among the computational threads mapped to it. There is a globally shared L2 cache to provide faster access to memory. As a typical example, the Nvidia Tesla K40 (Kepler generation), includes 15 SMs [30]. The size of the global memory, L2 cache, constant memory and shared memory are 12 GB, 1.5 MB, 64 KB and 48 KB respectively.

To illustrate the operation of the architecture, we describe how a general purpose function, written in CUDA or OpenCL, is run on the GPU. A CUDA application is launched using a CUDA runtime and driver. The driver provides the interface to the GPU. As demonstrated in Figure 2.3, a CUDA application consists of some parallel computation kernels representing the computations to be executed on the GPU. For example, a CUDA application may implement

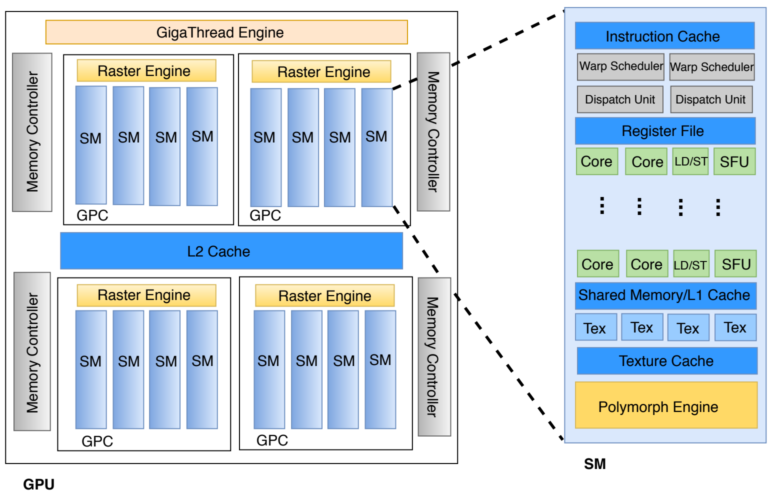


Figure 2.2: GPU architecture overview

parallel matrix multiplication in a computation kernel. Each kernel is decomposed into blocks of threads (CTA: Cooperative Thread Array) that are assigned to different SMs.

Internally, the threads are grouped into *warps* of typically 32 threads that are scheduled together using the Single Instruction Multiple Thread (SIMT) processing model to process the portion of the data assigned to this warp. The warps are assigned to one of (typically a few) warp schedulers on the SM. In each cycle, each warp scheduler can issue one or more instructions to the available execution cores. Depending on the architecture, each SM has a fixed number of various types of cores such as single precision cores, double precision cores, load/store cores and special functional units. Depending on the number of available cores an instruction takes one or more cycles to issue, but the cores are heavily pipelined making it possible to continue to issue new instructions to them in different cycles. Warps assigned to the same SM compete for access to the processing cores. In addition, warps assigned to the same warp scheduler may compete for the issue bandwidth of the scheduler.

The GPU memory is shared across all the SMs and is connected to the chip using several high speed channels (see memory controllers in Figure 2.2), resulting in bandwidths of several hundred gigabytes per second, but with a high latency. The impact of the latency is

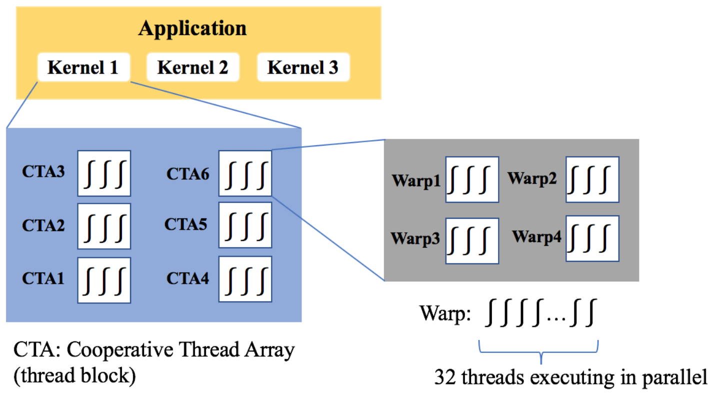


Figure 2.3: GPU programming model

hidden partially using caches, but more importantly, the large number of warps/threads ensures the availability of ready warps to take up the available processing bandwidth when other warps are stalled waiting for memory. This results in fine granularity and frequent interleaving of executing groups of threads.

With respect to graphics workloads, the application sends the GPU a sequence of vertices that are grouped into geometric primitives: points, lines, triangles, and polygons. The shader programs include vertex shaders, geometry shaders and fragment shaders: the programmable parts of graphics workloads that execute on SMs on the GPU. The GPU hardware creates a new independent thread to execute a vertex, geometry, or fragment shader program for every vertex, every primitive, and every pixel fragment, respectively, allowing the graphics workloads to benefit from the massive parallelism available on the GPU.

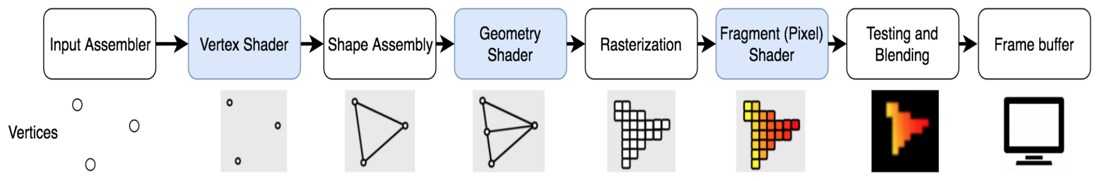


Figure 2.4: Graphics processing pipeline

Figure 2.4 demonstrates the logical graphics pipeline. The vertex shader program executes per-vertex processing, including transforming the vertex 3D position into a screen position. The geometry shader program executes per-primitive processing and can add or drop primitives. The setup and rasterization unit translates vector representations of the image (from the geometric primitives used by the geometry shader) to a pixel representation of the same shapes. The fragment (pixel) shader program performs per-pixel processing, including texturing, and coloring. The output of graphics workloads consists of the pixel colors of the final image and is computed in fragment shader. The fragment shader makes extensive use of sampled and filtered lookups into large 1D, 2D, or 3D arrays called textures, which are stored in the GPU global memory. The contention among the different threads carrying out operations on the image is dependent on the image.

2.3 Multiprogramming on GPU

Multiprogramming is required for our threat model since the spy and the trojan/victim are different programs that are running concurrently on the same GPU and trying to communicate indirectly or one of them extracts sensitive information from the other. Recent research has shown that there are significant performance advantages to supporting multiprogramming on GPUs [199, 218, 230]. For these reasons we believe that GPU manufacturers are moving to support multiprogramming.

The current generation of GPUs supports multiprogramming, or the ability to run multiple programs at the same time, through multiple streams with multi-kernel execution within the same process, or a multi-process service (MPS) [167], which allows concurrent kernels from different processes. MPS is already supported on GPUs with hardware queues such as the Hyper-Q support available on Kepler and newer microarchitecture generations from Nvidia, and is being improved in every generation. Volta GPUs [20] features the multi-process service with hardware support to dedicate separate address spaces to different processes. AMD multiuser GPUs [16] and NVIDIA vGPUs [15] both enable up to 16 concurrent clients to share a GPU. NVIDIA GPUs support asynchronous compute to concurrently run graphics and general purpose workloads since Maxwell [12]. This increasing support on GPU multiprogramming makes our attacks more dangerous and relevant.

Chapter 3

Related Work

We organize the discussion of related work into four different categories: (1) Covert and side channel attacks and defenses on CPUs (We briefly discuss the works that are most related to our attacks in this dissertation) (2) Side channel attacks and defenses on GPUs and Heterogeneous systems, (3) Attacks on GPU software stack; and (4) Timing and WebGL based attacks in web browsers.

3.1 Covert and Side Channels on CPUs: Attacks and Defenses

Microarchitectural side-channel attacks have been widely studied on a variety of resources on CPUs, including CPU L1 cache [181, 61, 78], shared LLC in multi-core CPUs [120, 143, 98, 139, 237, 239], branch predictors [82, 83], random number generators [80], and others [65, 122]. Several works exploit cache coherency protocols to develop timing channels on multi-core CPUs [234, 232] or multi-CPU systems [90].

There are many defense proposals to close side channel attacks on the CPUs which mostly focus on caches and memory controllers. These proposals include: (1) Static or dynamic partitioning of resources like L1 cache [76, 174, 185, 109] that can introduce unacceptable performance overhead and can only support a limited number of partitions with reasonable overhead and mitigation mechanisms like locking the critical cache lines with the support of OS and compiler [126, 216]. Liu et al. [141] proposes partitioning the LLC into secure and non-secure partitions and line locking the secure partition for defeating side channels. (2) Randomizing memory-to-cache mapping, including randomization in the replacement of the cache lines in the entire cache [217] and in the cache fill strategy [142]. (3) Adding noise to timing by manipulating time measurement structure of processor [148]. (4) Traffic control in memory controllers [213, 191]. Such defenses do not transit directly to GPUs or to covert channels. We propose a solution in Chapter 6 which uses GPU-appropriate forms of partitioning triggered by detection of covert communication.

Online detection of contention based covert communication is an alternative that is useful for closing covert channels. Chen et al. [65] present a framework to detect timing covert channel on shared hardware resources on a CPU by dynamically monitoring conflict patterns between processes. However, their framework is designed to detect alternating pattern of cache conflicts between Spy and Trojan and not to detect any variations of the attack—those channels that access to other shared resources concurrently. Yan et al. [233] propose a record and deterministic replay framework. It detects timing attacks by replaying execution on a different cache configuration to detect contention only on caches. In contrast, our presented solution in Chapter 6 monitors contention on all known resources as it occurs. There are also a number of online detection schemes based on hardware performance counters [72, 173, 121]. Unlike these prior solutions that only focus on malware detection, we present a framework that provides mitigation solutions for preventing information leakage through covert channels.

There are also a number of defenses against timing attacks that seek to equalize the performance of a single shared resource to hide the contention [213, 191, 212]. Due to the availability of multiple shared resources in GPUs, attackers will simply shift to use a different resource if one is available. Hence, we need a unified approach to mitigate covert channels across all shared resources. Moreover, many solutions require turning off hyperthreading or simultaneous multithreading (SMT) to minimize resource sharing within a core. Our proposal retains SMT to improve latency hiding, while ensuring security.

3.2 Side Channels on GPUs and Heterogeneous Systems: Attacks and Defenses

3.2.1 Side Channel Attacks on GPUs

Jiang et al. [114] conduct to our knowledge the first timing attack at the architecture level on GPUs. The attack exploits the difference in timing between addresses generated by different threads as they access memory: if the addresses are *coalesced* such that they refer to the same memory block, they are much faster than uncoalesced accesses which require several expensive memory operations. Execution time relies on the number of unique memory requests after coalescing. Thus, in a cryptographic application, the key affects the address pattern accessed by the threads, and therefore the observed run time of the encryption algorithm, opening the door for a timing attack where the encryption time is used to infer the likely key. The same group [115] presented another timing attack on table-based AES encryption. They found correlation between execution time of one table lookup of a warp and a number of shared memory bank conflicts generated by threads within the warp. They use these key-dependent differences in timing to correlate measured execution time to the key at the last round of the AES encryption as it executes on the GPU.

The self-contention exploited in the these attacks [114] cannot be used for a side-channel between two concurrent applications. Although memory coalescing and shared memory bank conflicts make a large difference in the timing of one kernel, these artifacts had little measurable effect on the timing of a competing kernel.

Luo et al. [146] present a timing side channel attack on GPU accelerated RSA encryption. They use two existing optimizations: Sliding Window Exponentiation and Montgomery Multiplication and exploit the correlation between total execution time of a message decryption and number of reductions in each window of decryption to extract the RSA private key.

These timing channels are measured from the CPU side, limiting their bandwidth, and requiring each kernel to be able to launch (or anticipate the launch) of the other to time it. In this dissertation, we explore a different threat model that investigates general covert and end-to-end side channels between two concurrent applications on the GPU.

Gao et al. [87, 88] and Luo et.al [147] study Electro-Magnetic and power side channel attack on AES encryption executing on a GPU. These attacks requires physical access to the GPU to measure the power or Electro-Magnatic traces.

Wang and Zhang [211] propose a profiling-based side-channel attack to fully recover the AES encryption secret key. Rather than execution time, they profile two performance matrices, the number of the memory load and memory store requests. Based on these, the number of unique memory load requests in the last round encryption for each byte except the first byte then can be determined. They recover 16-bytes AES key byte by byte. They calculate the number of unique memory load requests with 256 possibilities of a single byte of the AES key and compare it to the profiled number. By repeating the same procedure with different input data, they can successfully extract the exact byte key.

After publishing our side channel on neural network model extraction (Chapter 5), Wei et al. [221] conducted similar attack to infer the hyper-parameters of a Deep Neural Network model. Rather than concurrent running of applications on GPU (in our threat model), they consider fine grained time-sliced sharing of applications and exploit context-switching penalties on performance counters as leakage vector. Although, in direct response of our side channel attack, Nvidia released a patch [42] in their new GPU drivers to disabling the normal users to access to the performance counter, they bypass this patch by downgrading the GPU driver on the spy VM which is invisible to the victim. Zou et al. [247] also utilize performance counters on GPUs as features fed to machine learning based classification model to classify running workloads on GPU accelerated HPC systems.

Side channel attack has also been studied in virtualized GPU environment. After publishing our side channel attacks, Liu et al. [144] presented a side channel attack from one virtual machine to another where both share the same physical GPU, and each virtual machine has its own GPU system stack, making the attack more difficult. Because the weakness in GPU library no longer exists and GPU hardware performance counters are (typically) not available to VMs. They conduct a side channel on intel integrated GPUs. They launch an OpenCL based probing application which does some read-compute-write pattern of operations to create contention on different resources on GPU, and measure the execution time as a coarse grained information leakage. Then utilize machine learning approaches to identify the victim's GPU workload among a small dataset of several entertainment and deep learning workloads.

3.2.2 Side Channel Defenses on GPUs

Kadam et al. [117] proposed Rcoal, a redesign of GPUs to eliminate predictable memory coalescing behavior to prevent side-channels such as those used in the attack presented in [114]. They propose the memory coalescing randomization techniques in several ways including the number of subwarps, the threads assigned to each subwarp, or a combination of both. These randomization generate additional accesses and alleviate such correlation-based timing attacks by making the relationship between execution time and coalesced memory accesses less predictable.

Wang and Zhang [210] developed profiling based side channel attacks against RCoal focusing on the configurations with high variance in the number of coalesced accesses. Kadam et al. [118] propose BCoal, a mechanism to further reduces the variance making it a much stronger defense and to efficiently address the limitations of RCoal in terms of performance degradation. BCoal is a bucketing-based coalescing technique which generates the number of coalesced accesses equal to one of the pre-determined values (known as buckets), irrespective of program secrets. As the number of accesses is always equal to the pre-determined values, the variance in the number of accesses drops, reduces the correlation in timing attack. To reduce the performance overhead of additional accesses, they select optimal bucket features by analyzing the application level coalescing profile, such that overall fewer additional accesses are generated.

To protect the GPU accelerated AES encryption against both timing and cache based side channels, Lin et al. [137] proposed a new software-based mechanism. They rely on Scather and Gather approach which slices and re-organizes the pre-computation tables such that key-dependent table lookups will not leak any timing or address pattern information. This software based approach is specific to AES encryption. On the other hand, Rcoal and BCoal are generic hardware based coalescing mechanism applicable to all security-sensitive GPGPU applications that are vulnerable to coalescing-based correlation timing attacks.

3.2.3 Attacks on Integrated Heterogeneous Systems

In integrated heterogeneous systems, accelerators (such as GPUs and FPGAs) are tightly integrated on the same die as CPUs and share some hardware resources such as last level cache and memory subsystem. This sharing opportunity lead to cross-component microarchitectural attacks. Weissman et al. [223] study Rowhammer and last level cache based attacks on heterogeneous CPU-FPGA platforms.

Gravellier et al. [94] used FPGA voltage sensors to implement a remote power based side channel attack running on FPGA to spy on CPU computation, specifically to retrieve the secret key of AES crypto-algorithm. Zhao and Suh [242] demonstrates the same attack on RSA encryption, from FPGA to CPU in SoC. Remote power based attacks assume that the adversary has access to some of the LUTs in the remote FPGA shared with the victim and can implement a power monitor on the FPGA fabric.

The integrated GPU is available through APIs such as WebGL [38] even for remote JavaScript programs making this threat vector extremely dangerous. Frigo et al. [86] use WebGL timing APIs to implement GPU accelerated Rowhammer attack on memory in integrated CPU-GPU systems in mobile SOCs. They use a timing channel to find contiguous area of physical memory rather than extracting application state like our attacks in this dissertation, totally different threat model.

3.3 Other Attacks on GPU and GPU Software Stack

GPUs are a type of accelerator device; specialized accelerators are expected to become increasingly important components of the computing landscape due to their superior performance and power properties for specific application classes. Olson et al [171] developed a taxonomy of vulnerabilities and security threats for accelerators based on threat types into attacks that affect Confidentiality, Integrity, or Availability. They also classify the risk categories in terms of what part of the accelerator attacks affect. Although the paper offers no concrete attacks, it highlights that security of accelerators warrants significant attention. Olson et al. [170] also propose sandboxing accelerators when the CPU and the accelerators share the same address space (e.g., under a Heterogeneous System Architecture configuration). This type of defense does not protect against side- and covert-channel vulnerabilities.

Mittal et al. [193] present a survey of techniques for analyzing and improving GPU security. They classify GPU security vulnerabilities and attacks and discuss potential countermeasures. In this section, we review a number of attacks that have been published targeting GPUs or their software stack on the CPU.

3.3.1 Leftover Memory Leakage

Lee et al. [131] reveal three major security threats in GPUs including: lack of initialization of newly allocated memory pages, un-erasable portion of GPU memory (e.g., constant data) and lack of prevention of threads of a kernel to access the contents stored in the local and private memories, written by threads of other kernels. When multiple users share the same GPU, there is information leakage between concurrently running processes or from processes that recently terminated. They utilize information leakage that occurs due to not clearing newly-allocated memory in the GPU to extract rearranged webpage textures of Chromium and Firefox web browsers (both use GPU-accelerated rendering) from Nvidia and AMD GPUs. In particular, by using End-of-Context (EoC) attack, the attacker can easily obtain the computation results (e.g. rendered image), if the victim program does not clear its global memory before termination. In addition, End-of-Kernel (EoK) attacks are used to obtain intermediate data stored in per-SM local memory during long-running GPU kernels; since GPU runs long kernels as several kernels or the same kernel repeatedly. Similar attacks are presented by Di Pietro et al. [183] who also exploit non-zeroed state available in shared-memory, global-memory and registers. As a case study, they investigate the impact of this vulnerability on a GPU implementation of the AES encryption algorithm. Using this vulnerability, an adversary can recover data of a previously executed GPGPU application in a variety of situations. Wenjian HE et al. [104] exploit the similar vulnerability on integrated GPUs in which adversaries can steal the key of AES encryption and implement a website-fingerprinting attack against the Chrome browser. Zhou et al. [245] exploit this vulnerability to recover victim's data directly from the GPU memory residues of four popular applications: Google Chrome, Matlab, GIMP and Adobe PDF reader. Maurice et al. [150] show that one of these dangerous situations is GPUs in virtualized and cloud computing environments where the adversary launches a virtual machine after the victim's virtual machine using the same GPU.

This class of vulnerability can be closed by clearing memory before it gets reallocated to a different application. So in modern GPUs and programming platform, this type of vulnerability is likely to be removed.

3.3.2 GPU-based Malware

As malwares are getting more robust to evade detection by current anti-malware defenses and antivirus scanners, GPUs can provide great potential for malware writer to obfuscate their code. Vasiliatis et al. [204] implement GPU based packing and polymorphism, two most widely used techniques for evading malware scanners. Balzarotti et al. [56] also demonstrate that it is possible to successfully hide some malicious behavior, by offloading some computation to the GPUs. They provide four different techniques that a malware can evade detection on Intel Integrated GPUs: 1) Unlimited code execution, 2) Inconsistent memory mapping, 3) Process-less execution, and 4) Context-less execution. Danisevskis et al. [71] present DMA-based malware on mobile GPUs. An attacker exploit DMA and a bug in Android smartphone's GPU driver to bypass memory protection mechanism of GPU and gain access to privileged region of memory for reading or writing.

Ladakis et al. [130] describe a GPU-based keylogger, a malware that log keyboard activity for stealing sensitive data. The attacker monitor the systems' keyboard buffer directly from the GPU, using DMA. They used a CPU process to control the execution of their malware which requires root privileges to initialize the environment.

Because GPUs have separate physical resources, PixelVault [205] proposes using them as secure co-processors. Zhu et al. [246] show that GPUs are vulnerable to disclosure attacks through the driver from a privileged user on the CPU, bringing into question the security of the PixelVault model. Such attacks require root access and are outside our threat model in this dissertation.

To protect data against GPU malware and information leakage, Hayes et al. [102] present a taint tracking system on GPUs. They instrument programs statically on a per-application basis and when the program runs, every thread can dynamically track information flow by itself. They utilize GPU characteristics and architecture to optimize the taint tracking performance.

3.3.3 GPU Buffer Overflow

Miele [154] study buffer overflow vulnerabilities in CUDA software and exploit it to hijack GPU control flow. An attacker can overwrite function pointers to his/her injected code to take control of the GPU's operation.

Di et al. [74] also explore overflow vulnerabilities on GPUs software, specifically CUDA. In particular, they exploit stack and heap overflow, such that threads from the same warp or different blocks overwrite each other's content, which steers the execution flow. They also show that integer overflow can be used to overwrite a function pointer in a *struct*. However, format string and exception handling vulnerabilities are not exploitable on the GPU due to the limited support in CUDA.

As Modern GPUs share virtual, and sometimes physical, memory with CPUs, GPU-based buffer overflows are very dangerous threats, capable of producing program crashes, data corruption, and security problems on CPU as well. Erb et al. [79] present a tool to detect buffer overflows caused by OpenCL GPU kernels. This tool uses canaries to alert the users when any write occurs outside of a memory buffer. They minimize the overhead of the detector using GPU kernels to check canary values for overflows.

3.4 Attacks on Web Browsers

Different attack vectors have been proposed for website fingerprinting. Panchenko et al. [177] and Hayes et al. [103] capture traffic generated via loading monitored web pages. Felten and Schneider [85] utilize browser caching and construct a timing channel to infer the victim visited websites. Jana and Shmatikov [111] use the procfs filesystem in Linux to measure the memory footprints of the browser. Then they detect the visited website by comparing the memory footprints with the recorded ones. Weinberg et al. [222] presented a user interaction attack (victim's action on website leaks its browsing history) and a timing side channel attack for browser history sniffing. Leakage through the keystroke timing pattern is also a known effect which has been exploited both as a user authentication mechanism [156, 179, 69]. Keystroke timing has also been used to compromise/weaken passwords from the keystroke timing [192] or compromise user privacy [63]. In Chapter 5 we show how we develop GPU-based side channels to track user activity on the web through website fingerprinting and keystroke timing attack. Also, in Chapter 7, we demonstrate feasibility of GPU based attacks in JavaScript. Next, we overview some related work to this threat model.

3.4.1 Timing Attacks in JavaScript

Side channels, specifically timing attacks have been widely studied in JavaScript. Oren et al. [172] implement a prime+probe attacks on last level cache in JavaScript. They build cache covert channels and side channel to spy on the user's mouse movements and network activity through the cache. They use JavaScript's timer [44] to carry out timing measurement. Based on this timing interface, various attacks have been demonstrated. Goethem et al. [202] and Bortz et al. [60] propose cross-site timing attacks on web browsers to estimate the size of cross-origin resources or provide user private information leakage from other site. Gulmezoglu et al. [99] present a side channel on per-core/per-process CPU hardware performance counters (which are limited in number). Stone et al. [196] propose two timing channels by measuring the time elapsed between frames using the JavaScript API. They either detect the redraw events or measure the time of applying the SVG filter to perform history sniffing. These attacks are difficult currently since most browsers have reduced the timer resolution eliminating the timing signal used by the attack.

To protect against these timing attacks, all major browsers limited the resolution of the timer [68, 13, 240]. This low precision timer is still sufficient for conducting some attacks. Gruss et al. [96] proposed memory page deduplication timing attack to determine which websites the user has currently opened. More recently, Vila and Kopf [206] present a side channel attack on shared event loop in which the attacker enqueues several short tasks, and records the time these tasks are scheduled. The time difference between two consecutive tasks reveals the existence of the victim and duration of its task. They exploit this side channel to identify web pages, to build a covert communication channel, and to infer inter-keystroke timings.

However, to distinguish the cache hits from cache misses in a cache attack, a high resolution timer is required. Several works propose some timing primitives in JavaScript to recover highly accurate timestamps [93, 138, 125, 190]. Lipp et al. [138] propose a keystroke interrupt-timing attack implemented in JavaScript using a counter as a high resolution timer. Kohlbrenner et al. [125] study the clock-edge technique. They use the degraded JavaScript timer as a major clock to observe the edges and a tight incrementing for loop as the minor clock to build a high resolution timer in JavaScript. They also propose *FuzzyFox* as a mitigation on web browsers to mitigate all clocks, which introduces randomness in the JavaScript event loop to add noise to timing measurements. Gras et al. [93] proposed two mechanisms (Shared memory counter and time to tick) to craft high resolution timer in JavaScript. They use a dedicated JavaScript web worker for counting through a shared memory area (SharedArrayBuffers [158] interface) between the main JavaScript thread and the counting web worker. They showed that accurate timing information in JavaScript can be exploited to defeat address-space layout randomization. Schwartz [190] also use similar techniques to build high resolution timer and implement a new DRAM-based covert channel between a website and an unprivileged app. To respond these attacks, major browser vendors disabled the SharedArrayBuffers interface in JavaScript [28, 84].

All of these attack models are proposed at browser/OS level or CPU hardware level, providing different attack vector than our attacks which target the GPU hardware to extract sensitive information through side channels. We demonstrate that GPU side channels threaten not only graphics applications, but also computational workloads running on GPU.

3.4.2 WebGL Attacks

WebGL is a JavaScript API to accelerate rendering 3D and 2D graphics within web browser [38]. This APIs enables the attacker to launch GPU-based attacks in JavaScript, leading to dangerous remote attacks.

Yao et al. [235] summarize all WebGL vulnerability disclosures and find that most are related to reading GPU memory (either uninitialized or other process's memory). Based on this analysis and the WebGL security analysis by the Khronous group [39], no GPU side channels have been reported in the context of WebGL. Yao et al. leverage modern GPU virtualization to secure GPU acceleration in the web browser by separating WebGL computations into separate virtual GPUs. Yao et al. [236] also propose Milkomeda, a solution to automate and improve the existing WebGL security checks to further increase the security of the mobile graphics interface.

WebGL API has been utilized to implement browser fingerprinting [62, 196]. Browser fingerprinting refers to the process of collecting information through a web browser to build a fingerprint of a device [182].

To protect against these attacks, Wu et al. [227] study the WebGL API to identify the source of these attacks. They found out the results of floating-point operations can vary between devices across the various graphics layers of a system. They propose a purely software solution called UniGL to protect against webGL based browser fingerprinting. UniGL redefines floating operations explicitly written in GLSL programs or implicitly invoked by WebGL, such that every device running UniGL will have the exact same WebGL fingerprint for a specific rendering task. Although these defenses are able to contain software vulnerabilities within the WebGL stack, it cannot close side channels or micro-architectural attacks.

Frigo et al. [86] use WebGL timing APIs to implement GPU accelerated Rowhammer attack on memory in integrated CPU-GPU systems in mobile SOCs. They use WebGL timer as the major clock and implement a variant of clock-edging [125, 190] that executes a (padding) count down over an empty loop before checking for the new timestamp value. This WebGL timer [123] is driver- and browser dependent. Firefox supports it, while Chrome disables it due to compatibility issues [84]. In Chapter 7, we build a customized high resolution timer using GPU hardware resources, to enable GPU-based microarchitectural attacks in JavaScript, even without accessing to WebGL timers.

Chapter 4

GPU Covert Channel

Often applications that use GPGPUs operate on sensitive data [59, 73, 164], which can be compromised by security vulnerabilities present in GPGPUs. In this Chapter, We present a first study of vulnerability of GPGPUs to covert channel attacks. A covert channel attack may enable a malicious application (called Trojan) without network access to communicate data to another application (called Spy) to exfiltrate the data off the device. Covert channel attacks are dangerous because they allow intentional communication of sensitive data between malicious processes that have no direct channel between them.

Alternatively, covert communication can be used to bypass protections that track exposure of sensitive information such as sandboxing or information flow tracking, allowing sensitive data to escape containment [77]. The presence of a covert channel can also forecast the possibility of a side-channel attack that we explore in the next Chapter.

With multiprogramming starting to be available on GPUs [218, 230], covert channel attacks between two kernels running concurrently on a GPU become possible. The attack offers a number of advantages that may make them an attractive target for attackers compared to CPU covert channels including: (1) With GPU-accelerated computing available on major cloud platforms such as Google Cloud Platform, IBM cloud, and Amazon web service [166] this threat is substantial [187]. The model of sharing GPUs on the cloud is evolving but allowing sharing of remote GPUs is a possibility [75, 186, 58, 180]. Therefore, GPU covert channels may provide the attackers with additional opportunities to co-locate, which is a pre-requisite for these types of attacks [187]; (2) GPGPUs operate as an accelerator with separate resources that do not benefit from protections offered by an Operating system. In fact, due to this property they have been proposed for use as a secure processor [205]; and (3) GPGPU channels can be of high quality (low noise) and bandwidth due to the inherent parallelism and, as we demonstrate, the ability to control noise.

At the same time, constructing covert channels on GPGPUs introduces a number of challenges and operational characteristics different from those on CPUs. One of the new challenges is to how to establish co-location between the trojan and the spy by exploiting the hardware schedulers such that the communicating kernels can share resources. Thus, we first reverse engineer the hardware scheduling algorithms that determine where the different blocks and warps can be allocated to create contention (Section 4.2). A second problem is to identify which resources are most effective for communication given the throughput bound nature of GPGPUs. In particular, GPUs have substantial parallelism, which may enable high throughput covert communication, but only if effective isolated contention domains can be found. For example, a shared resource, such as the L2 cache, may have limited capacity limiting the bandwidth of communication through it. Alternatively, a resource may have high capacity (such as the memory bandwidth), making it difficult to create measurable contention. We construct different channels using contention on caches, contention for computational units, as well as contention for memory operations in Sections 4.3, 4.4 and 4.5 respectively.

Having demonstrated and characterized these different channels on three different GPGPUs, we explore improvements to the channel bandwidth. We use the inherent parallelism in GPGPUs to increase the bandwidth of the channel. We also implement synchronization to increase the robustness of the communication to increase the communication efficiency in Section 4.6. To improve resilience to interference from other applications, we propose exploiting the hardware schedulers to block out co-location from other applications; this is a new approach to managing noise unique to GPGPUs (Section 4.7). We demonstrate the success of exclusive co-location on current GPGPUs which support multiprogramming based on leftover policy and discuss some scenarios to extend our attack for proposed multiprogramming schemes on future GPGPUs, such as kernel preemption and intra-SM partitioning. Finally, Section 5.5 discusses the potential mitigations.

4.1 Threat Model

Our threat model consists of a standard covert communication scenario with a trojan and spy kernels from two different applications that co-exist concurrently on the same GPU. The two kernels wish to communicate covertly. We consider a case where the trojan and the spy are the only two applications running on the GPU to characterize the bandwidth of the channels under the best case scenario. We later relax that assumption and explore approaches to prevent or tolerate noise from other applications. We assume that the two kernels can launch applications to the same GPGPU; in a cloud setting a first problem is to establish this ability. Since no standard sharing model of GPGPUs in the cloud has emerged, we do not focus on this problem [75, 186, 58, 180]. In most existing settings, the GPGPU is shared among applications on the same physical node, via I/O pass-through. In such settings, the problem boils down to achieving co-location on the same cloud node [187]. In the case of non-cloud scenarios, typically a GPGPU is also shared among applications on the same machine/device.

4.2 Establishing Co-Location

Our goal is to create covert channels through contention on shared hardware resources in the GPGPU. As a first step, we need to control the placement of concurrent applications –in our case, the trojan and the spy. The placement defines what resources the applications share and therefore what covert channels are available for use. In this section, we show how to establish co-location as the first step in constructing covert channels. In particular, we reverse engineer the block assignment algorithm on real GPUs and show how to exploit it to establish co-location. It is likely that future GPGPUs may use alternative placement algorithms; however, we believe that the reverse engineering approach we use can be used for not only Nvidia GPUs, but also a large class of placement algorithms that are both deterministic and do not use preemption. We support this claim by considering the scheduling algorithms for other multiprogrammed GPUs that were recently proposed in literature [218, 230].

4.2.1 Co-location on Existing GPGPUs

The Nvidia thread block assignment and kernel co-location algorithms are unpublished; thus, it is necessary to reverse engineer the placement algorithm. First, we explore whether blocks belonging to two kernels can be co-located on the same SM. We launch two kernels on different streams. In each kernel, we read the SM ID register (`smid`) for each block to determine the ID of the SM on which the kernel is running. In addition, we use the `clock()` function to measure the start time and stop time of each block. By using this information, and repeating the experiment for different numbers and configurations of blocks, we reverse engineered the placement algorithm.

We found that the blocks for the first kernel are assigned to different SMs in a mostly round-robin manner. If there are SMs that are idle, or that have a leftover capacity, they can be used for blocks of the second kernel, again in a mostly round-robin assignment. Otherwise, the blocks of the second kernel are queued until at least one SM is released. Therefore, if each kernel is launched with a number of blocks equal to or exceeding the number of SMs on the device, such that each block does not exhaust the resources of the SM, they achieve co-residency within an SM. This multiprogramming mechanism on current GPUs is called the Leftover policy.

We also discovered that there is another level of sharing within the SM that impacts the contention behavior. In particular, on many GPGPUs, there are a number of warp schedulers available on each SM. Each warp is associated with one of these warp schedulers. If different warps share the same scheduler, we show that their contention behavior is different since the warps on the same scheduler compete for the issue bandwidth that is assigned to the scheduler.

We experimented with the assignment algorithm of warps to warp schedulers and discovered that it is also round robin. With this knowledge, the spy and the trojan can set up their kernel parameters to achieve co-location on the same SM and if desired on the same warp scheduler. For example, on the Tesla K40C, with 15 SMs and 4 warp schedulers, if each of the spy and the trojan launch a kernel with 15 blocks each using 4 warps (i.e., 128 threads), they will each have a warp on each of the warp schedulers of each of the 15 SMs on the GPGPU.

4.2.2 Co-location on other GPGPUs

The left-over policy allows co-location of kernels opportunistically. The current generation of GPUs supports multiprogramming, or the ability to run multiple programs at the same time, through multiple streams with multi-kernel execution within the same process, or a multi-process service (MPS) [167], which allows concurrent kernels from different processes. MPS is already supported on GPUs with hardware queues such as the Hyper-Q support available on Kepler and newer microarchitecture generations from Nvidia. To provide a uniform implementation including Fermi GPUs, we utilized streams for multiprogramming on GPU to develop covert channels.

Recent papers [218, 230] improve multiprogramming on GPUs using intra-SM resource partitioning that execute multiple kernels to more effectively utilize the GPU. We believe that this approach of varying the configuration of launched kernels and observing how they are scheduled can be used to reverse engineer any deterministic and non-preemptive co-location algorithm. In this section, we consider the co-location problem relative to these proposed schedulers.

Wang et al. [218] support a simultaneous multi-kernel by fine grained context switching at the granularity of thread block. To schedule thread blocks of the new kernel to an SM, those thread blocks of previously scheduled kernels that have the highest resource usage on the victim SM may be preempted. Thus, this scheduler makes co-location easier by allowing the spy and the trojan to reside on the same SM even if other applications are already running there. By using just one thread block for each spy and trojan on each SM, the spy and trojan will be guaranteed not to be preempted. However, the co-location of other workloads on the same SM possibly adds noise to the covert channel. We discuss this issue in Section 4.7.

Xu et al. [230] propose a dynamic intra-SM resource partitioning that does not use preemption. Intra-SM partitioning attempts to co-schedule kernels that are compatible in their resource usage to the same SM. Since this multiprogramming scheme does not use preemption, we can force exclusive co-location of the two kernels by manipulating their initial behavior so that the scheduler finds them compatible.

We also consider a case where the two kernels cannot be co-located on the same SM, for example, in cases where there is no leftover capacity on the SMs used by the first kernel. In this case, covert communication is still possible through contention on resources that are shared between all SMs such as global memory or the L2 cache. This inter-SM covert channels can also be applied to some proposed GPU multiprogramming which allows executing multiple kernels only on disjoint sets of SMs on GPUs (i.e., no intra-SM scheduling). For example, Adriaens et al. [48] use inter-SM resource partitioning. Similarly, Tanasic et al. [199] propose kernel allocation at the granularity of the whole SM to support multiprogramming. It is likely that these multiprogramming mechanisms are not as efficient as intra-SM resource partitioning, due to the limited number of SMs and large overhead of context switching on GPUs. Moreover, multiprogramming at the granularity of the full SM cannot address resource under-utilization occurring within an SM.

4.3 Cache Covert Channels

In this section, we present the first of three classes of covert channels we investigate: covert channels through the caches. We illustrate the principles using constant caches, but the attack applies to other caches on the system. We selected constant memory because the size of both the L1 and L2 caches is small allowing us to create contention easily.

The attack proceeds in two steps. First, an offline step uses the microbenchmarking approach introduced by Wong et al. [225] to infer the constant memory and cache characteristics at each level of the hierarchy. The second step is the communication step where we use contention to create the covert channel.

4.3.1 Step I: Offline Characterization of Constant Memory Hierarchy

The parameters of the constant memory hierarchy at each level of the cache hierarchy can be extracted from latency measurement of loading different size arrays from constant memory using a strided access pattern. The cache is first warmed by accessing the array, which is subsequently accessed again while timing the accesses [225]. The size of the array is increased and the access latency observed.

Figure 4.1 and Figure 4.2 show the latency measurements for the L1 cache and L2 cache respectively on the Kepler Tesla K40C GPU. Each point on the figure represents an experiment with the array size shown on the x-axis. While the latency remains constant, the array fits in cache. When the array spills out of the cache, the latency starts increasing. First, the spill causes misses only in one set. As we keep increasing the array size, spills in additional sets occur: the number of steps in the figure is equal to the number of cache sets. The cache line size corresponds to the width of each step. From the cache size, number of cache sets and cache line size, we can calculate the cache associativity. For example, for the Kepler (Tesla K40) and Maxwell (Quadro M4000) GPUs we find that the constant memory L1 cache is 2kB, 4-way set associative with 64 byte cache line, while the L2 cache is 32kB 8-way set associative with 256 byte cache line. In the Fermi (Tesla C2075) GPU, constant memory L1 cache is 4kB, 4-way set associative with 64 byte cache line and L2 cache has the same parameters as in Kepler and Maxwell.

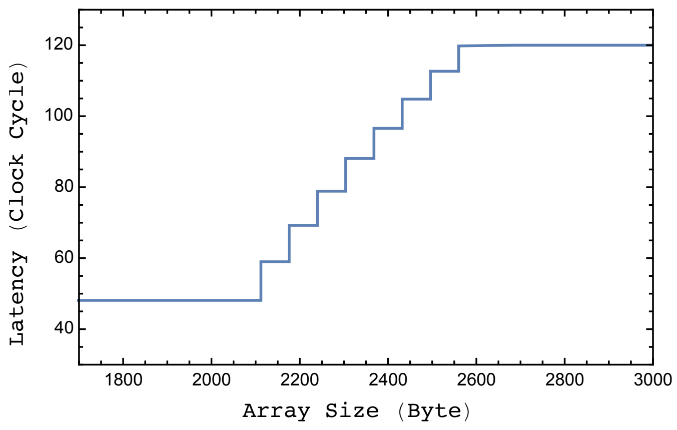


Figure 4.1: L1 constant cache, stride 64 bytes.

4.3.2 Step II: Constructing Covert Channel through L1

We set up an experiment with two concurrent kernels using different streams on the GPU. On the Kepler K40C device, there are 15 SMs, so we use 15 thread blocks for each kernel

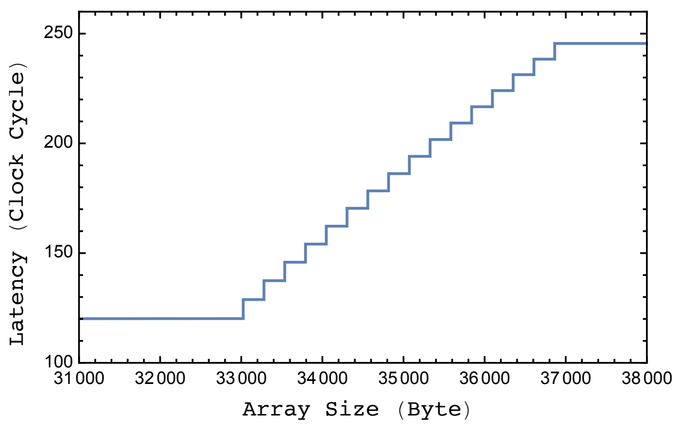


Figure 4.2: L2 constant cache, stride 256 bytes.

to be sure of co-residency on the same SM. The trojan kernel communicates by either creating contention or doing nothing to encode 1 or 0 respectively. To create contention on one set, the trojan allocates an array with the size of L1 cache (2 KB) and loads it with a stride of 512 bytes to make the accesses hash into the same set. The spy also loads a 2KB array with the same stride as the trojan while timing the access: a high latency indicates 1 since the array was replaced by the trojan, and a low latency indicates a 0. Our results show that in the case of contention (i.e. sending 1), the measured latency by the spy is about 112 clock cycles, but without contention (i.e. sending 0), the latency is 49 clock cycles. This difference allows the spy to easily determine the bit being transmitted. Note that we create contention over only a single set of the cache, rather than over the whole cache, reducing the memory traffic and accelerating the attack.

To communicate multiple bits, the trojan and the spy have to stay synchronized. Due to scheduling variability and/or the presence of noise, loss of synchronization can occur. To simplify this problem in this experiment, we launch two kernels to communicate each bit of the message. Clearly, this incurs some overhead to launch the kernels, but it simplifies synchronization by leveraging the stream operations, resulting in error free bandwidth of around 40Kbps. In Section 4.6, we use synchronization through covert communication (on different sets of the cache) to remove the need to continue to relaunch the kernels. Implementing this synchronization makes the attack more robust to noise, and more resilient to loss of synchronization; it also improves the channel bandwidth significantly.

4.3.3 Step II: Constructing Covert Channel through L2

When two kernels cannot be co-located on the same SM, they can still communicate through the L2 constant cache that is shared between all SMs. The process of creating a covert channel is the same as the L1 channel. However, the parameters of the L2 cache are different: we consider array size of 32kB and stride value of 4096 bytes (16 sets $\times$ 256 bytes) to fill just one cache set. The measured bandwidth in this scenario is about 20Kbps.

Figure 4.3 shows the bandwidth achieved by the attack on three Nvidia GPGPUs selected from three generations of microarchitecture (Fermi, Kepler and Maxwell). We modified the attack to fit the cache parameters, but otherwise left it unchanged. All bits were received correctly with no errors. Thus, high bandwidth covert channels are feasible through both levels of the cache.

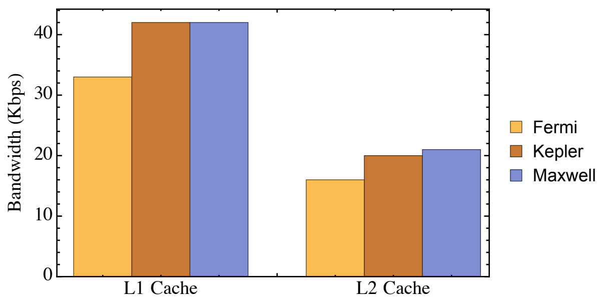
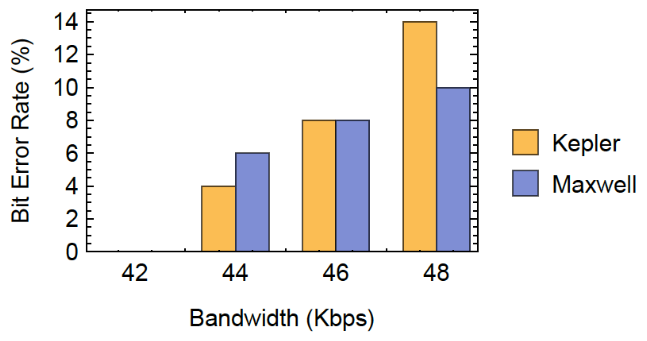


Figure 4.3: Cache channel bandwidth

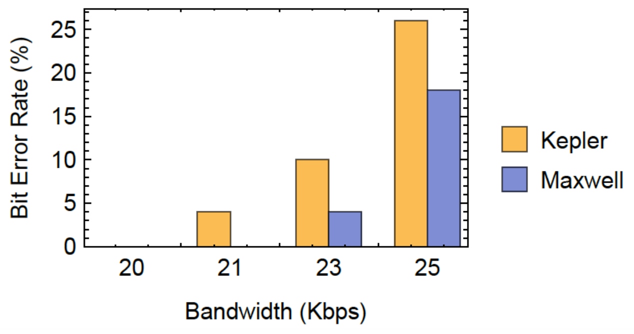
Note that, to ensure overlap between spy and trojan processes and error free communication, we need to iterate sending each bit a sufficient number of times (20 times for L1 channel and 2 times for L2 Channel for Kepler GPU in our experiments). The minimum number of iterations is limited by two factors. First, without synchronization, we must ensure that sufficient iterations are present for the spy and trojan to overlap. Moreover, the clock() function on the

GPGPU returns inconsistent results if the size of the code segment being timed is small. These factors place a limit on the minimum number of iterations necessary to detect contention.

Decreasing the duration below 20 iterations causes errors, since the two kernels sometimes do not overlap. Figure 4.4 demonstrates the bit error rate as the bandwidth of channel increases (by decreasing the number of iterations) for the Kepler and Maxwell GPUs. The L1 and L2 channels on Fermi GPU also show nearly identical behavior around the reported error-free bandwidth.



(a) L1 Channel



(b) L2 Channel

Figure 4.4: Bit error rate of L1 and L2 cache channels

4.4 Functional Unit Channels

Next we explore covert channels that use contention on the functional units (FUs) of the GPGPU. Conceptually, measurable contention can be created on functional units: when two kernels issue instructions to the same functional units, each should observe these instructions to execute slower than if either of them was issuing instructions on its own. While this general intuition holds true, we discover that contention behavior is significantly more complicated. The functional units are pipelined, isolating the contention to contention on the initial dispatch of operations to the functional units. This behavior is also moderated by the warp schedulers: we discover that mostly contention is isolated to warps belonging to the same warp scheduler, which must compete for the issue bandwidth of this scheduler. We first characterize the contention behavior, then show how we can exploit it to construct covert channels.

4.4.1 Step I: Characterizing Contention Behavior on the Functional Units

We set up an experiment to characterize the impact of contention on the performance of the different types of functional units. Each GPGPU has a number of computational cores that are specialized for different instructions; these include single precision units (SP), double precision units (DPU), and special function units (SFU). The number of functional units varies by the architecture and the type of the functional unit. We show only floating point operations because they have the most stringent issue limitations making it easier to create contention, and achieving the highest bandwidth.

For different types of operations, we launch just one kernel which executes a fixed number of operations to the functional unit being characterized. We increase the number of warps and measure the latency of each operation. Figure 4.5 presents latency plots for different single precision floating point operations (`__sinf` and `sqrt` which are executed on SFUs and Add and Mul which are executed on SPs) on different architectures. Figure 4.6 presents latency plots for double precision Add and Mul on Fermi and Kepler GPUs (Maxwell GPU does not have double precision units).

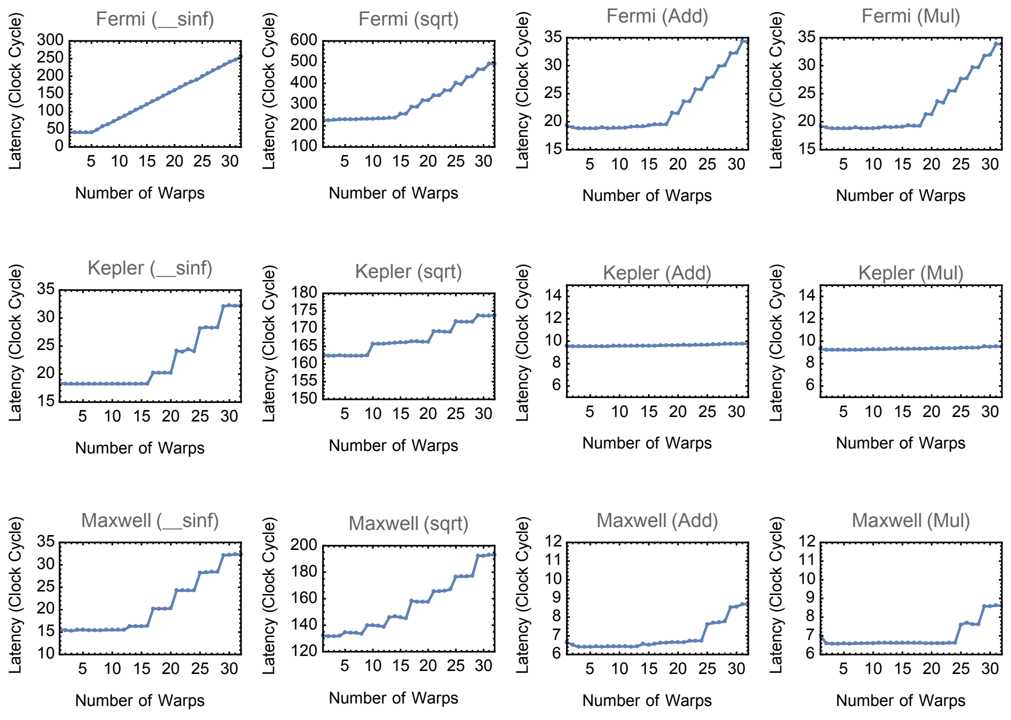


Figure 4.5: Latency of one single precision operation for different number of warps averaged over 128 iterations

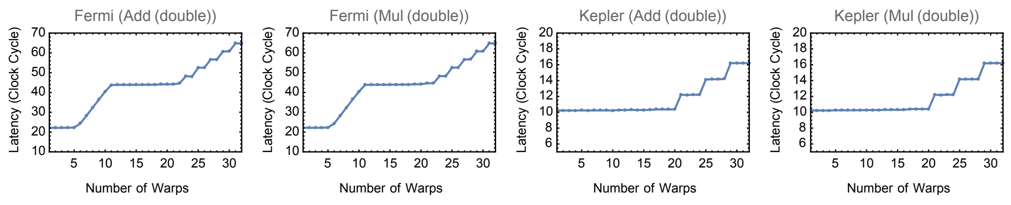


Figure 4.6: Latency of one double precision operation for different number of warps averaged over 128 iterations

The latency plotted in Figure 4.5 and Figure 4.6 is an approximate measure in this case since it is a function of not only contention but also the number of iterations of the experiment and the depth of the functional unit pipeline. The latency places an upper bound on the bandwidth of the channel since even if contention is possible with a single operation, the latency is the minimum delay of a communication cycle. However, the shape of the delay curve is more important since it establishes the degree of contention at which observable changes in measurable delay occur.

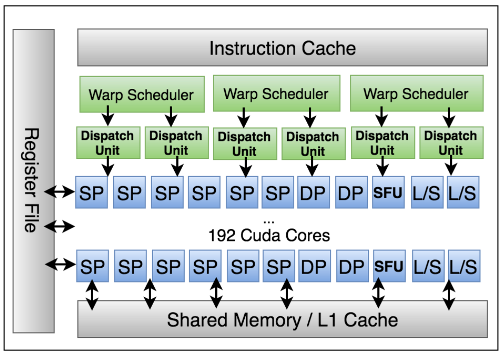


Figure 4.7: Kepler SM architecture overview

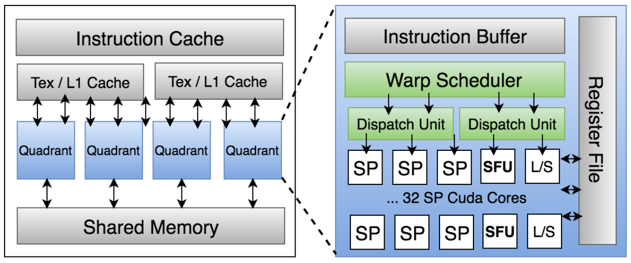


Figure 4.8: Maxwell SM architecture overview

The number of available resources in each SM is shown in Table 4.1 for the three GPUs. In the Maxwell GPU (Figure 4.8), each SM is divided into quadrants, and each quadrant has its own registers, instruction buffer, and scheduler that spans 32 single-precision CUDA cores and 8 SFUs. Each warp scheduler manages the resources for one of the quadrants. In contrast, the Fermi and Kepler GPUs (Figure 4.7) implement soft sharing where the warp schedulers do not have dedicated resources and instead issue instructions to a shared set of resources on the core.

Table 4.1: Number of available resources in each SM.

GPU	Warp Scheduler	Dispatch Unit	SP	DPU	SFU	LD/ST
Tesla C2075 (Fermi)	2	2	32	16	4	16
Tesla K40C (Kepler)	4	8	192	64	32	32
Quadro M4000 (Maxwell)	4	8	128	0	32	32

Main Observations: Due to the different number of warp schedulers, the number of functional units of each type and the depth of the pipeline for each functional unit, we can see different behavior in each plot. The figures show the latency observed by warp 0 which is assigned to the first scheduler on the GPGPU. As we add warps, we see a step in latency plots, at the points where the number of operations causes contention either for scheduler issue bandwidth or for available functional units. For all experiments, the latency stays fixed as we increase the number of warps up to the point where the number of issued instructions matches the number of functional units for that operation. After that point, increasing number of warps leads to an increase in latency for the warps that are co-located on the same warp scheduler with the last added warp. *Contention is isolated to warps belonging to the same warp scheduler.* This result held even for the Kepler GPU which has soft sharing of the resources. For Kepler, the latency steps are not visible for single precision Add and Mul operations, due to the large number of available SP units. In contrast, in Maxwell the latency steps are eventually observable because the resources are split into quadrants (Figure 4.8).

4.4.2 Step II: Constructing FU Covert Channels

The trojan generates operations to the target functional unit to create contention when it desires to communicate 1, while it stays idle when it desires to communicate 0. Because of limited number of SFUs and clear jumps in latency on all three architectures, we elect to use contention on the SFUs. In particular, we take advantage of the more clear steps and lower latency of `__sinf` operation to create and demonstrate a covert channel through the warp schedulers and special functional units. Similar channels can be constructed using other resources.

We launch two concurrent kernels on different streams on GPU. To be sure that thread blocks of the two kernels can be co-located on each SM, we consider a number of blocks for each kernel equal to the number of SMs for different architectures. To be compatible with latency plots in Figure 4.5, we use the minimum number of required warps that will cause observable latency difference using the `__sinf` operation on the Spy side. This requirement translates to having each block of the spy and the trojan use 3 warps, 12 warps and 10 warps, for the Fermi, Kepler and Maxwell architectures respectively. The spy kernel does a number of `__sinf` operations which are executed on the SFUs. To send 0, the trojan does nothing so just 3 warps of the spy (or 12, 10, on the Kepler and Maxwell respectively) are scheduled to issue instructions and execute on SFUs. The latency in this case is about 41 clock cycles for Fermi (18 for Kepler and 15 for Maxwell) which is equal to the latency when there is no contention. For sending 1, the trojan executes a number of `__sinf` operations, so that its warps are scheduled to issue instructions alongside of the spy warps. In this case, there is contention on the SFUs and latency is increased to 48 clock cycles for Fermi (24 for Kepler and 20 for Maxwell). The measured covert channel bandwidth is 21 Kbps, 24 Kbps and 28 Kbps on Fermi, Kepler and Maxwell GPUs respectively.

4.5 Global Memory Channels

In this section, we explore constructing covert channels through global memory, which provides an additional resource for contention when kernels are not co-located on the same SM. We also explore the impact of the access pattern on interference (coalesced vs. uncoalesced addresses). In particular, Jiang et al. [114] demonstrated a side channel attack on GPUs that times an AES encryption kernel running on the GPU from the CPU side. This attack relies on an observation that key-dependent differences in the coalescing behavior of memory accesses, lead to key-dependent encryption times which are used to infer the secret key. Thus, we wanted to explore whether the same phenomena can be exploited to produce high quality covert channels inside the GPU.

Using normal load and store operations, we did not observe reliable contention in the global memory. We believe that this is due to the high memory bandwidth. In particular, to saturate the memory bandwidth, many global memory operations are required, each with high latency, significantly harming achievable bandwidth. To create contention, we focused on atomic operations. Since the atomic operations rely on atomic units that are limited in number, it is possible to cause measurable contention. At the same time, atomic operations are extremely slow, which can limit the bandwidth of the covert communication; nevertheless, this channel achieves comparable bandwidth to other inter-SM channels. We define three scenarios to understand the observable contention in global memory as follows. As before, the spy executes the operation described in each scenario while the trojan executes the same operations to transmit 1, or does nothing to transmit 0. In all three scenarios, the spy and trojan kernels access two different arrays located in global memory.

- Scenario 1: Each thread does atomic additions on one particular global memory address. This address differs for different threads.
- Scenario 2: Each thread does atomic additions on strided global memory addresses. These addresses differ for different threads and accesses for all threads in each warp are **coalesced**.
- Scenario 3: Each thread does atomic additions on consecutive global memory addresses. These addresses differ for different threads and accesses for all threads in each warp are **un-coalesced**.

The bandwidth of the three scenarios for each of the three GPUs is shown in Figure 4.9. For each GPU, we tune the number of iterations to the minimum that will cause observable contention. On the Kepler and Maxwell GPUs the throughput of global memory atomic operation is significantly higher than that of the Fermi since atomic operations are supported through the L2 cache and due to the addition of more atomic units in hardware [30]. Atomic operation throughput to a common global memory access is improved by 9x to one operation per clock cycle causing the overall time to communicate a bit to decreased significantly. It is interesting to note that the coalescing behavior that benefited Jiang et al.'s attack can not be exploited to create the covert channel attack. Coalescing causes timing variability to a single kernel, benefiting an external adversary that times this kernel and has no effect on timing of a competing kernel. However, our experiments show that the coalescing behavior on GPUs improves the channel bandwidth. The poor coalescing significantly reduces the possibility of using the faster L2-level atomic operation support, significantly slowing down the covert communication across the two different kernels. We can see that scenario 3 results in the lowest achievable covert channel bandwidth.

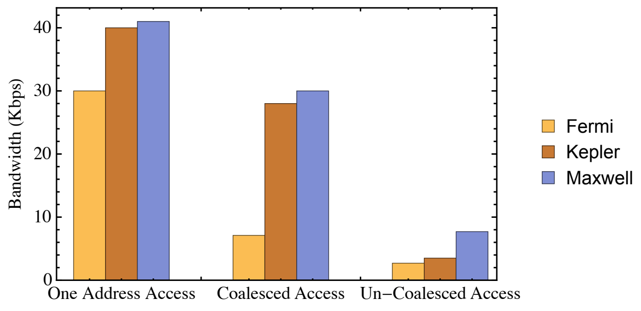


Figure 4.9: Global atomic covert channel bandwidth

4.6 Improving the Attack

In this section, we explore several improvements and optimizations to increase the bandwidth and reliability of the covert channels.

We use two general approaches to increase the bandwidth of a covert channel on a single resource: (1) identifying opportunities for parallelizing the communication so that multiple trojans are communicating to multiple spies concurrently; and (2) implementing synchronization to eliminate the loss of bandwidth that results from timing drift or, alternatively, the overhead of successively launching kernels. It is also possible to use multiple resources simultaneously to increase bandwidth. As an example, we experimented with sending two bits concurrently, one through L1 constant cache and one through the SFUs, achieving 56 Kbps bandwidth for Kepler and Maxwell GPUs. This approach is orthogonal to single-resource channel optimizations, allowing us to increase the bandwidth in those cases as well. We do not report multi-resource bandwidth since it is possible to use multiple resources on the CPU as well.

4.6.1 Improving the Cache Channels

The implementations we discussed so far are susceptible to loss of synchronization where the trojan and spy are not in sync with each other. This can occur due to natural drift due to unpredictable pipeline dependencies, or due to interference from other workloads. To overcome this issue, the covert channels we presented so far, forces overlap between the trojan and the spy by timing the launch of the kernel, leading to significant overhead and loss of bandwidth. Moreover, in the presence of competing applications, co-location may be difficult to achieve repeatedly.

To improve both the robustness and the bandwidth of the attack, in this section, we implement synchronization between the spy and the trojan through the covert channel. With synchronization, the two kernels are launched only once and use synchronization to communicate continuously. We illustrate the synchronization process for the L1 and L2 covert channels, but it is possible to implement synchronization for other channels as well.

The synchronized implementation uses three different sets of cache to fully synchronize sending and receiving bits, as follows. As with the basic cache side channel we use one set for communication. The two other sets are used to signal ready-to-send from the trojan to the spy and ready-to-receive from the spy to the trojan respectively. The synchronized protocol is shown in Figure 4.10 and explained below.

On the Trojan side, for sending each bit:

- First, the trojan sends a ready-to-send signal by filling the pre-agreed on cache set with its data.
- Next it waits on the ready-to-receive cache set to ensure the spy is ready to receive the bit.
- Finally, we send the bit by either filling the set or not. The algorithm moves back to step 1.

Correspondingly, on the Spy side, for receiving each bit:

- First, the spy repeatedly checks the ready-to-send set to check if the trojan has sent the signal.
- Once it detects the ready-to-send signal (by measuring cache misses on the set), it sends the ready-to-receive signal on the corresponding set.
- Finally, we receive the bit through checking the access time observed on the communication set.

Infrequently, due to noise or other factors loss of synchronization can occur leading to deadlock where the spy and trojan are each waiting in a different part of the communication loop. To address this situation, we changed the algorithm to time out (by bounding the number of wait iterations) when the expected signal is not received in time. In the case of a timeout at the sender or the receiver, we regain synchronization by repeating the step prior to the wait. With this modification, the communication works seamlessly. We use a three way handshake to ensure that the trojan and spy are concurrently active at the bit communication component of the program; attempting a two way handshake led to noise and frequent loss of synchronization.

Although communication through three sets (rather than just one in the original channel) is required to fully synchronize the communication, removing the overhead of launching kernels for each bit of message increases the bandwidth to 61, 75 and 75 Kbps for the Fermi, Kepler and Maxwell GPUs, respectively.

Table 4.2: Improved L1 Channels

GPU	L1 Baseline	Sync.	Sync. and multi-bits	Sync., multi-bits and parallel
Fermi	33 Kbps	61 Kbps	207 Kbps	2.8 Mbps
Kepler	42 Kbps	75 Kbps	285 Kbps	4.25 Mbps
Maxwell	42 Kbps	75 Kbps	285 Kbps	3.7 Mbps

To further improve the synchronized channel bandwidth, we utilize SIMT execution model of GPUs to send *M* bits through *M* different cache sets concurrently in each round. Two cache sets are used for signaling and we use the remaining cache sets for communication, such that one bit can be communicated through each cache set by different threads in parallel. For example, in Kepler and Maxwell GPUs L1 constant cache has 8 sets, enabling transfer of 6 bits concurrently. This parallelism increases the bandwidth to 207, 285 and 285 Kbps for the Fermi, Kepler and Maxwell GPUs, respectively. Our experiments demonstrate that by sending 2 bits, 4 bits and 6 bits concurrently, we are able to achieve 1.8x, 2.9x and 3.8x bandwidth improvement in Kepler GPU. Note that the ratio of bandwidth improvement is sublinear in the number of bits; we believe that this is due to both port contention, as well as the higher possibility of a cache miss in each kernel iteration.

The next approach to improve the bandwidth is to exploit the inter-SM parallelism available on the GPGPU. In particular, if we manage to colocate the trojan and the spy on multiple SMs, each of these instances can communicate independently using resources on the SM. With 15 SMs available in Tesla K40C device, the trojan is able to send 15 bits simultaneously and the bandwidth of communication is increased 15 times. Table 4.2 shows the covert channel communication bandwidth for the baseline attack (column 1), the attack with synchronization (column 2), the attack with synchronization and sending multiple bits through different cache sets (column 3) and the attack with all three improvements (column 4). Clearly, high quality, high bandwidth covert channels are feasible on GPGPUs.

For the L2 constant cache covert channel, we can create contention in parallel, either by filling the cache in parallel or by using each thread block to communicate through one particular L2 cache set. In theory, this should enable the trojan to send 16 bits (number of L2 cache sets) simultaneously. However, we observe only an 8x improvement in the best case, which we conjecture is due to cache port contention and cache bank collisions.

Table 4.3: Improved SFU covert channel communication bandwidth on different GPU architectures.

GPU	Baseline	Parallel through warp schedulers	Parallel through warp schedulers and SMs
Tesla C2075 (Fermi)	21 Kbps	28 Kbps	380 Kbps
Tesla K40C (Kepler)	24 Kbps	84 Kbps	1.2 Mbps
Quadro M4000 (Maxwell)	28 Kbps	100 Kbps	1.3 Mbps

4.6.2 Improving the SFU Channel

Like the L1 covert channel, there is an opportunity to improve the bandwidth by having a spy and a trojan communicate on each SM for a 15x increase in bandwidth. However, the warp scheduler offers additional opportunities to increase the bandwidth. In Section 4.4, we also reverse engineered the warp assignment algorithm and found it to be round robin. By measuring latency for each warp, we noticed that by increasing number of warps one by one we see latency increasing *only in the warps that are assigned to the same warp scheduler*. This effect is likely due to the limit on the number of instructions that can be dispatched by each warp scheduler. Certainly, this behavior is more clear in Maxwell GPUs in which each warp scheduler has its own functional units, but it is also present in the Kepler and the Fermi architectures where the functional units are soft-shared among the schedulers.

We use the observation that contention is isolated among the different warp schedulers to parallelize the covert channel attack by sending one bit through each warp scheduler concurrently. We consider *K* warps for spy which *K* is multiple of *N* (the number of warp schedulers) such that the latency of *K* warps alone is in the area of constant delay in the *__sinf* plots of Figure 4.5 for each architecture. For the trojan, we consider *M* warps such that *M* is also multiple of *N* and the latency of *M+K* is on one of the steps in the plot. Each warp of the trojan and the spy is assigned to one of the *N* different warp schedulers. We select one warp from each scheduler from the trojan to send one bit and another corresponding warp on the spy side to receive the bit. We are successfully able to parallelize the communication in this way, leading to a bandwidth that is *N* times more than the baseline channel. We are also able to carry out this communication independently on each of the *S* SMs, leading to increasing the bandwidth by another factor of *S*. The Covert channel bandwidth for the baseline channel is compared to the two parallelization steps (across SMs, and across warp schedulers in each SM) in Table 4.3. Note that the Fermi GPU has two warp schedulers per SM while Kepler and Maxwell have four warp schedulers per SM.

4.7 Mitigating Noise

We consider the presence of interference from other workloads which can affect the covert channel in two ways: (1) Co-location: the other workloads may prevent the spy and trojan from being launched together, or may cause them to be assigned to different SMs preventing the high quality covert channels that are present inside an SM; and (2) Noise: even if the spy and trojan are co-located, a third application may use the resources used for covert communication adding noise or even completely disrupting communication.

$D_{Send}[N], D_{Recv}[N]$: N is number of bits to transmit and receive
ReadyToSend(): sends ReadyToSendSig to Spy; ReadyToReceive(): sends ReadyToReceiveSig to Trojan
wait(S): while loop that breaks on signal S
prime(): fills the communication cache line; probe(): access the communication cache line and return 0 on a hit and 1 on a miss

Trojan protocol:	**Spy protocol:**	Description:
for $i \leftarrow 0$ **to** $N-1$ **do**	**for** $i \leftarrow 0$ **to** $N-1$ **do**	
ReadyToSend()	wait(ReadyToSendSig)	Handshake stage
wait(ReadyToReceiveSig)	ReadyToReceive()	
if $D_{Send}[i] = 1$ **then** prime()	$D_{Recv}[i] \leftarrow$ probe()	Transmitting and receiving stage

Figure 4.10: Synchronization communication protocol

The high parallelism available on GPGPUs can result in high degrees of noise that challenge error detection and correction. Thus, our primary approach to manage noise is to try to prevent it completely by manipulating the block scheduler to achieve exclusive co-location between the spy and trojan at the level of the SM or the full GPGPU. We take advantage of concurrency limitations on current GPUs which use leftover policy for multiprogramming. In particular, we have the trojan and spy ask for resources in a way that they can be co-located with each other but that also makes it difficult for other applications to co-locate with them. For example, the spy may ask for the maximum available amount of the shared memory (or registers, thread blocks, threads, etc..) while the trojan asks for only the leftover amount of that resource. If there are not enough registers or shared memory available per multiprocessor to process at least one block, a competing kernel will be blocked until these resources become available. In this way, once the spy and trojan are launched, they can execute on the GPU (or SM) exclusively and communicate without interference.

In our Fermi (Tesla C2075) and Kepler (Tesla K40C) GPUs, the maximum shared memory per thread block is equal to maximum shared memory per SM (48KB). Since each of the spy and trojan needs just one thread block for communication, if the spy block asks for the maximum shared memory, it can saturate the SM in term of shared memory. In this case, the trojan block can be co-located on the same SM, if it does not use any shared memory. Any other application that uses shared memory, cannot be executed until the spy finishes and there are enough resources for thread blocks of the third application to be scheduled.

To improve the chances for exclusive co-location, we can also launch additional kernels that do not generate noise to exhaust other resources but do not use the resources that are claimed by the trojan and spy. If such kernels are launched at the same time as the kernel and spy, the scheduler will prefer to run them with the trojan and spy since it prioritizes kernels based on their launch time.

Note that, on our Maxwell GPU architecture (Quadro M4000), the maximum shared memory per SM is twice the maximum shared memory per thread block. So if the thread blocks of both trojan and spy ask for the whole shared memory we can exclusively run on each SM and noise is prevented.

To evaluate our exclusive co-location strategies, we executed the Rodinia benchmark applications [64] on a third stream, alongside the spy and trojan communication using L1 cache channel. By forcing exclusive co-location of the spy and trojan through saturating shared memory on each SM, we were able to prevent interference against all interfering workloads and workload mixtures and achieved error free communication in all cases. These workloads include applications which use shared memory and those that do not. They also include workloads such as *Heart Wall* that uses constant memory and that would interfere with the L1 covert channel if it were co-located with the malicious kernels. We note that if the interfering workload is launched before the spy and trojan, the built in synchronization in the kernels allows one of them to wait for the other. When the second kernel is launched, the resource request pattern ensures exclusive co-location after that.

As we discussed in Section 4.2, Wang et al.'s [218] intra-SM resource partitioning mechanism simplifies co-location of two malicious kernels on the same SM. However, it also allows other workloads to execute on the same SM, necessitating approaches for noise avoidance or tolerance. In contrast, the approach by Xu et al. [230] does not support preemption and thus allows exclusive co-location similar to current GPUs. When exclusive co-location to prevent noise is not possible, noise could be avoided or tolerated using techniques such as the following.

- Dynamically identifying idle resources: The approach is similar to whitespace communication in wireless networks where the radios opportunistically discover and use available channels without prior agreement [54], and solutions from that space can be leveraged for our problem. For example, the sender may scan through available resources (e.g. cache sets) in a pre-agreed on order until it discovers idle ones and transmits a beacon pattern on them. The receiver follows by scanning sets until it observes the beacon.
- Error correction: transmit error correcting codes with the data (sacrificing some of the bandwidth).

Since we were able to establish exclusive co-location on our GPUs and achieved noise free communication, we did not pursue either of those directions.

4.8 Possible Mitigations

In this section, we provide a brief discussion of possible mitigations on GPU covert channel. One approach is to use partitioning to ensure possibly communicating applications do not have a way to effect measurable contention to each other. Partitioning can be done spatially (e.g., partitioning the cache [216, 76, 141], or temporally (e.g., ensuring instructions from different kernels do not execute in the same time period).

Specifically on GPUs, partitioning can be achieved at intra-SM and inter-SM level resources through scheduling of different application thread blocks or warps to separate them temporally or spatially. In addition, it is possible to fairly partition shared hardware resources among multiple simultaneous kernels based on their workload and resource requirements and make these partitions private to each application to eliminate interference. Although these approaches may lead to performance degradation and add some hardware overhead, they can prevent covert and side channel attacks which are results of unrestricted access to shared hardware resource from two or more co-located applications.

Another mitigation approach is to attempt to detect anomalous contention [65]. Given the different nature of GPGPU workloads and the degree of contention that is likely to arise naturally, a detailed evaluation of this class of solutions is necessary to assess its effectiveness. Solutions are possible that add entropy either to the assignment of the resources [217] or to the measurement of time [148]. Finally, scheduling algorithms that interfere with co-location, accommodate preemption, or prevent exclusive co-location (to introduce noise), can significantly complicate the attack.

In Chapter 6, we demonstrate how we combine machine learning based detection and fine grained partitioning to provide the efficient GPU-specific mitigation against contention based channels.

Chapter 5

GPU Side Channel

In chapter 4, we demonstrate that GPUs provide very high quality and bandwidth covert channels. In this chapter, we explore whether side channel attacks on GPUs are practical. GPUs often process sensitive data both with respect to graphics workloads (which render the screen and can expose user information and activity) and computational workloads (which may include applications with sensitive data or algorithms). Such attacks represent a novel and dangerous threat vector. There are a number of unique aspects of side channel attacks on the GPU due to the different computational model, high degree of parallelism, unique co-location and sharing properties, as well as attacker-measurable channels present in the GPU stack. We show that indeed side channels are present and exploitable, and demonstrate attacks on a range of Nvidia GPUs and for both graphics and computational software stacks and applications.

5.1 Attack Space

In this section, we first define three attack models based on the placement of the spy and the victim. We then describe the available leakage vectors in each model.

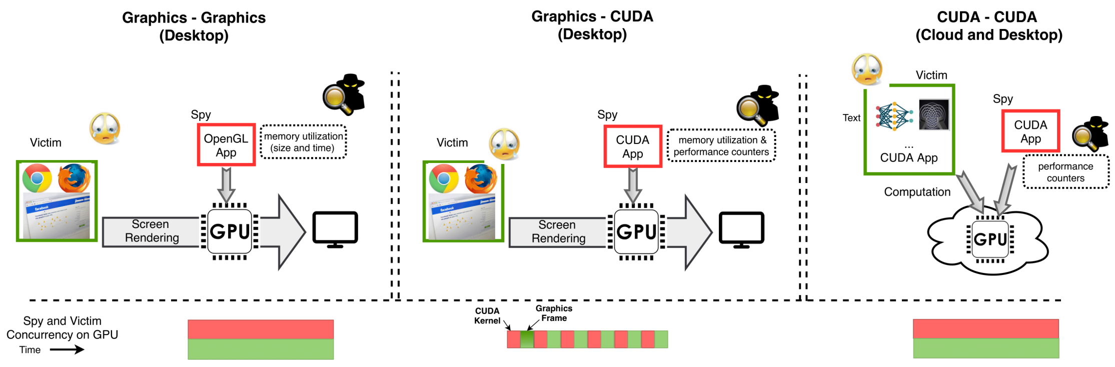


Figure 5.1: Attack Models

5.1.1 Attack Models

We consider three primary attack vectors. In all three cases, a malicious program with normal user level permissions whose goal is to spy on a victim program.

- Graphics spy on a Graphics victim: attacks from a graphics spy on a graphics workload (Figure 5.1, left). Since Desktop or laptop machines by default come with the graphics libraries and drivers installed, the attack can be implemented easily using graphics APIs such as OpenGL measuring leakage of a co-located graphics application such as a web browser to infer sensitive information.
- CUDA spy and graphics victim (Cross-Stack): on user systems where CUDA libraries and drivers are installed, attacks from CUDA to graphics applications are possible (Figure 5.1, middle).
- CUDA spy on a CUDA victim: attacks from a CUDA spy on a CUDA workload typically on the cloud (Figure 5.1, right) where CUDA libraries and drivers are installed.

In the first attack model, we assume that the attacker exploits the graphics stack using APIs such as OpenGL or WebGL. In attack models 2 and 3, we assume that a GPU is accessible to the attacker using CUDA or OpenCL.

5.1.2 Available Leakage Vectors on GPUs

In Chapter 4, we showed that two concurrently executing GPU kernels can construct covert channels using CUDA by creating and measuring contention on a number of resources including caches, functional units, and atomic memory units [161, 160]. However, such fine-grained leakage is more difficult to exploit for a side channel attacks: the large number of threads, and the relatively small size of caches, makes it difficult to conduct high-precision prime-probe or similar attacks on data caches [120].

Thus, instead of targeting fine-grained contention behavior, we focus on aggregate measures of contention through available resource tracking APIs to develop side channels.

There are a number of mechanisms available to the attacker to measure the victim's performance. These include: (1) the memory allocation API, which exposes the amount of available physical memory on the GPU; (2) the GPU hardware performance counters; and (3) Timing operations while executing concurrently with the victim. We verified that the memory channel is available on Nvidia GPUs [33] on any Operating System supporting OpenGL (including Linux, Windows, and MacOS). Nvidia GPUs currently support performance counters on Linux, Windows and MacOS for computing applications [29] and on Linux and Android [27, 36] for graphics applications. WebGL does not appear to offer extensions to measure any of the three channels and therefore cannot be used to implement a spy for our attacks. Although web browsers and websites which use WebGL (as a JavaScript API to use GPU for rendering) can be targeted as victims in our attacks from an OpenGL spy.

A. Measuring GPU Memory Allocation: When the GPU is used for rendering, a content-related pattern (depending on the size and shape of the object) of memory allocations is performed on the GPU. We can probe the available physical GPU memory using an Nvidia provided API through either a CUDA or an OpenGL context. Repeatedly querying this API we can track the times when the available memory space changes and even the amount of memory allocated or deallocated.

On an OpenGL application we can use the ”NVX_gpu_memory_info ” extension [33] to do the attack from a graphics spy. This extension provides information about hardware memory utilization on the GPU. We can query ”GPU_MEMORY_INFO_CURRENT_AVAILABLE_VIDMEM_NVX ” as the value parameter to glGetIntegerv. Similarly, on a CUDA application, the provided memory API by Nvidia is ”cudaMemGetInfo”.

B. Measuring Performance Counters: We use Nvidia profiling tools [29] to monitor the GPU performance counters from a CUDA spy. Table 5.1 summarizes some important events/metrics tracked by the GPU categorized into five general groups. Although the GPU allows an application to only observe the counters related to its own computational kernel, these are affected by the execution of a victim kernel: for example, if the victim kernel accesses the cache, it may replace the spy's data allowing the spy to observe a cache miss (through cache-related counters). We note that OpenGL also offers an interface to query the performance counters enabling them to be sampled by a graphics-based spy.

C. Measuring Timing: It is also possible to measure the time of individual operation in attack models where the spy and the victim are concurrently running to detect contention.

Leakage Vectors on Integrated GPUs: Although this chapter focuses on discrete GPU side channels, we verified that similar leakage and attacks are possible on integrated GPUs as well. In integrated GPU systems, there is no memory API to track GPU memory utilization, since memory is shared between CPU and GPU. Although userspace interfaces to query performance counters, available in almost all integrated and discrete GPUs, making our attacks effective on integrated GPUs such as Intel Graphics and Qualcomm Adreno as well. Specifically, Intel provides an OpenGL extension ”Intel_performance_query”[32] to access the GPU performance counters organized in some query types including ”Intel_GT_Hardware_Counters”. This query type includes counters like stall time and read and write memory throughput that can be affected by other co-running applications on GPU, providing side channel leakage.

Experimental Setup: We verified the existence of all the reported vulnerabilities in this chapter on three Nvidia GPUs from three different microarchitecture generations: a Tesla K40 (Kepler), a Geforce GTX 745 (Maxwell) and a Titan V (Volta) Nvidia GPUs. We report the result only on the Geforce GTX 745 GPU. The experiments were conducted on an Ubuntu 16.04 distribution, but we verified that the attack mechanisms are accessible on both Windows and MacOS systems as well. The graphics driver version is 384.11 and the Chrome browser version is 63.0.3239.84.

Table 5.1: GPU performance counters

Category	**Event/Metric**
Memory	Device memory read/write throughput Global/local/shared memory LD/ST throughput L2 RD/WR transactions Device memory utilization
Instruction	Control flow, INT, FP (single/double) instructions Instruction executed/issued, Issued/executed IPC Issued load/store instructions Issue stall reasons (data request, execution dependency,texture,...)
Multiprocessor	SP/DP function unit(FU) utilization Special FU utilization Texture FU utilization, Control-flow FU utilization
Cache	L2 hit rate (texture read/write) L2 throughput/transaction
Texture	Unified cache hit rate/throughput/utilization

5.2 Attack Model I : Graphics Spy on a Graphics Victim

We consider the first threat model where an application uses a graphics API such as OpenGL to spy on another application that uses the GPU graphics pipeline (Figure 5.1, left).

Reverse Engineering Co-location: To understand how two concurrent applications share the GPU, we carry out a number of experiments to see if the two workloads can run concurrently and to track how they co-locate. The general approach is to issue the concurrent workloads and measure both the time they execute using the GPU timer register, and SM-ID they execute on (which is also available through the OpenGL API). If the times overlap, then the two applications colocate at the same time. If both the time and the SM-IDs overlap, then the applications can share at the individual SM level, which provides additional contention spaces on the private resources for each SM.

We launch two long running graphics applications rendering an object on the screen repeatedly. OpenGL developers (Khronos group) provide two extensions: "NV_shader_thread_group" [34] which enable programmers to query the ThreadID, the WarpID and the SM-ID in OpenGL shader codes and "ARB_shader_clock" [22] which exposes local timing information within a single shader invocation. We used these two extensions during the reverse engineering phase in the fragment shader code to obtain this information. Since OpenGL does not provide facilities to directly query execution state, we encode this information in the colors (R, G, B values) of the output pixels of the shader program (since the color of pixels is the only output of shader program). On the application side, we read the color of each pixel from the framebuffer using the `glReadPixels()` method and decode the colors to obtain the encoded ThreadID, SM-ID and timing information of each pixel (representing a thread).

We observed that two graphics applications whose workloads do not exceed the GPU hardware resources can colocate concurrently. Only if a single kernel can exhaust the resources of an entire GPU (extremely unlikely), the second kernel would have to wait. Typically, a GPU thread is allocated to each pixel, and therefore, the amount of resources reserved by each graphics kernel depends on the size of the object being processed by the GPU. We observe that a spy can co-locate with a rendering application even it renders the full screen (Resolution 1920x1080) on our system. Because the spy does not ask for many resources (number of threads, shared memory, etc...), we also discover that it is able to share an SM with the other application. In the next two subsections, we explain implementation of two end to end attacks on the graphics stack of GPU.

5.2.1 Attack I: Website Fingerprinting

The first attack implements website fingerprinting as a victim surfs the Internet using a browser. We first present some background about how web browsers render websites to understand which part of the computation is exposed to our side channel attacks and then describe the attack and evaluation.

Web Browser Display Processing: Current versions of web browsers utilize the GPU to accelerate the rendering process. Chrome, Firefox, and Internet Explorer all have hardware acceleration turned on by default. GPUs are highly-efficient for graphics workload, freeing up the CPU for other tasks, and lowering the overall energy consumption.

As an example, Chrome's rendering processing path consists of three interacting processes: the renderer process, the GPU process and User Interface (UI) process. By default, Chrome does not use the GPU to rasterize the web content (recall that rasterization is the conversion from a geometric description of the image, to the pixel description). In particular, the webpage content is rendered by default in the renderer process on the CPU. Chrome uses shared memory with the GPU process to facilitate fast exchange of data. The GPU process reads the CPU-rasterized images of the web content and uploads it to the GPU memory. The GPU process next issues OpenGL draw calls to draw several equal-sized quads, which are each a rectangle containing the final bitmap image for the tile. Finally, Chrome's compositor composites all the images together with the browser's UI using the GPU.

We note that WebGL enables websites and browsers to use GPU for whole rendering pipeline, making our attacks effective for all websites that use WebGL [38]. For websites that do not use WebGL, Chrome does not use the GPU for rasterization by default, but there is an option that users can set in the browser to enable GPU rasterization.[1] If hardware rasterization is

[1]GPU rasterization can be enabled in `chrome://flags` for Chrome and in `about:config` through setting the `layers.acceleration.force-enabled` option in Firefox.

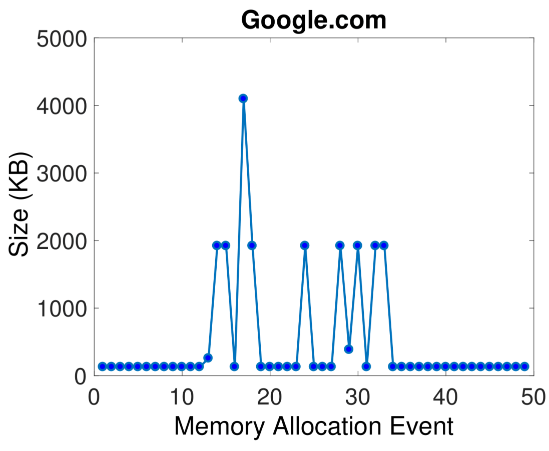
enabled, all polygons are rendered using OpenGL primitives (triangles and lines) on the GPU. GPU accelerated drawing and rasterization can offer substantially better performance, especially to render web pages that require frequently updated portions of screen. As a result, the Chromium Project's GPU Architecture Roadmap [25] seeks to enable GPU accelerated rasterization by default in Chrome in the near future. For our attacks we assume that hardware rasterization is enabled but we also report the experimental results without enabling GPU rasterization.

Launching the Attack: In this attack, a spy has to be active while the GPU is being used as a user is browsing the Internet. In the most likely attack scenario, a user application uses OpenGL from a malicious user level App on a desktop, to create a spy to infer the behavior of a browser process as it uses the GPU. However, a CUDA (or OpenCL) spy is also possible assuming the corresponding driver and software environment is installed on the system, enabling Graphics-CUDA side channel attack described in Section 5.3.

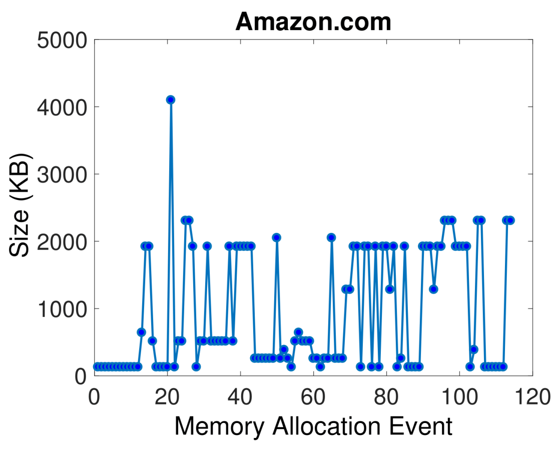
Probing GPU Memory Allocation: The spy probes the memory API to obtain a trace of the memory allocation operations carried out by the victim as it renders different objects on a webpage visited by the user.

We observe that every website has a unique trace in terms of GPU memory utilization due to the different number of objects and different sizes of objects being rendered. This signal is consistent across loading the same website several times and is unaffected by caching. To illustrate the side channel signal, Figure 5.2 shows the GPU memory allocation trace when Google and Amazon websites are being rendered. The x-axis shows the allocation events on the GPU and the y-axis shows the size of each allocation.

We evaluate the memory API attack on the front pages of top 200 websites ranked by Alexa [21]. We collect data by running a spy as a background process, automatically browsing each website 10 times and recording the GPU memory utilization trace for each run.



(a)



(b)

Figure 5.2: Website memory allocation on GPU (a) Google; (b) Amazon

Classification We leverage machine learning algorithms to classify the traces to infer which websites the victim is likely to have visited. We first experimented with using time-series classification through dynamic time warping, but the training and classification complexity was high. Instead, we construct features from the full time series signal and use traditional machine learning classification, which also achieved better accuracy. In particular, we compute several statistical features, including minimum, maximum, mean, standard deviation, slope, skew and kurtosis, for the series of memory allocations collected through the side channel when a website is loading. We selected these features because they are easy to compute and capture the essence of the distribution of the time series values. The skew and kurtosis capture the shape of the distribution of the time series. Skew characterizes the degree of asymmetry of values, while the Kurtosis measures the relative peakness or flatness of the distribution relative to a normal distribution [163]. We computed these features separately for the first and the second half of the time series recorded for each website. We further divided the data in each half into 3 equal segments, and measured the slope and the average of each segment. We also added the number of memory allocations for each website, referred as "*memallocated*", into the feature vector representing a website. This process resulted in the feature set consisting of 37 features.

We then used these features to build the classification models based on three standard machine learning algorithms, namely, K Nearest Neighbor with 3 neighbors (KNN-3), Gaussian Naive Bayes (NB), and Random Forest with 100 estimators (RF). We evaluate the performance of these models to identify the best performing classifier for our dataset. For this and all classification experiments we validated the classification models using standard 10-fold cross-validation method (which separates the training and testing data in every instance).

As performance measures of these classifiers, we computed the precision (*Prec*), recall (*Rec*), and F-measure (FM) for machine learning classification models. *Prec* refers to the accuracy of the system in rejecting the negative classes while the *Rec* is the accuracy of the system in accepting positive classes. Low recall leads to high rejection of positive instances (false negatives) while low precision leads to high acceptance of negative instances (false positives). FM represents a balance between precision and recall.

Table 5.2: Memory API based website fingerprinting performance (200 Alexa top websites): F-measure (%), Precision (%), and Recall (%)

	FM	**Prec**	**Rec**
	μ (σ)	μ (σ)	μ (σ)
NB	83.1 (13.5)	86.7(20.0)	81.4 (13.5)
KNN3	84.6 (14.6)	85.7 (15.7)	84.6(14.6)
RF	89.9 (11.1)	90.4 (11.4)	90.0 (12.5)

Table 5.2 shows the classification results. The random forest classifier achieves around 90% accuracy for the front pages of Alexa 200 top websites. Note that if we launch our memory API attack on browsers with default configuration (we do not enable GPU rasterization on browser), we still obtain a precision of 59%.

User Activity Tracking We follow up on the website fingerprinting attack with a side channel to track the user activity on the web. We define user activity as the navigation to sub-level webpages after a user accesses the main page of a website. For example, a user may sign in or sign up from the main page, or a user may browse through the savings or checking account after logging into the banking webpage. Figure 5.3 shows two examples of user activity signatures.

We test this attack on two websites: facebook.com and bankofamerica.com. In the facebook website, our goal is to detect whether the user is signing in or signing up (creating account). In the bankofamerica website, besides detecting signing in/opening an account, we track several user activities to detect which type of account the user intends to open, and other interactions with the website.

The intuition is that depending on the what buttons/links are clicked on the homepage, different subsequent pages (signing in or signing up) will be reached, creating a distinguishable time series signal. Using the same features as the fingerprinting attack, we show the classification performance for these two websites in Table 5.3. The Random Forest classifier could identify the users' web activities accurately with the precision of 94.8%. The higher accuracy is to be expected since the number of possible activities is small.

Table 5.3: Memory API based user activity detection performance: F-measure (%), Precision (%), and Recall (%)

	FM	**Prec**	**Rec**
	μ (σ)	μ (σ)	μ (σ)
NB	93.5(7.9)	93.9 (9.9)	93.3 (7.0)
KNN3	90.8 (12.6)	92.6 (12.1)	91.1 (16.9)
RF	94.4 (8.6)	94.8 (8.8)	94.4 (10.1)

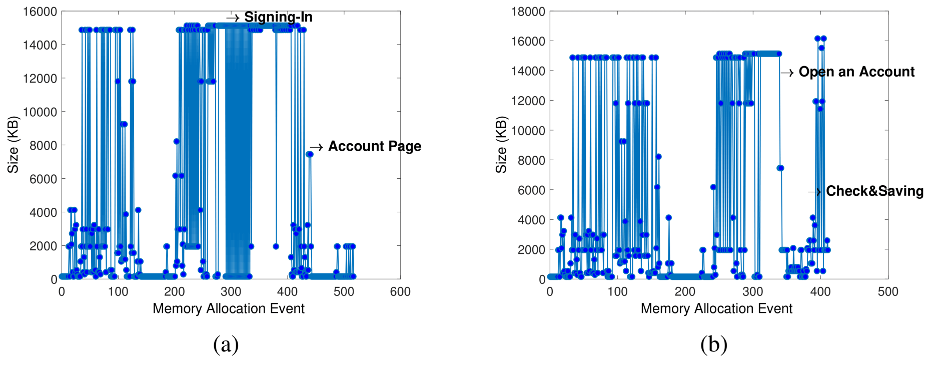


Figure 5.3: User activity memory trace on Bank of America website (a) sign-in ; (b) Open account: Checking & Saving

5.2.2 Attack II: Password Textbox Identification and Keystroke Monitoring

After detecting the victim's visited website and a specific page on the website, we can extract additional finer-grained information on the user activity. By probing the GPU memory allocation repeatedly, we can detect the pattern of user typing (which typically causes re-rending of the textbox). More specifically, from the same signal, it contains (1) the size of memory allocation by the victim, which we use to identify whether it is a username/password textbox (e.g., versus a bigger search textbox); (2) the inter-keystroke time which allows us to extract the number of characters typed and even infer the characters using timing analysis.

As an example, we describe the process to infer whether a user is logging in by typing on the password textbox on facebook, as well as to extract the inter-keystroke time of the password input. Since the GPU is not used to render text in the current default options, each time the user types a character, the character itself is rendered by the CPU but the whole password textbox is uploaded to GPU as a texture to be rasterized and composited. In this case, the monitored available memory will decrease with a step of 1024KB (the amount of GPU memory needed to render the password textbox on facebook), leaking the fact that a user is attempting to sign in instead of signing up (where the sign-up textboxes are bigger and require more GPU memory to render). Next, by monitoring the exact time of available memory changes, we infer inter-keystroke time. The observation is that while the sign-in box is active on the website, waiting for user to input username and password, the box is re-rendered at a refresh rate of around 600 ms. However, if a new character is typed, the box is immediately re-rendered (resulting in a smaller interval). This effect is shown in Figure 5.4, where the X-axis shows the observed *nth* memory allocation events while the Y-axis shows the time interval between the current allocation event and the previous one (most of which are 600ms when a user is not typing). We can clearly see six dips in the figure corresponding to the 6 user keystrokes, and the time corresponding to these dips can be used to calculate inter-keystroke time.

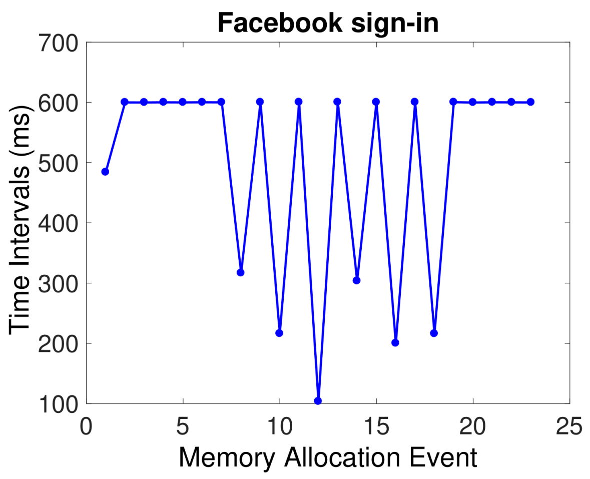


Figure 5.4: Timing memory allocations: 6-character password

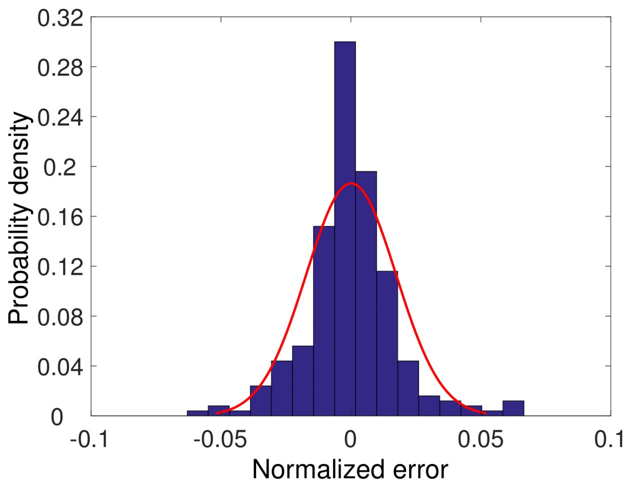


Figure 5.5: Error distribution of inter-keystroke time

Prior work has shown that inter-arrival times of keystrokes can leak information about the actually characters being typed by the user [192]. To demonstrate that our attack can measure time with sufficient precision to allow such timing analysis, we compare the measured inter-keystroke time to the ground truth by instrumenting the browser code to capture the true time of the key presses. We compute the normalized error as the difference between the GPU measured interval and the ground truth measured on the CPU side. Figure 5.5 shows the probability density of the normalized measurement error in an inter-keystroke timing measurement with 250 key presses/timing samples. We observe that the timing is extremely accurate, with mean of the observed error at less than 0.1% of the measurement period, with a standard deviation of 3.1% (the standard deviation translates to about 6ms of absolute error on average, with over 70% of the measurements having an error of less than 4ms). Figure 5.6 shows the inter-keystroke timing for 25 pairs of characters being typed on the facebook password bar (the character a followed by each of b to z), measured through the side channel as well as the ground truth. The side channel measurements (each of which represents the average of 5 experiments) track the ground truth accurately.

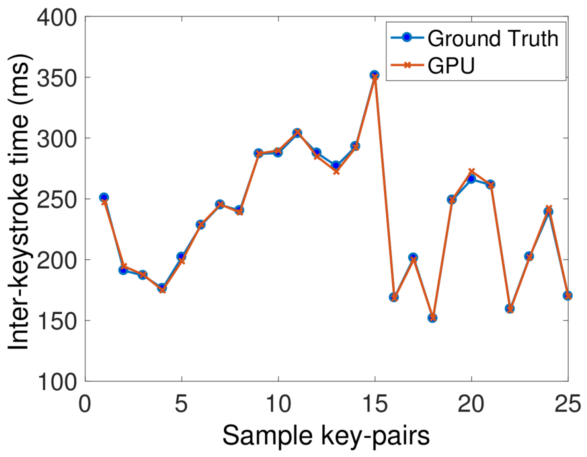


Figure 5.6: Keystroke timing: Ground Truth vs. GPU

5.2.3 Robustness to window sizes

In attacks on the graphics stack, the size of the window being rendered affects the side channel signal leaked to the attacker. We checked the robustness of the classification on the website fingerprinting attack described earlier, which was evaluated under the assumption that the browser used the full screen. However, users may browse websites in the browser of different screen sizes.

Hence, to make our website fingerprinting attack robust, we have to generalize the attack across various window sizes.

Robustness analysis: We discover that changing the window size results in a similar signal with different amplitude for a few websites, and for the responsive websites that have dynamic content or do not scale with window size, there is some variance in the memory allocation signal (e.g., some objects missing due to a smaller window). We collected data for seven different window sizes, including some standard sizes of iPhone, iPad, Laptop and Desktop. These window sizes are 320*568, 600*800, 800*600, 1024*768, 1440*900, 1680*1050 and full screen (1920*1080). To measure the robustness of our classification model on different window sizes, we , first, trained the model on full-screen window dataset and tested on other window sizes. We observed a decrease in the precision of a Random Forest classifier to less than 10% for top 100 Alexa websites. Thus, the attack described thusfar is not robust to change in the window size.

To make our attack robust to a size of a window, we introduce a new attack that estimates the size of the browser window. The intuition is that the intensity of the signal increases with the size of the window as more objects that are larger are drawn. After detecting the window size, the correct classifier (trained at that window size) is used for website fingerprinting. We describe this attack in the remainder of this section.

Detecting window sizes: We computed several features, including minimum, maximum, slope, variation, kurtosis, and skew from the memory allocations associated with the Alexa top 100 websites loaded in the given browser window size. We then trained a model with Random Forest classifier on these features to detect the size of the browser window. We evaluated the performance of our model using 10-fold cross-validation. The performance metrics, *viz.*, f-measure, precision, and recall, of the model, are listed in Table 5.4. To identify the prominent features in the memory allocations representing the window sizes, we computed the information gain on each feature. We observe the maximum, the variation, and the skew of the memory allocations as the top-three features representing different window sizes.

Table 5.4: Window size prediction performance: F-measure (%), Precision (%), and Recall (%)

	FM	Prec	Rec
	μ	μ	μ
320_568	94	96	95
600_800	95	96	96
800_600	93	94	93
1024_768	95	97	96
1440_900	96	98	97
1680_1050	97	92	95
FullScreen	99	97	98

Website Fingerprinting: Similar to the implementation of website fingerprinting on the full-screen browser, we trained a Random Forest classifier to model websites browsed in a specific window size. We evaluate performance with 10-fold cross validation. The classification results are presented in Table 5.5. With this improvement, we believe that the attacks become robust to changes in window size.

Table 5.5: Website fingerprinting performance on different window sizes (100 Alexa top websites): F-measure (%), Precision (%), and Recall (%)

	FM	Prec	Rec
	μ (σ)	μ (σ)	μ (σ)
320_568	93 (0.07)	93 (0.07)	93 (0.06)
600_800	92 (0.09)	92 (0.09)	91 (0.08)
800_600	93 (0.07)	92 (0.08)	92 (0.05)
1024_768	95 (0.06)	94 (0.07)	94 (0.05)
1440_900	94 (0.07)	94 (0.09)	94 (0.07)
1680_1050	94 (0.06)	94 (0.08)	94 (0.05)
Full Screen	94 (0.07)	94 (0.09)	93 (0.06)

5.3 Attack Model II: CUDA Spy on a Graphics Victim

In this section, we demonstrate the attack model II, where a spy from the computational stack attacks a victim carrying out graphics operations. This attack is possible on a desktop or mobile device that has CUDA or openCL installed, and requires only user privileges.

5.3.1 Reverse Engineering the Colocation

We conduct a number of experiments to reverse engineer the GPU schedulers when there are both graphics and computing applications. In the first experiment, we launch a CUDA process and an OpenGL process concurrently on the GPU. On the CUDA side, we write an application to launch a very long CUDA kernel doing some texture memory load operations in a loop and size the application such that there is at least one thread block executing on each SM. We measure the start time and stop time of the CUDA kernel, the start and stop time of each iteration of operations for each thread, and report the SM-ID on which the thread is executing.

On the OpenGL side, we launch a very long application and probe the execution time and the SM-ID at each pixel (thread) as described in Section 5.2. From the experiment above, we observed that when the CUDA kernel starts execution, the graphics rendering application is frozen until the CUDA kernel is terminated. So there is no true concurrency between CUDA and OpenGL applications on the GPU SMs. This behavior is different than multiple CUDA applications (or multiple OpenGL applications) which we found to concurrently share the GPU when the resources are sufficient.

In the next experiment, we launch many short CUDA kernels from one application and keep the long running graphics rendering application. We use the previous methodology to extract the ThreadID, WarpID and timing information on both sides. We observe interleaving execution (not concurrent execution) of CUDA kernels and graphics operations on the GPU. For short CUDA kernels, we achieve fine-grained interleaving (even at the granularity of a single frame), enabling us to sample the performance counters or memory API after every frame.

Although the same Graphics-Graphics attacks through the memory API can also be implemented through a CUDA spy, we demonstrate a different attack that uses the performance counters. Our attack strategy is that we launch a CUDA spy application including many consecutive short CUDA kernels, in which the threads access the texture memory addresses that are mapped to different sets of texture caches (e.g. each SM on GTX 745 has a 12KB L1 texture cache with 4 sets and 24 ways). To make our attack fast and to optimize the length of each CUDA kernel, we leverage the inherent GPU parallelism to have each thread (or warp) access a unique cache set, so all cache sets are accessed in parallel. Note that number of launched CUDA kernels is selected such that the spy execution time equals to the average rendering time of different websites. The spy kernels collect GPU performance counter values. Although the spy can only see its own performance counter events, the victim execution affects these values due to contention; for example, texture cache hits/misses and global memory events are affected by contention.

5.3.2 Website Fingerprinting from CUDA Spy Using Performance Counters

On a CUDA spy application, we launch a large number of consecutive CUDA kernels, each of which accesses different sets of the texture cache using different warps simultaneously at each SM. The intuition is to create contention for these cache sets which are also used by the graphics rendering process. We run our spy and collect performance counter values with each kernel using the Nvidia profiling tools (which are user accessible) while the victim is browsing webpages. Again, we use machine learning to identify the fingerprint of each website using the different signatures observed in the performance counters. We evaluate this attack on 200 top websites on Alexa, and collect 10 samples for each website.

Classification: Among all performance counters, we started with those related to global memory and L2 cache: through these resources, a graphics application can affect the spy as textures are fetched from GPU memory and composited/rasterized on the screen. We used information gain of each feature to sort them according to importance and selected the top 22 features to build a classifier (shown in Table 5.6).

We summarized the time series of each feature as before by capturing the same statistical characteristics (min, max, slope, average, skew and kurtosis independently on two halves of the signal). Again, we trained three different machine learning algorithms (NB, KNN3, and RF) and use 10-fold cross-validation.

Table 5.7 reports the classification model based on the random forest classifier has the highest precision among all the tested classifiers. The average precision of the model on correctly classifying the websites is 93.0% (f-measure of 92.7%), which represents excellent accuracy in website fingerprinting. We also ranked the features based on their information gain and validated the capability of the random forest based machine learning model and observed prediction accuracy of 93.1% (with f-measure of 92.8%). This proves the feasibility of the program counter based machine learning models on identifying the websites running on the system. We obtained similar classification performance both with and without the GPU rasterization option. The classification precision on the default browser configuration (without GPU rasterization) is about 91%. Since still the texture should be fetched from memory and has effect on the texture cache and memory performance counters, while the GPU is only used for composition.

Table 5.6: Top ranked performance counter features

GPU Performance Counter	**Features**
Fb_subp0/1_read_sectors[2]	slope
Device memory read transactions	slope, mean
L2_subp0/1/2/3_read_sector_misses[3]	slope, sd
L2_subp0/1/2/3_total_read_sector_queries[4]	slope, sd
Instruction issued	skew, kurtosis

[2]Number of read requests sent to sub-partition 0/1 of all the DRAM units
[3]Accumulated read sectors misses from L2 cache for slice 0/1/2/3 for all the L2 cache units
[4]Accumulated read sector queries from L1 to L2 cache for slice 0/1/2/3 of all the L2 cache units

Table 5.7: Performance counter based website fingerprinting performance: F-measure (%), Precision (%), and Recall (%)

	FM	**Prec**	**Rec**
	μ (σ)	μ (σ)	μ (σ)
NB	89.1 (10.8)	90.0 (10.8)	89.2 (11.3)
KNN3	90.6 (6.6)	91.0 (7.6)	90.6 (8.2)
RF	92.7 (5.9)	93.0 (6.1)	92.7 (8.4)

5.4 Attack Model III: CUDA Spy on a CUDA Victim

To construct the side channel between two computing applications, multiprogramming (the ability to run multiple programs at the same time) on the GPUs is needed to enable the spy to run alongside the victim. As discussed in Chapter 2, modern GPUs support multiprogramming through multiple hardware streams with multi-kernel execution using a multi-process service (MPS) [167], which allows execution of concurrent kernels from different processes on the GPU and is supported in Kepler and newer microarchitecture generations from Nvidia. Multi-process execution eliminates the overhead of GPU context switching and improves the performance, especially when the GPU is underutilized by a single process. The trends in newer generations of GPUs is to expand support for multiprogramming; for example, the recent Volta architecture provides hardware support for 32-concurrent address spaces/page tables on the GPU. All three Nvidia GPUs we tested support MPS.

We assume that the two applications are launched to the same GPU. Co-location of attacker and victim VMs on the same cloud node is an orthogonal problem investigated in prior works [187, 53]. Although the model for sharing of GPUs for computational workloads on cloud computing systems is still evolving, it can currently be supported by enabling the MPS control daemon which start-ups and shut-downs the MPS server. The CUDA contexts (MPS clients) will be connected to the MPS server by MPS control daemon and funnel their work through the MPS server which issues the kernels concurrently to the GPU provided there is sufficient hardware resources to support them.

Once colocation of the CUDA spy with the victim application is established, similar to graphics-computing channel, a spy CUDA application can measure contention from the victim application. For example, it may use the GPU performance counters to extract some information about concurrent computational workloads running on GPU.

5.4.1 Attack III: Neural Network Model Recovery:

In this attack model, a spy computational application, perhaps on a cloud, seeks to co-locate on the same GPU as another application to infer its behavior. For the victim, we choose a CUDA-implemented back-propagation algorithm from the Rodinia application benchmark [64]; in such an application, the internal structure of the neural network can be a critical trade secret and the target of model extraction attacks. This attack is a proof of concept attack, and we believe that we can extend the same principles to explore general model extraction on arbitrary machine learning models.

We use prior results of reverse engineering the hardware schedulers on GPUs [161] to enable a CUDA spy to co-locate with a CUDA victim on each SM. We launch several hundred consecutive kernels in spy to make sure we cover one whole victim kernel execution. These numbers can be scaled up with the length of the victim. To create contention in features tracked by hardware performance counters, the spy accesses different sets of the cache and performs different types of operations on functional units. When a victim is running concurrently on the GPU and utilizing the shared resources, depending on number of input layer size, the intensity and pattern of contention on the cache, memory and functional units is different over time, creating measurable leakage in the spy performance counter measurements. We collect one vector of performance counter values from each spy kernel.

Data Collection and Classification: We collect profiling traces of the CUDA based spy over 100 kernel executions (at the end of each, we measure the performance counter readings) while the victim CUDA application performs the back-propagation algorithm with different size of neural network input layer. We run the victim with input layer size varying in the range between 64 and 65536 neurons collecting 10 samples for each input size.

As before, we segment the time-series signal and create a super-feature based on the minimum, maximum, slope, average, standard deviation, skew and kurtosis of each signal, and train classifiers (with 10-fold cross validation to identify the best classifiers for our data set).

Feature selection: We used information gain of each feature to sort them according to the importance and selected the top 20 features to build a classifier. Table 5.8 summarizes the most top ranked features selected in the classification and Figure 5.7 shows the information gain of the top features. We expected that cache and memory related features are most affected by concurrently running kernels. ”Issue stall” is also important as it measures contention from the victim on functional units and memory.

Table 5.8: Top ranked counters for classification

GPU Performance Counter	**Features**
Device memory write transactions	skew, sd, mean, kurtosis
Fb_subp0/1_read_sector[5]	skew, kurtosis
Unified cache throughput(bytes/sec)	skew, sd
Issue Stall	skew, sd
L2_subp0/1/2/3_read/write_misses[6]	kurtosis

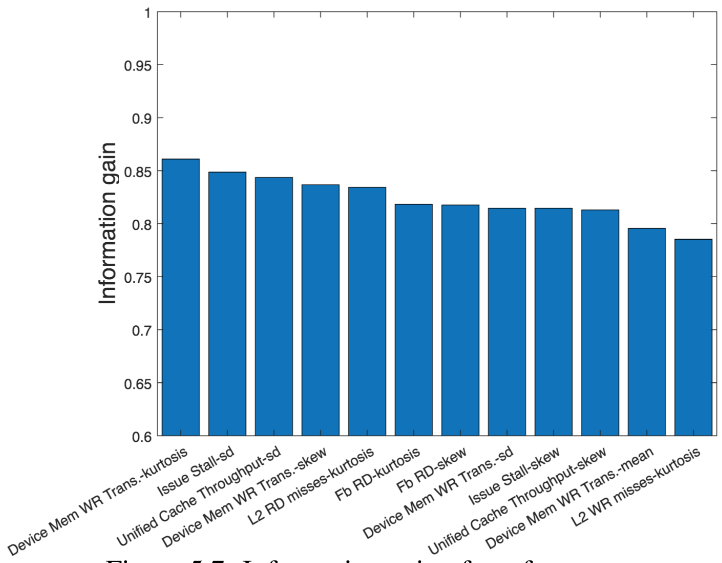


Figure 5.7: Information gain of top features

Table 5.9: Neural Network Detection Performance

	FM %	**Prec** %	**Rec** %
	μ (σ)	μ (σ)	μ (σ)
NB	80.0 (18.5)	81.0 (16.1)	80.0 (21.6)
KNN3	86.6 (6.6)	88.6 13.1)	86.3 (7.8)
RF	85.5 (9.2)	87.3 (16.3)	85.0 (5.3)

Table 5.9 reports the classification results for identifying the number of neurons through the side channel attack. Using KNN3, we are able to identify the correct number of neurons with high accuracy (precision of 88.6% and f-measure 86.6%), demonstrating that side channel attacks on CUDA applications are possible.

Attack in interleaved kernel execution model: In case that MPS is not activated on Desktop GPUs or is not the multiprogramming model on the cloud for concurrent running of spy and victim, kernels from two applications are scheduled based on time-sliced scheduling and context switching. We studied this scenario for machine learning models that launch several sequential GPU kernels and we launch a large number of very short spy kernels (each doing some memory and functional units operations) to make sure that spy and victim kernels are interleaving. We observed that per kernel performance counters read by spy kernels are affected by victim model parameters, since this context switching causes performance penalty (specifically on cache and memory related features) on the following kernel, enabling the spy to extract information from victim application.

5.5 Mitigation

The attack may be mitigated completely by removing the shared resource APIs such as the memory API and the performance counters. Since legitimate applications need these APIs, rather than removing them, our goal is to weaken the signal that the attacker gets. In the

[5]Number of read requests sent to sub-partition 0/1 of all the DRAM units
[6]Accumulated read/write misses from L2 cache for slice 0/1/2/3 for all the L2 cache units

future, GPU scheduling algorithms may be developed to create separation between workloads or to decorrelate the observed contention from the sensitive data operated on by the application. In Chapter 6, we propose a GPU-specific intra-SM partitioning scheme to isolate contention between victim and spy and eliminate contention based channels after detection.

We evaluate reducing the leakage by either (1) Rate limiting: reducing the frequency that an application can use an API such as the memory API or the performance counters; and (2) Precision limiting: limit the granularity of the reported information.

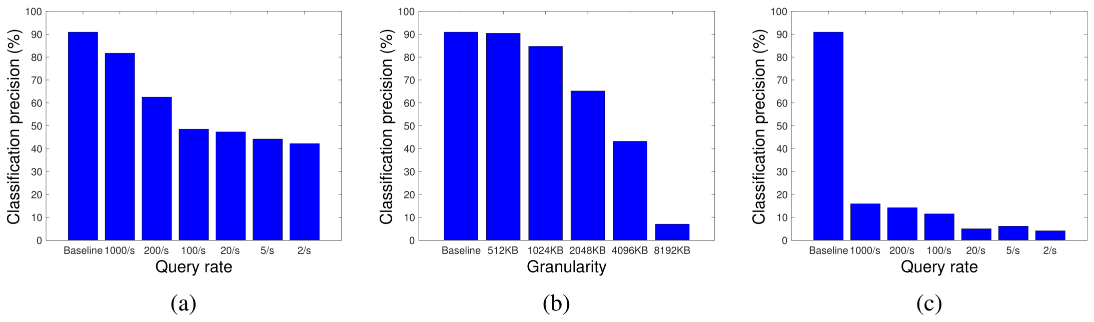


Figure 5.8: Classification precision with (a) Rate limiting; (b) Granularity limiting; (c) Rate limiting at 4MB granularity

We retrain the machine learning model with the leakage data obtained with the defenses in place on the Alexa top 50 websites. The classification precision decreases with rate limiting defense as shown in Figure 5.8a, and with reducing the granularity in Figure 5.8b. Reducing the query rate to two queries per second reduces precision but retains classification success of around 40%. In contrast, decreasing the granularity to 8192KB, the accuracy will be significantly decreased to about 7%. By combining the two mentioned approaches, using 4096KB granularity and limiting the query rate we can further decrease the precision to almost 4%, as demonstrated in Figure 5.8c. While reducing precision, we believe these mitigations retain some information for legitimate applications to measure their performance, while preventing side channel leakage across applications.

Although we evaluate the defense only for the website fingerprinting attack, we believe the effect will be similar for the other attacks, since they are also based on the same leakage sources. We also believe similar defenses can mitigate performance counter side channels.

Finding the right balance between utility and side channel leakage for general applications is an interesting tradeoff to study for this class of mitigations.

Chapter 6

Mitigation: GPUGuard

In Chapter 4 and 5, we demonstrated that GPUs are vulnerable to microarchitectural covert and side-channel attacks. GPU manufacturers are offering increasing support for multi-programming on GPUs to fully utilize the growing resource availability in GPU – wider data paths and more streaming multiprocessors (SMs). GPUs are now offered as a resource in cloud computing systems. A malicious VM can now spy on other applications that share a GPU (a side channel attack) or collude with another to covertly communicate sensitive information to bypass information isolation boundaries.

In this chapter, we provide a comprehensive solution to mitigate contention based covert- and side-channel attacks between two kernels co-executing on a GPU. At the core of these attacks an adversary/spy exploits contention in hardware resources to indirectly infer information about a victim kernel in the case of side channel attacks, or a colluding trojan kernel in the case of covert channel attacks. While solutions for such attacks in CPUs have been proposed, GPUs have a substantially different execution model with massive parallelism, internal hardware schedulers that impact colocation and contention, as well as several unique (micro)architectural structures such as constant cache, which provide a varied range of paths for contention based channel formation.

Our attack detection relies on the increased resource contention that is exhibited when a GPU is facing a covert or side-channel attack. In the context of CPUs, Hunger et al. have already shown that resource contention is one of the most quantifiable impact of an attack [106]. As such, GPUGuard non-intrusively monitors resource contention across kernels through a set of well defined features and resource utilization metrics. It then uses these features and metrics to classify kernel interaction behaviors, and to identify covert and side-channel formation. Once an attack is identified, the second component of our solution separates contending kernels into separate security domains, uniquely possible in GPUs due to the inherent spatial parallelism available, to close the identified contention channels. We use security domains at different hierarchy levels to maximize sharing (and performance) when it is safe, but to close contention based channels when there is a possibility for the existence of such a channel.

One simple solution to mitigate any information leakage through shared resources is to temporally partition the resources. But it has been shown in many recent studies that GPUs benefit greatly from fine-grain sharing, including intra-SM sharing [176, 230, 219, 220, 70]. As we show in Section 6.7, temporal partitioning alone results in nearly 2X performance penalty compared to our proposed scheme.

6.1 Threat Model: Covert and Side Channel Attacks on GPUs

We consider two threat models: covert channel and side channel attacks. For a covert channel attack we assume two colluding kernels that concurrently share the same GPU and desire to communicate sensitive data across protection boundaries (Chapter 4). Contention channels exploit differences in observed behavior caused by the presence or absence of contention on microarchitectural resources. They are able to measure contention by timing their operations, or by inspecting the hardware performance counters which are available in user mode in the current generation of GPU drivers. In a side channel attack context, one kernel (the attacker) is observing contention to infer secret information about a victim if its resource access pattern is dependent on sensitive data, as shown in Chapter 5.

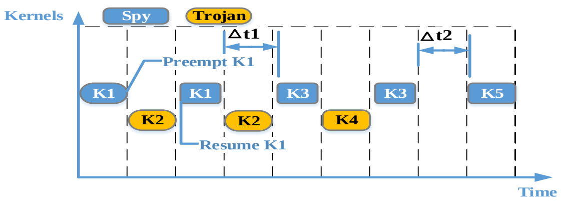
As an example covert channel scenario, a Trojan application can create contention on shared resource by replacing the contents of a cache set to encode '1' and leave the resource idle to encode '0'. The Spy application, on the other side accesses the cache and measures its access time to decode the transferred bit. Similarly, a Trojan application can create contention by excessively using execution units, warp scheduler, and instruction fetch units to encode '1' and leave those resource idle to encode '0', which spy can then decode.

6.2 GPUGuard Key Idea: Hierarchical Security Domains

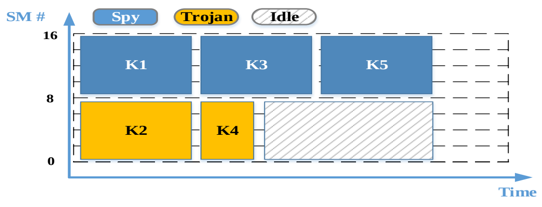
We propose GPUGuard, a holistic protection framework for GPUs to detect and defend against contention-based attacks. Figure 6.2a presents an illustrative example of GPUGuard. In this example, we assume that there are four applications concurrently running on the GPU, including two regular applications, a Trojan application, and a Spy application. Each application launches kernels to the shared GPU, which may be assigned to execute on the same SM. A GPUGuard classifier is designed to detect collusion between two kernels. Our defense mechanism will reschedule the suspected kernels into isolated security domains. For example, in Figure 6.2a, the GPUGuard identified that Kernel 3 and 4 are suspicious. The GPU now creates three isolated security domains (SD1-3) and issues Kernel 1 and 2 to SD1, Kernel 3 to SD2, and Kernel 4 to SD3. In this way, the timing channel between Kernel 3 and 4 is closed.

6.2.1 Security Domains

It is critical to define the scope of a security domain to minimize the performance overheads. Rather than using a one-size-fits-all approach GPUGuard uses a hierarchy of security domains with varying scopes. Depending on the type of attack detected GPUGGuard employs a security domain that encompasses only those resources that are being used in channel formation.



(a) Temporal Partitioning



(b) Spatial Partitioning

Figure 6.1: Existing partitioning techniques on GPUs

Coarse-grain security domain using temporal partitioning. At a coarse grain level, GPUGuard uses temporal partitioning which is essentially time division multiplexing. As shown in Figure 6.1a, using temporal partitioning each kernel can only execute in its assigned time slots. Note that in this example the Spy kernel $K1$ (blue) can only execute in the odd time slots, while the Trojan kernel $K2$ (yellow) can only execute in the even time slots. In this case, the execution of the kernels are completely isolated. As shown, the kernel waiting time $\Delta t1$ and $\Delta t2$ are independent of the execution time of the other kernel.

GPUs rely on kernel level preemption (essentially context switch-es) to enforce temporal partitioning: when $K1$ reaches the end of its assigned slot, the GPU needs to save the kernel context, preempt the kernel and then schedule the next kernel $K2$ to run on the GPU. After $K2$ uses up its time slot, $K1$ must reload its context and then resume execution. Context switching on GPUs is more expensive than on CPUs [178, 219]. On the Pascal architecture which supports optimized preemption, kernel preemption takes 100 micro seconds – a 100K cycle penalty even

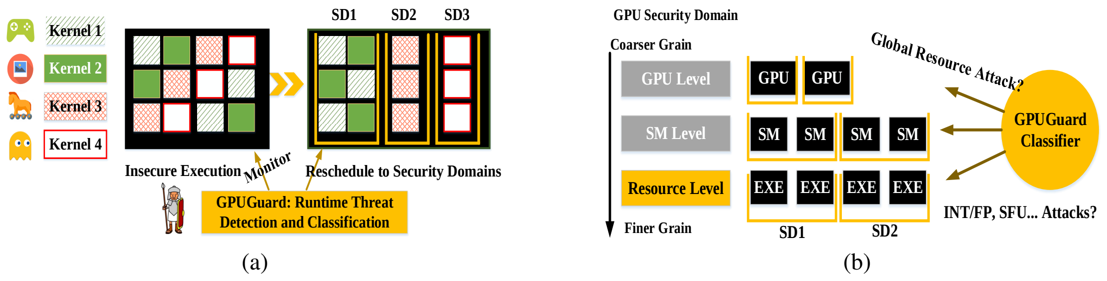


Figure 6.2: An illustrative example of a GPU system with GPUGuard (a) monitor and detect timing attacks; (b) select a security domain level based on specific attack type (our contributions are highlighted)

when using a 1GHz GPU. Thus temporal partitioning alone is an expensive solution for providing isolation.

Finer-Grain Security Domains Using Spatial Partitioning. The next level of isolation can be achieved through a hierarchy of spatial partitioning approaches. Adriaens et al. proposed spatial partitioning at the granularity of SM to partition GPU resources across multiple kernels, primarily to improve resource utilization [48] . This technique can be easily adapted to create multiple security domains on the same GPU. Figure 6.1b shows an example with 16 SMs and with spatial partitioning the Trojan kernel $K1$ occupies SMs 0-7 while the Spy kernel $K2$ occupies SMs 8-15. Because the kernels are separated on different SMs, no contention can be established through intra-SM resources, such as execution units or L1 caches. However, spatial partitioning at the granularity of an SM does not protect against attacks through globally shared resources such as the L2 cache, memory channels, and interconnection network.

Spatial partitioning is a heavy handed solution: the entire resources are strictly partitioned leading to significant performance hit when there are no attacks. A finer grain isolation can be provided using intra-SM partitioning. It has been demonstrated that sharing a single SM (intra-SM sharing) across multiple kernels can provide higher system throughput and better utilization of GPU resources compared to spatial partitioning [230, 219, 220, 70]. Hence, we argue that the benefits of intra-SM sharing must be delivered to the end user, without compromising the potential security risks.

Putting it all together. Figure 6.2b summarizes how GPUGuard may use both temporal and spatial partitioning at various granularities to achieve the required isolation with minimum performance overhead. As shown in Figure 6.2b, we can partition the four SMs into two security domain SD1 and SD2, each containing two SMs. GPUGuard may also use intra-SM partitioning of parallel execution lanes or utilizing other underutilized resources inside an SM to create multiple security domains. For example, assuming that there are four execution lanes in the special functional units inside an SM, GPUGuard can assign the first two lanes to SD1 and the remaining two lanes to SD2. This partitioning can be achieved through security aware warp folding which we will introduce shortly. Note that many GPU workloads have shown significant warp level divergence [229, 112], and lane level partitioning in many cases improves the resource utilization. Through this hierarchy GPUGuard activates the right amount of isolation for preventing collusion while still maximizing the benefits of fine-grain intra-SM resource sharing across kernels.

6.3 Attack Types

We assume a conventional covert communication scenario with a Trojan and Spy kernels from two different applications that are co-executing on the same GPU and wish to communicate covertly. The attack benchmarks we used in this work are intra-SM and inter-SM microarchitectural covert channels on GPUs, categorized into five groups modeled on attacks in Chapter 4 and summarized in Table 6.1. Based on current generation GPU microarchitectural details that are publicly known, we believe these five attacks cover a wide range of information leakage through shared resources.

In our experiments, we couldn't reliably measure timing variance through shared memory bank conflicts to construct a covert communication, hence, we do not consider such attacks in this work. It is possible with careful reverse engineering that such an attack can be constructed, but we will leave it for future work. We also do not directly address L2 cache attacks, where the L2 cache may be shared across multiple SMs. In this case we believe that GPUGuard can simply switch to coarse grain temporal partitioning to mitigate L2 cache attacks. Finally, we believe that our decision tree classification approach is general enough to tackle new intra-SM attacks by including such attacks into the training set to retrain the classifier and reprogram the detection units.

6.4 Attack Detection

To detect such covert channel attacks, GPUGuard continuously monitors the activation and resource usages of running kernels. GPUGuard employs a decision tree classifier that reads readily available performance counter metrics to track kernel behaviours and identify suspicious contentions. We elected to use decision tree, a machine learning model, as it is robust to noise that can fool deterministic threshold based detectors.

The attack detector continuously monitors the execution status, resource utilization and various other performance counters (e.g. cache miss rates) for different active kernels. A selection of features is extracted periodically (once every 1000 cycles in our setup) from the collected performance counter statistics and are used by a machine learning classifier to detect whether there are suspect timing channels between any two concurrent kernels. The output of the classifier is a label we assigned to different attacks and normal application. We develop a multi-class classifier that not only detects suspected timing channel presence, but also determines the target shared resources that are used to communicate covertly.

Table 6.1: Description of attacks in our dataset.

L1 Cache Attack: Trojan accesses one or multiple cache set(s) to send ”1” and Spy accesses the same set(s) and measures the access time. Attacks can target L1 constant, instruction, data or texture caches.
Execution Unit Attack: Trojan threads do a number of double or single precision ops to create contention on INT/FP units to send ”1”. Spy threads do the same ops and measure the execution time.
SFU Attack: Trojan threads do a number of special function operations (like __sinf) to create contention on SFUs to send ”1”. Spy threads do the same operations and measure the execution time.
Scheduler Attack: These are timing channels created as a side effect of a primary EU and SFU attack, typically leaking information by observing warp scheduler contention.
Atomic Attack: Trojan threads do atomic ops on global memory addresses (one particular address or strided addresses to achieve coalesced or uncoalesced accesses) to send ”1”. Spy accesses the same pattern and measures the access time.
**In all attack scenarios, high measured latency by the Spy is decoded as ”1” and low latency is decoded as ”0”.

Decision Tree Classifier. Without loss of generality, we use a decision tree based classification algorithm to classify the attack type. Decision trees are a supervised learning algorithm in which the classification model is built by breaking down a dataset into progressively smaller subsets based on a feature value at every decision point. This classification may be viewed as a tree structure where each level progressively refines the classification of an input. The two most important advantages of decision trees over other classification models are: (1) small hardware overhead (see Section 6.7.3); and (2) direct isolation of relevant feature elements through an estimate of information gain. We use the ID3 algorithm to build the tree [184], which uses entropy and information gain to identify appropriate decision points, and employs a top-down greedy search through the space of possible branches with no backtracking. Once the tree is constructed based on the training data, a new instance is classified by starting at the root node of the tree, testing the attributes (or feature elements) specified by this node, then moving down the tree branch corresponding to the value of the attribute. This process is then repeated to reach a leaf node [155].

As with any classification algorithm there are two issues that we must address. First, the classifier may have both false-positive and false-negative classifications. We show later in our results that 0% of the malicious kernels were classified as benign and only 8% of benign kernels were classified as malign. Hence, we believe misclassification is not a concern. The second challenge is that an attacker may design Trojan/Spy pairs that continuously shifts between benign behavior and malicious behavior that may cause the decision tree to lower its threshold. But to alter such a behavior the attacker must first observe the information leakage to adapt, but such covert channel bandwidth is going to be drastically reduced in the first place using GPUGuard. Hence, the time to adapt will be extended significantly which we believe is the primary deterrent in covert and side-channel attacks.

Feature Selection. The decision tree model is built using a training input set consisting a large collection of features that correspond to various resource utilization indicators, covering different types of covert channels. Table 6.2 lists all the features that were collected as inputs for the decision tree model.

The feature vectors were divided based on whether they came from the training or testing data set benchmarks. The data in each of the sets was obtained from running the benchmarks and attacks that belong to that set only. The decision tree model is trained using the training input set. Once the training is complete, we obtain the decision tree model parameters

Table 6.2: All collected features to create dataset.

Instruction features: # of **SP-INT**, **SP-FP**, **SFU** and LD/ST issued # of decoded **ALU**, **SFU**, **ALU-SFU**, **INT**, **FP**, Load & Store ops, # of decoded Branch, Barrier, Memory barrier, Call, Ret, **Atomic ops**, # of decoded INT MUL/DIV, FP MUL/DIV, **FP sqrt/log/sin/exp**, **SP-FP**, INT/FP/SFU/MEM instructions processed in decode stage) # of decoded Tex/non-Tex ops, some cache related instruction opcodes such as LD_OP, LDU_OP, Total stall: # of cycles warp scheduler issues no warp to execute Total issue: # of cycles warp scheduler issues a warp to execute
FU features: SP-INT, **SP-FP**, **SFU** and **LDST util.** at execute stage
Mem features:# of cycles warp scheduler stalled by long memory latency L1D accesses/misses/evictions, **L1C accesses/misses/**evictions **L1I accesses/misses/**evictions, L1T accesses/misses/evictions **L2 accesses/misses/**evictions, L1C, L1D, **L1I** and L1T **accesses per set** L1D read hit/miss per set, L1D write hit/miss per set L2 read hit/miss per bank, L2 write hit/miss per bank

which include the weight of each of the features in determining the attack type. We then optimize the tree by pruning unimportant features. To identify the most important feature elements, we use the decision tree model for multiclass classification to compute the importance factor of each feature element based on the training subset. Through this selection, we were able to reduce the 234 features to a set of 24 that are identified as the most important features. As a result our trained decision tree classifier takes as input a select set of microarchitecture features collected at runtime, which are listed in bold font in Table 6.2.

Pruning the feature set allows us to reduce the complexity of the classifier, without sacrificing detection accuracy. The identified important features are related to the resources that are used by our attack benchmarks to create contention; intuitively, the decision tree checks the utilization of each resource to identify the presence and type of contention. We then classify the test set based on our decision tree model using two-fold cross validation. Section 6.7 evaluates the accuracy of our online detection, based on classification results for each instance in the test set.

Feature Collection. The 24 selected features are sampled for each active kernel periodically (every 1000 cycles) and then fed to the classifier. An effective online detector needs to filter out occasional false classifications and quickly signal true malicious behavior. Ozsoy et al. [173] adopted an Exponentially Weighted Moving Average (EWMA) approach for their binary classification. For the same reason, we design a voting algorithm that first considers classification results from the decision tree classifier (making the time-series consist of 0's for benign application and 1-5's for five attack types.) We then use a window of 10 of these decisions and pick the majority decision as the correct answer and output whether an attack is in progress and, if so, which of the five attack types is being used. Synthesis results show that the extra hardware overhead of the classifier is not high (shown in Section 6.7.3).

Detector Adaptation. Although we believe the attacks in our training set cover exploitable shared resources, if in the future new contention channels are encountered due to microarchitectural enhancements to SMs, they can be addressed by retraining the model with new training data. Since classification is implemented in hardware we need to provide the ability to adapt the detector at runtime if new attack vectors are identified during in-field operation. To make this adjustment feasible we expect the boot loader to essentially re-program the node weights in the tree, which are implemented using registers.

6.5 Tangram: Attack Mitigation

The second component of GPUGuard is a defense against covert channel attacks once an attack is detected. We refer to the GPUGuard's defense mechanism alone as the Tangram (shown in Figure 6.3). Tangram uses a hierarchy of resource slicing to prevent the attack types that are detected. The approach to partition resources is similar to a dissection puzzle called *Tangram*. Tangram uses hierarchical security domains to separate colluding kernels, starting with fine grain intra-SM slicing and then gradually moving towards coarser-grain inter-SM slicing, depending on the attack type. In particular, GPUGuard uses intra-SM spatial partitioning of resources to mitigate L1 cache attack, execution unit, and SFU attacks. For Atomic attack, GPUGuard uses temporal partitioning. The only partitioning that is not used in GPUGuard is partitioning of SMs into clusters of security domains, since all attacks can be mitigated with either intra-SM spatial partitioning or temporal partitioning.

As shown in Figure 6.3, and described in more details below, security domain are created using sliced data pipelines (1), controlled memory request traffic within memory units (2), rate limited scheduling in the warp scheduler (3), and L1 fetch arbiter (4). The maximum number of security domains for each resource is set as four. Hence, at most four suspicious kernels can be isolated in a given SM. This is a reasonable limitation since more than four concurrent kernels inside the same SM have diminishing returns in performance benefits [219].

Mitigating Execution Units and SFU Attack: Datapath Slicing. As in our baseline GPU, we assume 32 execution lanes within a single SM. Datapath slicing splits the 32 execution lanes into four slices, each with eight lanes. Note that GPUs already treat the 32 execution lanes as a collection of clustered lanes that are operated semi-independently of others. For instance, AMD's subwarp execution model allows multiple warps to share the same set of datapaths. No matter if they are from different kernels or protection domains subwarp execution requires the same control logic for each of the sub datapaths.

GPUGuard relies on the subwarp execution model where each slice is allocated to a single kernel thereby preventing one kernel from observing the SFU and execution unit usage of another kernel. Note that datapath slicing does not change the number of threads inside a warp or the number of register file banks in the SM. Datapath slicing folds the 32 threads in a warp into four quarter-warps, which are then issued in succession. The threads in a warp are shifted in successive cycles to align with the slice in a linear fashion: threads 0-7 are mapped to lane 0-7 in the first cycle, threads 8-15 are mapped to lane 0-7 in the second cycle, and so on.

Datapath slicing allows for concurrent warps from different security domains to be executed simultaneously on isolated datapath slices. With four slices each warp executing on a slice needs to be executed in four consecutive cycles, incurring a three cycle delay for a given warp. While a sliced pipeline delays the execution of each warp, the total throughput of the GPU is similar to, or in some cases even better than, a unified 32-lane pipeline. When there are control divergence, a sliced pipeline can help fill out the idle resources more effectively and improves the performance.

Tangram relies on some additional hardware support for executing multiple sub-warps concurrently. Tangram adds a set of 32 registers immediately before and after each data pipeline

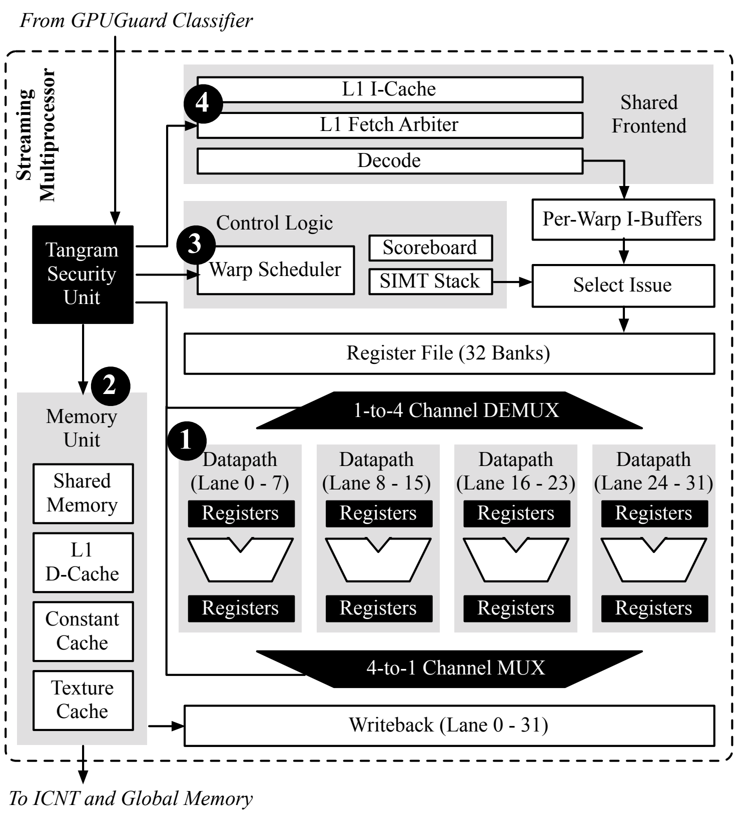


Figure 6.3: Tangram overview (black units are modifications)

slice (shown in Figure 6.3), so that the registers can be consumed and updated in multiple successive cycles. However, the proposed design does not require any modification to the interconnection network between the register banks and the data pipeline. The interconnection still forwards the registers into the same original 32-wide registers of the pipeline. Then a 1-to-4 channel de-multiplexer is added to shift the register to the add-on 32-wide register of a particular slice. The write-back process follows a similar path: the results of a slice will be stored locally in the add-on registers and shifted out to the original 32-wide write-back register through the 4-to-1 channel multiplexer. After each sub-warp finishes, the caching register shifts left by 8 words to feed into the next sub-warp.

Cache partitioning is a reasonable mechanism in CPUs to create separation and remove contention channels. Since GPUs have multiple cache types, we use a novel cache redirection approach instead of partitioning. To mitigate covert channels from constant cache accesses, Tangram dynamically re-routes traffic from constant cache to use the L1-D cache. For this purpose, Tangram relies on the decision tree classifier to categorize the attack type as the constant data cache attack, and in which case the constant values accessed by one malicious kernel are moved to the L1-D cache. Once an attack is detected, Tangram monitors constant data load operations from one of the malicious kernel. The load address is first looked up in the constant cache. If there is a hit in the constant cache, Tangram marks it as a miss and evicts the data from constant cache and places a miss fill request to bring the data back into the L1-D cache. For this purpose, Tangram uses the constant data address to lookup the L1D cache to find a victim cache line. The victim cache line is evicted and constant data is then stored in that line. Thus, the channel through the constant cache is eliminated. If the attacker detects the protection and then changes to use L1 data cache, Tangram will eliminate the covert channel formed through L1 data cache using cache bypassing. Previous studies show that the GPU L1D cache miss rate is so high so that the performance is not harmed when the GPU L1-D cache is bypassed [134, 228, 201, 66, 113, 133]. Therefore, Tangram selectively bypasses the L1-D cache requests if the attacks are detected on the L1D cache instead. Since it is not possible to re-purpose the read-only constant cache for potentially read/write operations from a regular L1-D cache, our approach simply picks either the Spy or Trojan kernel and mark all its load/store operations as non-cacheable.

Mitigating Scheduler Attacks. When a kernel modulates Execution Units/SFU/Cache accesses, in addition to the contention on one specific unit, the attack often creates weaker side attack contention on other shared resources, including shared warp scheduler and instruction fetch units. Thus we add the following techniques to mitigate those side attack channels.

Rate Limiting Warp Scheduler. The GPU warp scheduler selects which warps will be issued to execute in the next cycle from a pool of all the active warps in the SM. The warps from multiple security domains can compete for the scheduling bandwidth and issue timing attacks.

Our baseline warp scheduler selects the next available warps to issue based on the last issued first and then the oldest order. When all the warps from a kernel are stalled, all the scheduling cycles will be given to the next kernel. On the other hand, if the warps from a kernel are always ready to execute, it will consume all the scheduling bandwidth and starve the other kernel. This interference in scheduling can be manipulated for timing attacks. Therefore, we enhance the warp scheduler with a rate limiter, so that scheduling cycles will be fairly distributed. For example, if one warp scheduler can issue up to two warps in each cycle, and there are two security domains, we will ensure that only one warp from each security domain can be issued. In the case of four security domains, one warp from each security domain can only be issued every other cycle.

Instruction Fetch Arbitration. Tangram prevents contention on the instruction cache using instruction fetch arbitration. A malicious kernel may intentionally saturate the instruction fetch bandwidth. Tangram alters the control unit in the L1 fetch arbiter so that it will successively fetch from different security domains in a round-robin manner. Therefore, each security domain gets fair access while simultaneously preventing resource hogging by a single kernel.

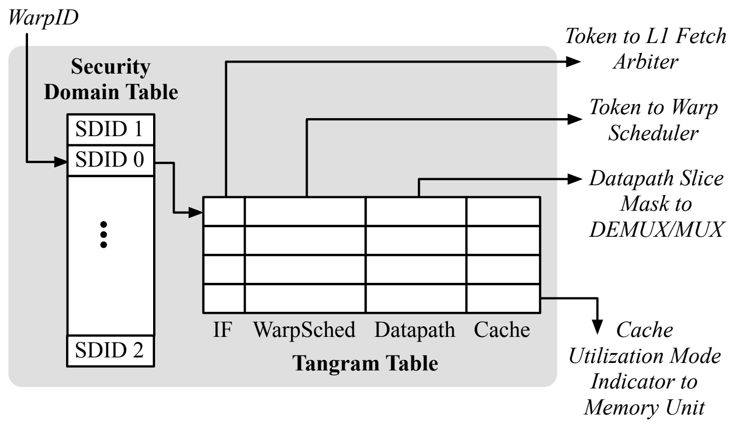


Figure 6.4: Logical View of Tangram Security Unit

Mitigating Atomic Attack: Temporal Partitioning Global memory attacks are primarily carried through atomic operations to measure contention in memory channel. When such an attack is detected by our classifier we fall back on temporal partitioning of the SM. In particular, we context switch the two malicious kernels (without perturbing the normal kernels). Since context switching is an expensive operation, we only use this option for tackling covert channels formed through atomic operations. The associated performance penalty is primarily paid by the colluding kernels, and only in very rare cases by regular kernels if they were misclassified as colluding kernels.

6.5.1 Security Unit Implementation.

The various schemes described above defend against different types of attacks. Based on the attack classification, the coordination across various schemes is handled using the *Tangram Security Unit* (TSU), shown in Figure 6.4. When all the kernels are executing normally, TSU keeps all kernels in a single security domain and none of the resource partitioning schemes described above are activated. However, when the classification algorithm detects an attack, the warp ids of the two colluding warps are sent to the TSU. TSU then activates the resource splitting across security domains. Each kernel is assigned a security domain id and all warps in that kernel execute within that security domain.

TSU maps warp IDs to security domain IDs using the *Security Domain Table* (SDT). The obtained security domain ID is used as the index of *Tangram Table*, which tracks the resources assigned to each security domain. In this way, TSU guarantees kernels are executed in isolated security domains.

Each Tangram table entry consists of a 3-bit instruction fetch token, a 3-bit warp scheduling token, a 4-bit datapath slice mask, and a 16-bit cache utilization mode indicator. The instruction fetch token and warp scheduling token are used to determine a warp's scheduling slots out of the total scheduling cycles during a given observation window. For example, if the warp scheduling token for SD1, SD2, and SD3 are one, one, and two, respectively, and the warp scheduler can issue two warps in each cycle, then in a two cycles window, the number of warps can be issued by each of them is one warp for SD1, one warp for SD2, and two warps for SD3. To ensure fair access for different security domains, all the tokens will be initially set to one.

The datapath slice mask has 4 bits, each corresponding to a datapath slice (Recall that we have four datapath slices). If bit 1 of the datapath slice is cleared (set to 0) for a warp then that warp cannot be issued to datapath slice 1. Thus each warp is restricted to execute only on those slices whose corresponding bits in the datapath slice number are set to 1.

The last field in the Tangram entry is the 16-bit cache redirection mode, which is used to provide fine grained security protection while accessing caches. In our baseline GPU there are four caches (shared memory, L1 D-Cache, constant cache, and texture cache). Each of the above four caches has a corresponding 4-bit cache redirection mask (so a total of 16-bits). When an incoming memory request is bound for a given cache type, Tangram looks up the 4-bit mask associated with that cache to determine if the request need to redirected to another cache type. For instance, if a constant cache access request from a security domain is isolated to use L1 D-cache then all requests to the constant cache will use the corresponding 4-bit mask in Tangram to initiate that redirection. Similarly if all the four bits associated with a given cache type are zero, the request traffic control logic will redirect all requests from that cache type to go to the global memory (in response to a detected atomic attack). In this way, the access to caches are always going to be re-directed to the other under-utilized resources or the global memory to guarantee the execution isolation.

The TSU access latency is smaller than a clock cycle, and is off the critical path: the instruction fetch token is obtained one instruction in-advance; the warp scheduling token is retrieved in parallel with accessing the SIMT stack; the datapath slice mask and cache access mode are collected by the operand collector with other operands. The average power and area overhead of TSU are negligible compared to the entire system. Detailed analysis is presented in Section 6.7.3.

6.6 Experimental Setup

In this section, we discuss the experimental methodology.

6.6.1 Architecture

We use GPGPU-Sim v3.2.2 [55], a cycle accurate timing simulator, in our evaluation. Our configuration parameters are described in Table 6.3. For Volta architecture, the parameters are set based on NVIDIA's white paper [20], and HBM2 timing is set based on previous work [169]. The simulator was extended to run multiple applications concurrently, and we abide by the GPGPU-Sim model to assume that all the data fed into a kernel fits in the GPU device memory. We used the same set of attack benchmarks from Chapter 4. Those attacks are fully validated in GPGPU-Sim against real GPU hardware, and hence our results using the GPUGPU-Sim simulation infrastructure accurately model the attacks observed in hardware.

Table 6.3: Configurations of Fermi, Kepler and Volta architecture.

	Fermi (GTX580)	Kepler (K40)	Volta (Titan V)
Compute Units (SIMD width 32)	16x2 SP, 16x1 SFU, 700 MHz	16x6 SP, 16x1 SFU, 745 MHz	84x2 SP, 84x1 SFU, 600 MHz
Regs / Shmem / Max # CTA	32768 / 48 KB / 8	32768 / 48 KB / 8	262144 / 96 KB / 32
Warp Schedulers (1 per SM)	max issue 2 warps per cycle, default gto	max issue 4 warps per cycle, default gto	
L1 Cache	16KB 4-way L1D, 4KB 4-way L1C		16KB 4-way L1D, 8KB 4-way L1C
L2 Cache	768 KB	768KB	4608KB
Memory Model	6MCs, GDDR5: t_{CL}=12, t_{RP}=12, t_{RC}=40, t_{RAS}=28, t_{RCD}=12, t_{RRD}=6		64MCs, HBM2: t_{CL}=12, t_{RP}=12, t_{RC}=40, t_{RAS}=28, t_{RCD}=12, t_{RRD}=6

We evaluated the performance impact of multiple defense schemes: temporal partitioning (labeled as TP in all our results), spatial partitioning through clustered SMs (labeled as SP in all our results), and GPUGuard against having Trojan and Spy kernels insecurely sharing an SM without protection. In temporal partitioning, each kernel is assigned an execution window of 50K cycles in a round robin manner. At the end of the 50K cycle window, we will preempt the current kernel and switch to the kernels in the next security domain. In clustered SM approach SMs are evenly allocated to Trojan and Spy kernels. In GPUGuard, datapath slicing, fair warp scheduling and instruction fetch are turned on for all benchmarks and for Atomic attacks GPUGuard falls back on temporal partitioning as we described earlier.

6.6.2 Workloads

We selected 40 readily available applications as benign samples from a collection of benchmark suites [64, 55, 197, 165]. We then extended the GPU covert channel attack applications based on attacks from Chapter 4 and hand-coded 250 different pairs (Spy and Trojan) of malicious applications, which cover atomic operation attacks (Atomic), constant cache attacks (Cache_A and Cache_B), attacks on execution units (ADD and MUL), and special functional units (SFU). These different attacks create orthogonal types of channels between Trojan and Spy by using different resources. They also differ with respect to implemented optimizations (e.g., Synchronization via handshaking through different cache sets [161] and Multi-bit communication), as well as the communication rate and the communicated data. Thus, the attack variants exhibit substantially different contention behavior. We also implemented prime-and-probe style side-channel attacks on constant cache which is run with different normal programs.

We split both the benign applications and attacks into separate training and testing sets, so that 60% of the benchmarks are used as training set and other benchmarks as testing set. The benchmarks are run on the GPGPU-Sim simulator [55] to collect a 24-entry feature vector for each kernel at each sampling window. Nvidia nvprof report GPU performance counter values only after each kernel termination, that is too coarse grain for our scheme. We empirically set the default window size to 1000 cycles, while providing a comparison of window sizes in Section 6.7.1. These feature vectors are the input to the decision tree classifier. The output is a label we assign: (0) normal application, (1) L1 cache attacks, (2) global memory attacks with atomics, (3) execution units attacks, and (4) SFU attacks.

To reduce the simulation time, a subset of the covert channel attacks is used for system evaluation: four versions for each type of attack, with various inputs, programming styles, and implemented optimization. For constant cache attacks, we evaluated both the base attacks that contend using one fixed set (Cache_A) and the improved attacks that continues to probe all the cache sets to communicate (Cache_B). To quantify the performance impact of control divergence, half of the execution unit benchmarks have little divergence, while the other half have 25% - 50% of control divergence. We further run the defense schemes on 34 randomly selected, normal application pairs from the detection sets to study the performance penalty of FP predictions. The application parameters follow the benchmark sets used in [230].

6.6.3 Synthesis

The decision tree based classifier and control logic of GPUGuard were designed and verified in Verilog RTL, and synthesized with the FreePDK 45nm library [195] using the Synopsys Design Compiler [198]. FabMem [67] is used to model the security domain table and the Tangram table within the Tangram Security Unit, and also register buffers. The latency, energy, and area overheads were all taken into account.

6.7 Evaluation

In this section, we evaluate and discuss the accuracy of the attack classifier, and analyze the performance and energy impact of the proposed defense scheme to the entire system. We also report the latency, area, and power overheads of the components in the proposed GPUGuard technique.

6.7.1 Detection Accuracy

Figure 6.5a shows the confusion matrix. The first column indicates the actual attack type and the last row shows the predicted attack type; attack types are numbered to match description of attacks shown in Table 6.1. As shown by the strong diagonal matrix the predicted and actual attacks are close in most cases. The classification accuracy is measured to be 93.8% with window size of 1000 cycles. The detection accuracy with window size of 5000 cycles was even higher at 97% for this experiment, but we selected 1000 cycles for faster detection.

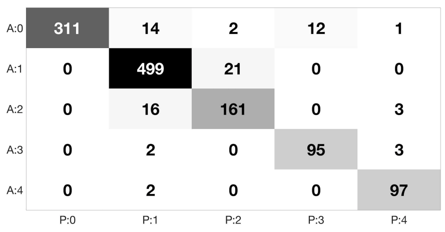
The accuracy is also measured using true positive (TP), false negative (FN), true negative (TN), and false positive (FP). In our results, 8.5% of regular applications were misclassified as malicious applications (FP), but 0% malicious applications were misclassified as regular applications (FN). FP cases cause the system to react unnecessarily (performance penalty) while FN evades detection which represents a security concern. Note that the TP rate of the classification is 91.5% and TN rate is 100%, indicating that our detection can reliably signal a malicious behavior, and if not, the running applications are truly benign.

Covert channel attacks rely on contention on shared resources to communicate encoded messages, and the decision tree classifier takes into account many features that are related to such contention, including resource utilization, cache misses and many others. The model trains a decision tree predictor by optimally setting the thresholds of the features to detect the contention level. The structure of the decision tree model fits very well with the problem we are solving, and therefore, yields a high accuracy. It must also be noted that the data set we used to build the decision tree is *disjoint* from the data set of benchmarks used for evaluation.

Multiple Channel Attack. Training and evaluation described above are performed using applications communicating over a single channel. Since we monitor performance counters that capture the contention for all cache, memory, and execution units, if the attacker changes its behavior to communicate over a different hardware resource or attempts a multiple channel attack, the contention is also detected. To get more accurate classification, we need to add those samples to our training set with the correct label. We hand crafted a multiple channel attack that combines all four attacks listed in Table 6.1. The existing scheme successfully classified the application as an attack. To support concurrent multiple channels on different resources, one straightforward solution is to enable temporal partitioning, once a multiple channel attack is detected.

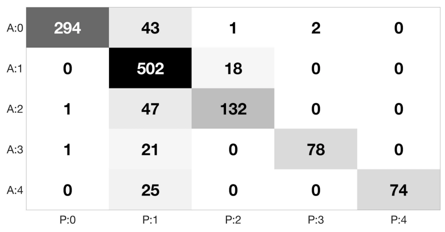
Comparison to Neural Network Model Detection. We implemented a multi-layer perceptron (MLP) artificial neural network model to compare the classification results to our decision tree based detection. In our MLP implementation the input layer contains neurons equal to the number of features (24 important features in our case), and the output layer contains neurons equal to the number of classes (five in our case). The data is fed to the input layer of the network, and after the feed-forward propagation, the output layer of the network contains a vector of values. The neuron containing maximum value determines the class of the data. The error in prediction is calculated and using this error the weights of the network are modified by gradient descent algorithm. The MLP based classification accuracy is measured at 87%. Figure 6.5b shows the total confusion matrix which visualizes the performance of MLP classification. Decision tree based classification outperforms MLP in our dataset.

Robustness. We also evaluated the detector accuracy when attack kernels (Spy and Trojan) are running with normal kernels at the same time. In this situation, it is harder for the detector to accurately classify attacks due to the contention noise introduced by normal kernels. Our results show that the classification accuracy for decision tree based and MLP based detection are 91.5% (95% with a window size of 5000) and 85.1% respectively. Figure 6.6 shows the total confusion matrix for these two detection schemes.



	P:0	P:1	P:2	P:3	P:4
A:0	311	14	2	12	1
A:1	0	499	21	0	0
A:2	0	16	161	0	3
A:3	0	2	0	95	3
A:4	0	2	0	0	97

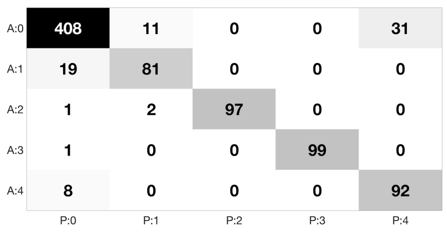
(a) Decision tree classification



	P:0	P:1	P:2	P:3	P:4
A:0	294	43	1	2	0
A:1	0	502	18	0	0
A:2	1	47	132	0	0
A:3	1	21	0	78	0
A:4	0	25	0	0	74

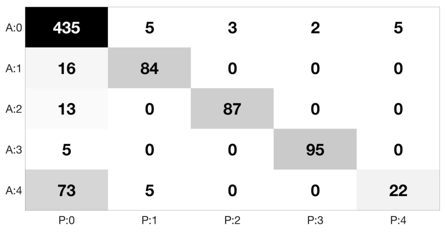
(b) MLP based classification

Figure 6.5: Confusion matrix for decision tree and MLP



	P:0	P:1	P:2	P:3	P:4
A:0	408	11	0	0	31
A:1	19	81	0	0	0
A:2	1	2	97	0	0
A:3	1	0	0	99	0
A:4	8	0	0	0	92

(a) Decision tree classification



	P:0	P:1	P:2	P:3	P:4
A:0	435	5	3	2	5
A:1	16	84	0	0	0
A:2	13	0	87	0	0
A:3	5	0	0	95	0
A:4	73	5	0	0	22

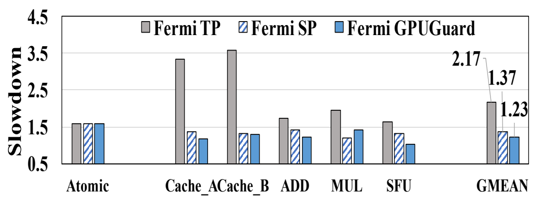
(b) MLP based classification

Figure 6.6: Confusion matrix for decision tree and MLP based classification under higher contention noise.

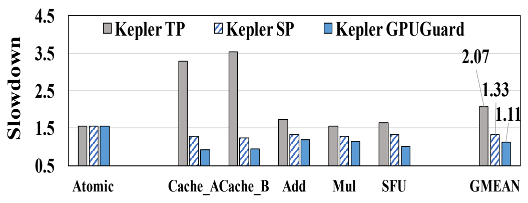
To further evaluate the detection robustness, we consider the attack benchmarks that are intentionally designed to avoid detection by lowering the communication bandwidth. Specifically, we change the Spy and Trojan codes by adding extra delay between communicating two consecutive bits, to reduce the channel bandwidth by 2x, 10x up to 10^5x. 10^5x slow down reduces the absolute BW from 30kbps to 0.3bps for constant cache attacks. Based on our experiments, our detector accurately detects the contention when Trojan tries to send a '1' to Spy (with the same accuracy of the attacks without slowdown). On the other hand, the classifier will not detect applications as attacks in the longer idle periods (no communication), since there is no contention.

6.7.2 Performance Impact

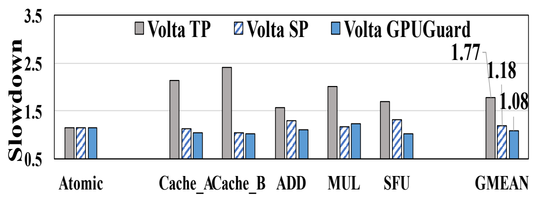
Figures 6.7a to 6.7c show the performance of all the defense schemes compared to intra-SM slicing without protection on NVIDIA Fermi, Kepler, and Volta architecture. For constant cache attacks, temporal partitioning (labeled TP) alone slows down program execution



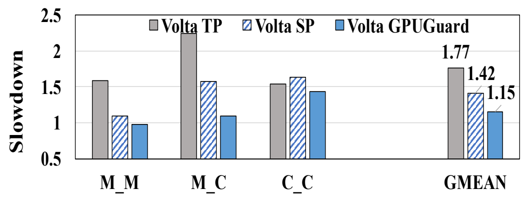
(a)



(b)



(c)



(d)

Figure 6.7: Performance impact of GPUGuard on (a) Fermi; (b) Kepler; (c) Volta; (d) Benign Apps on Volta, grouped by M_M (two memory bound apps), M_C (one memory + one compute bound app) and C_C (two compute bound apps). Measured by normalizing the execution time over intra-SM sharing without protection. (SP: Spatial Partitioning, TP: Temporal Partitioning)

by at least 2.1x, while GPUGuard has significantly lower overhead. Clustered SM partitioning alone (labeled as SP) improves performance over temporal partitioning by 42% on average across three architectures, since it avoids kernel preemption. GPUGuard further reduces the overhead by 18% and 30% for ADD and SFU attacks over spatial partitioning. In our baseline Fermi configuration, the initiation interval of MUL application is as long as 16 cycles, and the longer latency caused by warp folding leads to some performance penalty. However, recent generations of GPUs greatly improved the latency of matrix multiply operation, which is likely to amortize this performance penalty. Overall, GPUGuard provides robust defense across multiple attacks and incurs less than 15% overhead, only when actively defending against an ongoing attack. Considering the mitigation techniques in GPUGuard almost only turned on when there is an attack detected, the performance overhead is much smaller than simply clustering SMs all the time.

In the attack on Atomic primitives, the Trojan kernel chooses to perform Atomic operations or not to encode '1's and '0's. The Spy kernel, on the other hand, will always issue Atomic operations. The back-pressure in memory system leaks whether the Trojan kernel is sending the '1' or '0'. The Spy kernel measures the Atomic instruction latency to decode this fluctuation. While concurrently executing a Trojan and Spy kernel in the same GPU amortizes the memory pressure, sharing the same SM can further reduce the congestion in load and store units. However, GPUGuard and spatial partitioning cannot close global memory channels. Since GPUGuard falls back on temporal partitioning for global memory attacks, GPUGuard's performance is the same as temporal partitioning in Figures 6.7a to 6.7c for Atomic attacks.

Overall, the geometric mean of GPUGuard's performance is 94%, 96%, and 69% faster than of temporal partitioning, and 14%, 22%, and 10% faster than spatial partitioning in Fermi, Kerpler, and Volta architecture, showing that it is possible to benefit from multiprogramming while maintaining protection against covert-channel attacks. Volta architecture has much higher HBM bandwidth, that reduces the number of warps stalled by long latency memory accesses. As we have more ready warps to schedule to execute, the protection schemes see less performance overhead. It's worth to note that, GPUGuard only incurs 8% of performance overhead against a baseline with no protection in Volta. Thus, another key benefit of intra-SM protection is system robustness. A mitigation technique with high performance penalty provides opportunity for Denial of Service attacks. By minimizing the performance slowdown, GPUGuard also minimizes such potential security risks. Moreover both temporal partitioning and spatial partitioning require preempting the running Trojan or Spy kernel. GPUGuard partitions the resources within a core, and does not incur preemption overhead. Finally, GPUGuard triggers defense only when attack is detected, thus has no overhead when there are no attacks detected.

Impact of False Positives. FP cases are rare. Nonetheless, we further study the performance impact of those cases when normal kernels are inaccurately classified as malicious in Volta. As shown in Figure 6.7d, temporal partitioning incurs 77% slowdown, spatial partitioning incurs 42% slowdown, while GPUGuard reduces that to only 15%. GPUGuard benefits the most for kernels that are memory + compute case, when complimentary sharing inside one SM is favored, and the least for compute + compute case, when datapath slicing reduced the opportunity for complimentary compute operation to share pipeline cycles.

6.7.3 Hardware Overhead

GPUGuard requires 24 performance counters for sampling the selected features for threat detection (each 10 bits), once every 1000 cycles. Among those, 10 counters, including the $SP-INT$, $SP-FP$, SFU issued counters, the $SP-FP$, SFU, and LD/ST utilization counters, L1C and L2 accesses and misses counters, are already provided in the modern GPUs [29]. The V100 clock rate is 1.53GHz and the required sampling bandwidth is 45.9MB/s, negligible compared to the 900GB/s off-chip memory bandwidth.

The data collected by the counters are fed into the decision tree classifier. Synthesis results show that the classifier consumes $0.21mW$ per detection with $0.62ns$ latency. Since the classification is not on the performance critical path, the one cycle latency does not affect the overall system performance. The area overhead of the classifier is $0.001mm^2$, which is small compared to the die area.

Table 6.4: Extra hardware overhead per SM at 45nm.

	Latency (ns)	**Power** (mW)	**Area** (mm^2)
RegBuffers	0.21	95.5	0.09
DEMUXs	0.06	0.28	0.002
MUXs	0.06	0.45	0.004
TangramUnit	0.57	1.08	0.001

Tangram security units require 22.5B RAM for keeping track of the security domain IDs and scheduling information. As described in Section 6.5, our datapath slicing design simply folds a warp and has low hardware overhead. Different from a full-blown variable sized warp architecture, proposed by Rogers et al. [188], our design does not require any modification to the interconnection network between the register banks and the data pipeline. To support our datapath slicing, 128B register buffers and four multiplexers/de-multiplexers (32-bit width) are required per SM. The latency, power, and area overheads are broken down in Table 6.4. The area overhead is $0.1mm^2$ per SM. Based on the activity factors collected by the timing simulator, the average power of the added hardware is $8.3mW$ per SM. We extract area and average power of 16 SMs from GPU-Wattch [132], which are $704mm^2$ and $73W$, respectively. GPUGuard results in 0.2% area overhead for GPUs with 16 SMs. The total power consumption was 0.18% of the overall system power.

6.8 Mitigating Side Channel Attacks

Similar to covert channel scenario, in contention based side channel, a malicious application (Spy) accesses to different hardware resources and either measures the access time or its own performance counters to extract some information about concurrent workloads (victim) running on GPU. Due to the large number of active threads, and the relatively small cache structures, it is hard to achieve high-precision prime-probe or similar timing attacks on GPUs. In Chapter 5 we demonstrated that GPU side channels are feasible by aggregate measures of contention through available GPU performance counters. To the best of our knowledge, this work is the only proposed side channel between two concurrent applications on GPU. In this subsection, we intend to evaluate our defense on this contention-based side channel.

We re-implemented the CUDA-CUDA side channel attack in Chapter 5 on GPGPU-Sim. In this attack, a Spy application runs concurrently with a back-propagation workload from Rodinia benchmark and extracts the number of neurons in the input layer of neural network through side channel. We collected runtime per kernel performance counters for Spy application when it is concurrently running with back-propagation algorithm with input layer size varying in the range between 64 and 65536 neurons. We trained a Random Forest classifier with 10-fold cross validation and achieved accuracy of about 70% recovering the input layer size. The performance counter set available on real GPUs through NVIDIA tools are a bit different than those collected on GPGPU-sim during runtime, leading to lower side channel accuracy using the simulator. Since the Spy accesses different hardware resources to create contention, similar to the Spy and Trojan in the detection benchmarks, can be classified correctly as attack by our threshold-based classifier. Once the Spy has been detected as an attack, Tangram will be enabled promptly to separate the malicious Spy from other normal concurrent applications into different security domains. By our intra-SM isolation between two concurrent applications, we observed that the accuracy of attack significantly decreased obtaining essentially a random guess accuracy: Tangram was able to mitigate the attack by isolating contention between victim and spy.

Chapter 7

Microarchitectural Attacks in Integrated CPU-GPU systems

Modern computing platforms are heterogeneous, combining latency (CPU) and throughput (Accelerators) oriented processors to gain both performance and energy efficiency for a variety of computational tasks. These new computing architectures add security vulnerability to the systems and lead to new attacks. GPUs as accelerators/co-processors in such systems can be either the target of attacks themselves, or serve as a vector for launching or amplifying attacks that compromise the main processor or the whole system, bypassing existing mitigations already in place.

In previous chapters, we studied the security of discrete GPUs in heterogeneous systems in terms of microarchitectural covert and side channel attacks and defenses. These attacks have been widely studied on CPUs, as well. All prior attacks create contentions from identical components with identical pathways to a shared resource (e.g., two CPU processes, or two GPU kernels). In this chapter, we study the microarchitectural attacks between two asymmetric components.

In contrast to discrete GPUs which have a dedicated graphics memory, integrated GPUs (iGPUs) are tightly integrated on the same die with the CPU and share resources such as the last level cache and memory subsystem with the CPU. This integration opens up the potential of new attacks that exploit use of common resources to create interference between these components, leading to cross-component micro-architectural attacks. Specifically, in this Chapter, we investigate micro-architectural covert and side channel attacks on integrated heterogeneous systems in which two applications, located on two different components (CPU and iGPU) transfer or extract secret information via shared hardware resources. We develop two instances of cross-component attacks: covert channel in native code (OpenCL) and side channel in JavaScript (WebGL).

These iGPUs are extensively used in portable electronic devices to provide graphics, compute, media, and display capabilities. Understanding microarchitectural vulnerabilities in such environments is essential to the security of these widely used systems. Moreover, iGPUs exemplify a trend to gradually increase the level of heterogeneity in modern computing systems, as further scaling of fabrication technologies allows formerly discrete components to become integrated parts of a system-on-chip, and provides for integration of specialized hardware accelerators for important workloads. Thus, these attacks help illuminate potential threats to general heterogeneous computing systems.

7.1 Integrated CPU-GPU Systems

We target Intel integrated GPUs for our attacks, as the most widely used integrated systems. In this section, we introduce the organization of Intel's integrated GPU systems, to provide background necessary to understand our threat model and attacks.

Starting in 2010, Intel's CPUs have iGPU incorporated on the same die with the conventional CPU, to support increasingly multi-media heavy workloads without the need for a separate (bulky and power hungry) GPU. This GPU support has continued to evolve with every generation providing more performance and features; for example the current generation of Intel Graphics (Iris Plus on Gen11 Intel Graphics Technology [108]) offers up to 64 execution units (similar to CUDA cores in Nvidia terminology) and at the highest end, over 1 Teraflops of GPU performance. Thus, modern processors already use complex System-on-Chip (*SoC*) designs.

The architectural features and programming interface for the iGPU are similar to those of discrete GPUs in many aspects. For general purpose computing on integrated GPUs, the programmer uses OpenCL [31] (equivalent to CUDA programming model on Nvidia discrete GPUs [24]). Based on the application, programmers launch the required number of threads that are grouped together into work groups (similar to thread blocks in Nvidia terminology). Work groups are divided into groups of threads executing Single Instruction Multiple Data (*SIMD*) style in lock step manner (called wavefronts, analogous to warps in Nvidia terminology). In the context of Nvidia GPUs, this group of threads is called warp (typically, 32 in number). In integrated GPUs the SIMD width is variable; it changes depending on the register requirements of the kernel.

3mm

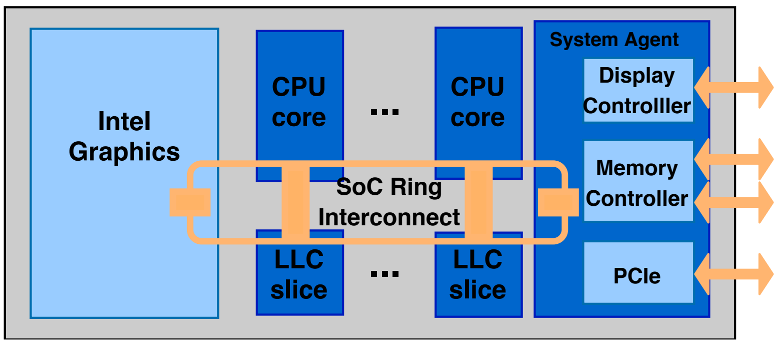


Figure 7.1: Intel SoC architecture

iGPUs reside on the same chip and share the system RAM and connect to the same memory hierarchy as the CPU (typically at the LLC level). For both CPU cores and for GPU, LLC seeks to reduce apparent latency to system DRAM and to provide higher effective bandwidth. Figure 7.1 shows the architecture of an Intel SoC processor, integrating several CPU cores and an iGPU [108]. The iGPU is connected with CPUs and the rest of the system through a ring interconnect: a 32 byte wide bidirectional data bus. The GPU shares the Last Level Cache (*LLC*) with the CPU, which much like the CPU, serves as the last level of the GPUs cache hierarchy. The whole LLC is accessible by the GPU through the ring interconnect with a typical implementation of address ranges hashing to different slices of the LLC. The GPU and CPU can access the LLC simultaneously. However, there is an impact on the access latency if the GPU and CPU contend for accessing, due to factors such as delays in accessing the bus and access limitations on the LLC ports. The GPU and CPU share other components such as the display controller, the PCIe controller, the optional eDRAM controller and the memory controller.

The architecture of the iGPU is shown in Figure 7.2. The computational units in the integrated GPUs are called Execution Units *(EU)* (similar to CUDA core in Nvidia terminology). A group of 8 EUs (analogous to CUDA cores) are consolidated into a single unit which is called a *Subslice* (similar to SM in Nvidia terminology) and typically 3 subslices create a *Slice*. The number of slices varies with the particular SoC model even within the same generation, as the slices are designed in a modular fashion allowing different GPU configurations to be created. Experimentally, we discovered that multiple work groups are allocated to different subslices in a round robin manner. The global thread dispatcher launches the work groups to different subslices. A single SIMD width equivalent number of threads in a single subslice is launched to EUs in a round robin manner as well. A fixed functional pipeline in the slice is dedicated for graphics processing.

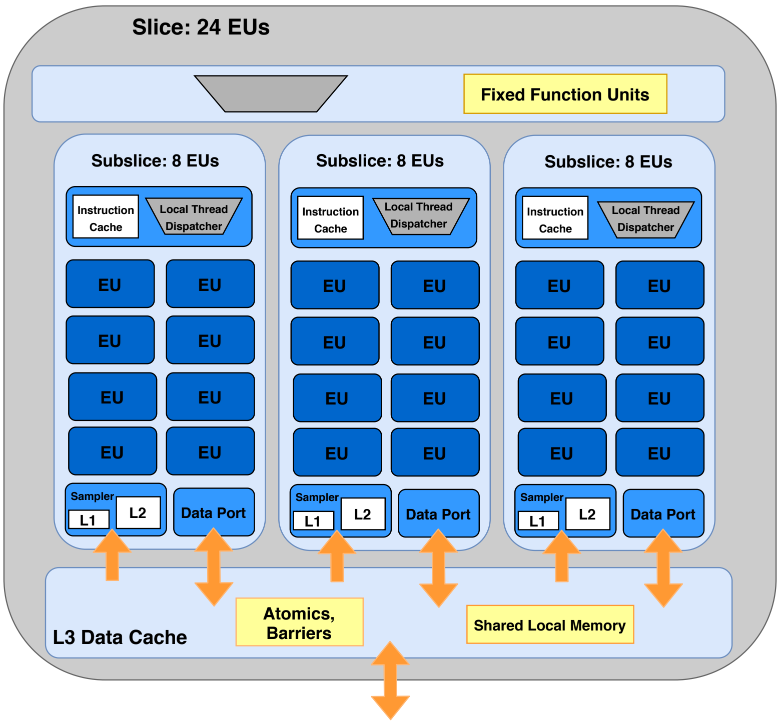


Figure 7.2: Intel integrated GPU architecture

The iGPU uses three levels of cache (in addition to the LLC). The first two levels, L1 and L2, are called sampler caches and are used solely for graphics. The third level cache, L3, is universal and can be used for both graphics and computational applications. The L3 cache is common to all the subslices in a single slice. The L3 cache fabric in different slices is also interconnected, giving a consolidated L3 architecture shared by all the EUs in all slices. In each slice, there is also a shared local memory (SLM), a structure within the L3 complex that supports programmer managed data sharing among threads within the same work group [108].

7.2 Cross-Component Covert and Side Channels

In this chapter, we explore microarchitectural attacks from one component to another within a heterogeneous system, more specifically, CPU-GPU systems. First, we investigate the possibility of covert channels, then we show how we can use the knowledge from cross-component covert channel to develop attacks in more dangerous setup, which is remote side channel attacks in web browsers. In order to develop this new type of channels , we have to solve a number of new challenges, including:

- Synchronization across heterogeneous components with frequency disparity
- Reconciling asymmetric computational models and memory hierarchies
- Creating reliable fine-grained timing mechanisms

7.2.1 Covert Channel in Native Code

In collaboration with Sankha Dutta[1] and Pacific Northwest National Lab., we developed two covert channels between CPU and iGPU: a Prime+Probe channel targeting the LLC, and a contention based channel exploiting contention on the shared access pathway to the LLC. Next, we briefly present our key contributions and results in this work.

Although at a high level this attack strategy is similar to other covert channel attacks, there are a number of unique challenges that occur when we try to implement the channel between the CPU and GPU. The challenges generally arise from the heterogeneous nature of the computational models on the two components, as well as the different memory hierarchies they have before the shared LLC. We overview these challenges next.

- **Absence of a GPU timer:** Prime+Probe attacks rely on the ability to time the difference between a cache hit and a cache miss to implement communication. Usually, a user level hardware counter is available on the system to measure the access latency. While this is true on the CPU side, unfortunately OpenCL on iGPUs does not provide any such means to the programmer. We leverage GPU parallelism and hardware shared local memory to build the custom timer. The details are provided in the next section.

[1]sdutt004@ucr.edu

- **Reverse Engineering the LLC viewed from the GPU** To be able to target specific sets in the LLC for covert communication, we require the knowledge of the physical addresses mapping to cache addresses from both CPU and GPU side (the LLC is physically indexed). Modern GPUs come with their own page tables and paging mechanisms. We use the mechanism of shared virtual memory [107] and zero copy memory to maintain the same physical and virtual addresses across the device. When a CPU process initializes and launches the GPU kernel, the CPU page table is shared with the GPU in this scenario. This sharing allows us to reverse engineer the cache from the CPU using established techniques [238] and use these results on the GPU.

- **Reverse engineering the GPU cache hierarchy:** While the Intel CPU cache hierarchy is well understood, the GPU cache hierarchy details are not published. It is critical to understand the cache hierarchy since it determines how memory accesses spill over to the LLC where the covert channel is being implemented. Since L1 and L2 cache are not used by OpenCL, we need to reverse engineer the GPU L3 to understand how to control the memory references that are evicted from it. First, we needed to understand whether the LLC is inclusive of the L3 which would make simplify eviction from the L3 from the CPU side. However, we discover that it is not inclusive, which requires us to understand the L3 in detail in order to control evictions from it.

- **Optimization around heterogeneous components:** Since the spy and the trojan use completely different computation models operating at substantially different clock rates, determining how to best implement the channel to improve bandwidth and reduce noise is tricky. For example, the CPU we are using operates at the 4.2 GHz and GPU operates at 1.1 GHz. This frequency imbalance imposes an unique challenge as the prime and probe would take place at different frequency. We also take advantage of GPU parallelism by launching multiple threads to overcome this frequency imbalance.

By addressing all the challenges, we developed two reliable covert channels, a shared LLC cache based channel and a contention based channel targeting the ring bus connecting the CPU and iGPU. The LLC based channel achieves a bandwidth of 120 kbps with a low error rate of 2%, while the contention based channel delivers up to 400 kbps with a 0.8% error rate.

7.2.2 Side Channel in JavaScript

In previous section, we showed possibility of cross-component covert channel in native environment (OpenCL applications). The presence of a covert channel is a prerequisite for side-channel attacks. The integrated GPU is available through APIs such as WebGL for remote JavaScript programs making this threat vector extremely dangerous. In this section, we investigate remote side channel attack in JavaScript from iGPU, spying on CPU.

WebGL: WebGL (Web Graphics Library) is a JavaScript API to accelerate interactive 3D and 2D graphics within any compatible web browser without the use of plug-ins. WebGL API closely conforms to OpenGL ES 2.0 and can take advantage of hardware graphics acceleration provided by the user's device [38]. Based on [40], 97% of computing devices support WebGL version 1.0 and 54% support WebGL version 2.0 (conforms to OpenGL ES 3.1 API).

WebGL Compute Shader: Although WebGL is primarily designed to improve the rendering performance of web browsers, it provides an extension called Compute Shader [45] which supports the use of general-purpose compute functionality within the web browsers. Currently, this extension can be enabled by some flags in experimental versions of Google Chrome browser (Chrome Canary). We use this interface to launch our GPU-based side channel within the web browser.

WebGPU: GPU based general purpose computation on the web, is being more widely developed through other interfaces. WebGPU [46, 57] is an emerging Javascript API which provides access to the hardware accelerated graphics and computing capabilities on the web. It is being developed by all major browser vendors. WebGPU uses its own shading language called WGSL. Currently, WebGPU can be accessed in Safari as an experimental feature.

As discussed in Chapter 3, several work studied microarchitectural attacks in browsers, with the use of JavaScript timers or crafting customized high resolution timers. However, major browser vendors disabled all the interfaces on which the timers were built. We believe that our WebGL based attack, launched on GPU will bypass all mitigations which are in place to prevent remote JavaScript side channel attacks.

The main steps of our PRIME+PROBE side channel attack from iGPU to CPU are as follows:

- Building a high resolution timer on the iGPU to track the LLC hits and misses to conduct a timing side channel, since there is no clock() function or similar instructions to measure the time on compute shader.

- Identify the eviction sets on the GPU corresponding to interesting cache regions on CPU. The eviction set is a set of physical addresses that mapped onto the same cache set [207]. Once the attacker acquires the addresses that are in the same cache set, she can monitor the victim's activity by manipulating the cache set (PRIME+PROBE).

- Finding the interesting cache regions from the CPU side to attack a victim application. The attacker needs to induce the victim to perform an action, and then examine which cache sets were touched by this action. Machine learning methods can be employed to monitor the cache behaviour, as presented in [143, 241].

- Conduct the PRIME+PROBE on the eviction set and synchronize the spy with the victim, by using the iGPU parallelism. A possible attack is tracking user activity on the CPU, either within a native or a web-based application.

In addition to challenges unique to asymmetric heterogeneous systems that we addressed in previous section for the covert channel, developing a last-level cache attack in JavaScript brings new challenges into the scenario, making the attack development quite challenging. Two of the most important challenges are crafting the high resolution timer in WebGL and identifying the LLC eviction set without any system or kernel support on JavaScript. Next, we overview our approach to overcome these challenges:

Crafting a high resolution Timer on iGPU: Access to a high-resolution timer is essential to the ability to carry out cache based covert channels; without it we are unable to discriminate a cache hit from a cache miss, which is the primary phenomena used in the communication. Therefore, we need to come up with an alternative approach to measure the access latency within the GPU application. We followed the same idea from our OpenCL based covert channel to create the timer in WebGL compute shader.

We leverage GPU parallelism and hardware shared local memory to build the custom timer. Shared local memory in Intel based iGPUs is a memory structure, shared across all EUs in a subslice. 64 Kbytes of shared local memory is available per subslice and is private to all the threads from a single work-group. We launch a work-group for which certain number of threads are used to conduct the attack and the rest of the threads are used to increment a counter value stored in shared memory. The threads that are responsible for carrying out the attack read the shared value as timestamps before and after the access to measure the access time (the principle of this technique was used in CPU attacks on the ARM where the hardware time is not available in user mode [139] and also in JavaScript between two web workers [93, 190]). Rather than shared buffers between two threads in these work, we build our timer on GPU hardware shared memory. Shared memory uses a separate data path than that used for accessing L3, which makes sure that there is no resource contention that can lead to erratic counter updates. We use atomic addition to increment the counter value to ensure that the variable is accessed and incremented properly. Due to branch divergence within the wavefronts (SIMD width of threads), the execution of two groups of threads in a single wavefront gets serialized. So the number of threads that are used for counter increment start at a wavefront boundary till the end of the workgroup.

To conduct a PRIME+PROBE attack, the attacker needs to distinguish 3 levels of access time, *i.e.* system memory, LLC and L3. The average measured values while accessing the three level of hierarchy using our crafted timer are 510, 390, and 215. Although the timer works perfectly to distinguish the cache hits or misses, as a standalone, when it is embedded in the real attack code, there is a high level of noise, which we believe it mostly is stemmed from the instability of our experiment environment.

Creating an LLC eviction set on the iGPU side: Although the reverse engineering results from our cross component covert channel in native code, enable us to understand the memory hierarchy of both CPU and GPU, it can not be directly used to identify the eviction sets in JavaScript. In JavaScript, there is no access to system or kernel level support, for example large pages(2MB) in which the lower 21 bits of the physical and virtual addresses are identical, and by the additional use of an iterative algorithm, the unknown upper (slice) bits of the cache set index are resolved. In addition, JavaScript has no notion of pointers, so even the virtual addresses of our own variables are unknown to us. This makes it very difficult to provide a deterministic mapping of memory address to cache sets.

To overcome these challenges, we need to use heuristic algorithm to identify the eviction sets on LLC to conduct PRIME+PROBE attack. We implemented the same methodology as presented in [172] in compute shader for creating the eviction set. To avoid thread level parallelism on GPU and ensure the in-order execution of critical code part, we used some dummy dependent instructions to the critical memory access instruction and making the timing measurement code artificially dependent on the eviction set iteration code. We accessed the eviction set addresses in the form of a linked list and we randomly permute the order of elements in the eviction set.

Due to instability of our experiment environment (compute shader is not released yet, we had to use the experimental version on Chrome Canary) and also the timing jitter in compute shader, we were not able to identify the exact eviction sets on the GPU side for some interesting cache regions on CPU side (the victim application accesses). As our future work, we will implement this attack on a stable and final version of GPU based general purpose computation interface (WebGPU), when it is released in major web browser.

Depending on the success of threats in such environments, we will also study the development of both hardware and software mitigations to these attacks.

Chapter 8

Conclusion and Future Work

In this dissertation, we demonstrated for the first time that GPUs are vulnerable to microarchitectural covert and side channel attacks. Over last several years, such attacks have been demonstrated on a variety of CPU microarchitectural structures such as the caches and branch predictors. Graphics Processing Units (GPU) have become an integral part of modern computing platforms, present from end-user devices to large scale high performance clusters and data centers. It is critical to understand whether this type of vulnerability can manifest in GPUs as well.

At the same time, the success of GPUs is part of a trend of increasingly heterogeneous computing systems that integrate a combination of latency oriented processors (CPU), throughput oriented accelerators (GPUs), and potentially application specific or reconfirgurable accelerators to gain both performance and energy efficiency for a variety of computational tasks. Thus, our work is also a first step towards recognizing that microarchitectural attacks commonly studied in the context of CPUs are also possible on other components within heterogeneous systems.

In this chapter, I will summarize the primary conclusions of the dissertation and discuss potential follow-up future work.

8.1 Security of Discrete GPUs

This dissertation explored the security of GPUs in end-user devices and computational clouds where multiple applications are running simultaneously. We demonstrated the vulnerability of GPUs to both covert and side channel attacks and proposed secure architecture for future GPUs to prevent these attacks.

Covert Channel Attacks: First, we reverse engineered the scheduling of multiple applications on GPUs and based on co-location options, we identified the shared hardware resources and constructed high quality and bandwidth microarchitectural covert channels on three generations of Nvidia GPGPUs through different intra-SM and inter-SM shared resources: cache, global memory and functional units. We optimized the channel by synchronizing the communication and remove the noise from other concurrent applications. We achieved the error-free bandwidth of over 4Mbps on Nvidia Kepler GPUs (Chapter 4).

Side Channel Attacks: Then we demonstrated side channel attacks that can be launched within both the computational and graphics software stacks, as well as across them (Chapter 5). Both computational and graphics applications process sensitive data and these attacks represent a novel and dangerous attack vector. GPUs have a radically different execution model from CPUs, with massive parallelism and internal hardware schedulers; this makes attacking them substantially different from CPU attacks. We reverse engineered GPU scheduling in both the graphics and the computational stacks of Nvidia GPUs, and characterized the co-location opportunities for concurrent applications. Armed with the co-location knowledge, we explored available leakage vectors in each situation and demonstrated a family of end-to-end attacks where the spy can interleave execution with the victim to extract side channel information.

The first attack implements website fingerprinting through GPU memory utilization API or GPU performance counters. The spy probes the memory API to obtain a trace of the memory allocation operations carried out by the victim as it renders different objects on a webpage visited by the user. The second attack tracks user activities as they interact with a website or type characters on a keyboard. We can accurately track re-rendering events on GPU and measure the timing of keystrokes as they type characters in a textbox (e.g., a password box in Facebook website), making it possible to carry out keystroke timing analysis to infer the characters being typed by the user. The proposed side channel contributes a new vector for these well-known attacks through the GPU where state-of-the-art mitigations are ineffective.

A third attack uses a CUDA spy on computational stack of GPU to infer the internal structure of a neural network application, which is often a trade secret (a model extraction attack). When a victim is running concurrently on the GPU and utilizing the shared resources, depending on number of input layer size or other parameters of neural network model, the intensity and pattern of contention on the cache, memory and functional units is different over time, which we demonstrated creates measurable leakage in the spy enabling the model to be extracted.

GPUs are now offered as a resource in virtualized cloud environments including Google cloud, Amazon AWS cloud and Microsoft Azure. Despite their improving performance and increasing range of applications, the security vulnerabilities of GPUs in such multi-tenancy environments have not been addressed. Using our proposed computational based side channel, a malicious VM can now spy on other applications that share a GPU (even if they are not co-located on a CPU), and bypass information isolation boundaries.

As more sensitive applications are accelerated by GPUs, we will need to investigate their security implications. There is a trend to accelerate DNN using GPUs, publicly accessible in major cloud platforms. In our future work, we will investigate model extraction attack to steal all the hyper-parameters of a victim user's model and expose the internal structure of deep learning model which is a critical trade secret, violating the privacy of cloud clients. In addition, Both software and hardware are increasingly evolving to support the trend. As an example, Nvidia released some software frameworks, such as cuDNN library [19] and there are specialized cores in new generations of Nvidia, called tensor core to operate the DNN operations more efficiently [20]. Leveraging the knowledge of reverse engineering and attack implementation in our work, we will study new microarchitectural attacks that can be established through these optimization infrastructures, exposing sensitive information on the clouds.

Given the growing number of accelerators being used in safety-critical applications that require secure execution, these attacks stress the need to address security implications in GPU hardware design, GPU sharing models and task scheduling on the clouds.

Mitigations: To mitigate contention based covert and side channel attacks on GPUs, we proposed GPUGuard, a dynamic detection and defense mechanism (Chapter 6). The detection uses a decision tree based design that is able to accurately detect covert and side-channel attacks (100% sensitivity in our experiments). The detection algorithm feeds the classiication results to Tangram, a GPU-speciic covert channel elimination scheme. Tangram uses a combination of warp folding, pipeline slicing, and cache remapping mechanisms to close the channels with 8%-23% performance overhead when there are active attacks, and 15% for normal benchmarks categorized as attacks with only a small (8.5%) false positive rate. In all other cases GPUGuard pays nearly zero performance overhead. Our proposed GPUDuard shows that it is possible to gain substantial performance from executing concurrent kernels on a single SM while securing GPUs against these attacks.

8.2 Cross-Component Attacks in Integrated Heterogeneous Systems

As systems are increasingly heterogeneous, integrating CPUs with accelerators such as GPUs on the same die and providing the opportunities for resource sharing, it is important to understand the threat of microarchitectural attacks on these emerging systems. New computing models are a combination of processors augmented by a variety of interconnect and memory architectures. Accelerators as co-processors in such systems can be either the target of attacks themselves , or serve as a vector for launching or amplifying attacks that compromise the main processor or the whole system.

In addition to the study the vulnerabilities of standalone GPUs (as we demonstrated in Chapters 4, 5, and 6), we explored attacks that spill over from the GPU to CPU or vice versa or more generally, from one component to another within a heterogeneous system (Chapter 7).

First, we developed covert channels (secret communication channels that exploit contention) on integrated heterogeneous systems in which two malicious applications, located on two different components (CPU and iGPU) transfer secret information via shared hardware resources. This is the first asymmetric covert channel where the two sides of the channel are completely different in their computational model, organization, and view of the shared resource. Then, we explored an extremely dangerous threat vector in the integrated CPU-GPU systems. GPU accelerated general purpose computation is being developed in major web browsers. By leveraging this interface, we studied the possibility of GPU based side channel attacks in JavaScript to spy on the CPU through the shared last level cache.

Although we demonstrated two instances of cross-component attacks (specifically, an integrated GPU and CPU) in heterogeneous systems, the threat model can be extended to include any other accelerator or components, sharing resources with CPUs. Having experience with these channels improves our understanding of the threats posed of microarchitectural attacks beyond a single component which is a threat model increasing in importance as we move increasingly towards heterogeneous computing platforms.

In our future work, we will investigate all threat surfaces in heterogeneous system architectures in end user devices and computational clouds, as well as high performance computing (HPC) clusters. Each of these computing environments has a unique architecture to connect dierent components based on performance and eciency of the system, leading to special security concerns and challenges.

We will study characterizing the threat surface of attacks that will arise in such computing environments; including denial of service attacks, cross component side channels and fault injection attacks (e.g. Rowhammer). With the assessment of the threats discussed above, we will work on extending the CPU-based hardware isolation primitives to the heterogeneous accelerator components, including GPUs and FPGAs.

Bibliography

[1] Deep Learning on GPUs. http://on-demand.gputechconf.com/gtc/2015/webinar/deep-learning-course/intro-to-deep-learning.pdf.

[2] GPU cloud rendering. http://www.nvidia.com/object/gpu-cloud-rendering.html.

[3] Preemption improved: Fine-grained preemption for time-critical tasks. https://www.anandtech.com/show/10325/the-nvidia-geforce-gtx-1080-and-1070-founders-edition-review/10.

[4] Top 500 supercomputing sites. http://www.top500.org/statistics/list/.

[5] Training with multiple GPUs using model parallelism. $https : //mxnet.incubator.apache.org/how_{t}o/model_{p}arallel_{l}stm.html.$

[6] Network Covert Channels:Subversive Secrecy. `https://www.sans.org/reading-room/whitepapers/covert/network-covert-channels-subversive-secrecy-1660`, 2006.

[7] NVIDIA CUDA compute unified device architecture - programming guide, 2008. http://developer.download.nvidia.com.

[8] Whitepaper: NVIDIA's Next Generation CUDA™ Compute Architecture: Fermi™. Technical report, NVIDIA, 2009.

[9] AMD graphics cores next (GCN) architecture. Technical report, AMD, 2012.

[10] Whitepaper: NVIDIA's Next Generation CUDA™ Compute Architecture: Kepler™ GK110. Technical report, NVIDIA, 2012.

[11] The freepdk process design kit. `https://www.eda.ncsu.edu/wiki/FreePDK`, 2014.

[12] Whitepaper: NVIDIA GeForce GTX980. Technical report, NVIDIA, 2014.

[13] Chromium: window.performance.now does not support sub-millisecond precision on windows, 2015. `https://bugs.chromium.org/p/chromium/issues/detail?id=158234#c110`.

[14] Reduce resolution of performance.now, 2015. `https://bugs.chromium.org/p/chromium/issues/detail?id=506723`.

[15] Grid virtual GPU. Technical report, Nvidia, 2016.

[16] Whitepaper: AMD multiuser GPU: hardware-enabled GPU virtualization for a true workstation experience. Technical report, AMD, 2016.

[17] Whitepaper: NVIDIA GeForce GTX1080-Gaming Perfected. Technical report, NVIDIA, 2016.

[18] window.performance.now does not support sub-millisecond precision on windows, 2016. `https://bugs.chromium.org/p/chromium/issues/detail?id=158234#c110`.

[19] NVIDIA cuDNN. Technical report, NVIDIA, 2017. https://developer.nvidia.com/cudnn.

[20] NVIDIA Tesla V100 GPU Architecture. Technical report, NVIDIA, 2017.

[21] Alexa Top Sites, 2018. `https://www.alexa.com/topsites`.

[22] ARB Extenstion, Khronos Group, 2018. `https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_shader_clock.txt`.

[23] CUDA-enabled GPUs, Nvidia, 2018. `https://developer.nvidia.com/cuda-gpus`.

[24] CUDA, Nvidia, 2018. `https://developer.nvidia.com/cuda-zone/`.

[25] GPU Architecture Roadmap, The Chromium Projects, 2018. `https://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome/gpu-architecture-roadmap`.

[26] GPU Cloud Computing, Nvidia, 2018. `https://www.nvidia.com/en-us/data-center/gpu-cloud-computing/`.

[27] Linux Graphics Debugger, Nvidia, 2018. `https://developer.nvidia.com/linux-graphics-debugger`.

[28] Mitigations landing for new class of timing attack, 2018. `https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/`.

[29] NVIDIA Profiler User's Guide, 2018. `http://docs.nvidia.com/cuda/profiler-users-guide/index.html`.

[30] NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110, 2018. `https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`.

[31] OpenCL Overview, Khronos Group, 2018. `https://www.khronos.org/opencl/`.

[32] OpenGL Extension, Intel, 2018. `https://www.khronos.org/registry/OpenGL/extensions/INTEL/\INTEL_performance_query.txt`.

[33] OpenGL Extenstion, Khronos Group, 2018. `https://www.khronos.org/registry/OpenGL/extensions/NVX/NVX_gpu_memory_info.txt`.

[34] OpenGL Extenstion, Khronos Group, 2018. `https://www.khronos.org/registry/OpenGL/extensions/NV/NV_shader_thread_group.txt`.

[35] OpenGL Overview, Khronos Group, 2018. `https://www.khronos.org/opengl/`.

[36] Tegra Graphics Debugger, Nvidia, 2018. https://developer.nvidia.com/tegra-graphics-debugger.

[37] Vulkan Overview, Khronos Group, 2018. https://www.khronos.org/vulkan/.

[38] WebGL Overview, Khronos Group, 2018. https://www.khronos.org/webgl/.

[39] WebGL Security, Khronos Group, 2018. https://www.khronos.org/webgl/security/.

[40] WebGL Statistics, 2018. http://webglstats.com/.

[41] NVIDIA CUPTI metric api, 2019. https://docs.nvidia.com/cupti/Cupti/r$_main.htmlmetrics-reference-7x$.

[42] Nvidia Security Notice, 2019. https://nvidia.custhelp.com/app/answers/detail/a_id/4738.

[43] OpenGL ES, Khronos Group, 2020. https://www.khronos.org/opengles/.

[44] performance.now, 2020. https://developer.mozilla.org/en-US/docs/Web/API/Performance/now.

[45] WebGL2 Compute Shader, 2020. https://www.khronos.org/registry/webgl/specs/latest/2.0-compute/.

[46] Webgpu, 2020.

[47] Mohammad Abdel-Majeed, Daniel Wong, Justin Kuang, and Murali Annavaram. Origami: Folding warps for energy efficient GPUs. In *Proceedings of the International Conference on Supercomputing (ICS)*, June 2016.

[48] Jacob T. Adriaens, Katherine Compton, Nam Sung Kim, and Michael J. Schulte. The case for gpgpu spatial multitasking. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pages 1–12, February 2012.

[49] P. Aguilera, K. Morrow, and N. S. Kim. Fair share: Allocation of GPU resources for both performance and fairness. In *Proceedings of the International Conference of Computer Design (ICCD)*, Oct. 2014.

[50] Mansaf Alam and Shuchi Sethi. Detection of information leakage in cloud. volume abs/1504.03539, 2015.

[51] Amazon AWS. Amazon Elastic Graphics, 2019. https://aws.amazon.com/ec2/Elastic-GPUs/.

[52] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse Engel, Linxi Fan, Christopher Fougner, Tony Han, Awni Hannun, Billy Jun, Patrick LeGresley, Libby Lin, and Zhenyao Zhu. Deep speech 2: End-to-end speech recognition in english and mandarin. In *arXiv preprint arXiv:1512.02595*, 2015.

[53] Ahmed Osama Fathy Atya, Zhiyun Qian, Srikanth V. Krishnamurthy, Thomas La Porta, Patrick McDaniel, and Lisa Marvel. Malicious co-residency on the cloud: Attacks and defense. In *IEEE Conference on Computer Communications*, INFOCOM'17, pages 1–9, 2017.

[54] Paramvir Bahl, Ranveer Chandra, Thomas Moscibroda, Rohan Murty, and Matt Welsh. White space networking with wi-fi like connectivity. In *ACM SIGCOMM Computer Communication Review*, volume 39 of *SIGCOMM '09*, pages 27–38, October 2009.

[55] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2009.

[56] Davide Balzarotti, Roberto [Di Pietro], and Antonio Villani. The impact of gpu-assisted malware on memory forensics: A case study. *Digital Investigation*, 14:S16 – S24, 2015. The Proceedings of the Fifteenth Annual DFRWS Conference.

[57] François Beaufort. Get started with gpu compute on the web, 2020.

[58] Michela Becchi, Kittisak Sajjapongse, Ian Graves, Adam Procter, Vignesh Ravi, and Srimat Chakradhar. A virtual memory based runtime to support multi-tenancy in clusters with gpus. In *21st international symposium on High-Performance Parallel and Distributed Computing*, HPDC'12, pages 97–108, Delft, The Netherlands, June 2012.

[59] Andrea Di Biagio, Alessandro Barenghi, Giovanni Agosta, and Gerardo Pelosi. Design of a parallel aes for graphic hardware using the cuda framework. In *IEEE International Symposium on Parallel & Distributed Processing*, IPDPS'09, Rome Italy, May 2009. IEEE.

[60] Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 621–628, New York, NY, USA, 2007. ACM.

[61] Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, pages 667–684, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[62] Yinzhi Cao, Song Li, and Erik Wijmans. (cross-)browser fingerprinting via os and hardware level features. In *NDSS*, 2017.

[63] P. Chairunnanda, N. Pham, and U. Hengartner. Privacy: Gone with the typing! identifying web users by their typing patterns. In *IEEE International Conference on Privacy, Security, Risk and Trust*, pages 974–980, October 2011.

[64] 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 *Proceedings of the International Symposium on Workload Characterization (IISWC)*, Oct. 2009.

[65] Jie Chen and Guru Venkataramani. Cc-hunter: Uncovering covert timing channels on shared processor hardware. In *47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO'14, pages 216–228, Cambridge UK, December 2014. IEEE.

[66] Xuhao Chen, Li-Wen Chang, Christopher I. Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. Adaptive cache management for energy-efficient GPU computing. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, Dec. 2014.

[67] Niket K. Choudhary, Salil V. Wadhavkar, Tanmay A. Shah, Hiran Mayukh, Jayneel Gandhi, Brandon H. Dwiel, Sandeep Navada, Hashem H. Najaf-abadi, and Eric Rotenberg. FabScalar: composing synthesizable RTL designs of arbitrary cores within a canonical superscalar template. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2011.

[68] Alex Christensen. Reduce resolution of performance.now, 2015. `https://bugs.webkit.org/show_bug.cgi?id=146531`.

[69] N. Clarke and S. Furnell. Authenticating mobile phone users using keystroke analysis. *International Journal on Information Security*, 6, December 2006.

[70] Hongwen Dai, Zhen Lin, Chao Li, Chen Zhao, Fei Wang, Nanning Zheng, and Huiyang Zhou. Accelerate GPU concurrent kernel execution by mitigating memory pipeline stalls. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018.

[71] Janis Danisevskis, Marta Piekarska, and Jean-Pierre Seifert. Dark side of the shader: Mobile gpu-aided malware delivery. In Hyang-Sook Lee and Dong-Guk Han, editors, *Information Security and Cryptology – ICISC 2013*, pages 483–495, Cham, 2014. Springer International Publishing.

[72] 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 International Symposium on Computer Architecture (ISCA)*, June 2013.

[73] Renan Correa Detomini, Renata Spolon Lobato, Roberta Spolon, and Marcos Antonio Cavenaghi. Using gpu to exploit parallelism on cryptography. In *6th Iberian Conference on Information Systems and Technologies*, CISTI'11, Chaves Portugal, June 2011. IEEE.

[74] Bang Di, Jianhua Sun, and Hao Chen. A study of overflow vulnerabilities on gpus. In Guang R. Gao, Depei Qian, Xinbo Gao, Barbara Chapman, and Wenguang Chen, editors, *Network and Parallel Computing*, pages 103–115, Cham, 2016. Springer International Publishing.

[75] Khaled M. Diab, M. Mustafa Rafique, and Mohamed Hefeeda. Dynamic sharing of gpus in cloud systems. In *IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW, pages 947–954, Cambride, MA, USA, 2013. May, IEEE.

[76] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization*, 8(4), 2012.

[77] William Enck, Peter Gilbert, Seungyeop Han, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(5), 2014.

[78] Dag Arne Osvik Eran Tromer and Adi Shamir. Efficient cache attacks on aes, and countermeasures. In *Journal of Cryptology*, pages 667–684, 2009.

[79] C. Erb, M. Collins, and J. L. Greathouse. Dynamic buffer overflow detection for gpgpus. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 61–73, 2017.

[80] Dmitry Evtyushkin and Dmitry Ponomarev. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *CCS*, 2016.

[81] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *Proceedings of the International Symposium on Microarchitecture(MICRO)*, 2016.

[82] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Understanding and mitigating covert channels through branch predictors. *ACM Transactions on Architecture and Code Optimization*, 13(1):10, 2016.

[83] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, 2018.

[84] I. Ewell. Disable timestamps in webgl, 2017. `https://codereview.chromium.org/1800383002`.

[85] Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, CCS '00, pages 25–32, New York, NY, USA, 2000. ACM.

[86] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand pwning unit: Accelerating microarchitectural attacks with the gpu. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 357–372, 2018.

[87] Yiwen Gao, Hailong Zhang, Wei Cheng, Yongbin Zhou, and Yuchen Cao. Electro-magnetic analysis of gpu-based aes implementation. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, New York, NY, USA, 2018. Association for Computing Machinery.

[88] Yiwen Gao, Yongbin Zhou, and Wei Cheng. How does strict parallelism affect security? a case study on the side-channel attacks against gpu-based bitsliced aes implementation. *IACR Cryptol. ePrint Arch.*, 2018:1080, 2018.

[89] Google Cloud Platform. Cloud GPUs, 2019. https://cloud.google.com/gpu/.

[90] Thomas R Eisenbarth Gorka Irazoqui and Berk Sunar profile. Cross processor cache attacks. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 353–364, 2016.

[91] Prashant Goswami, Philipp Schlegel, Barbara Solenthaler, and Renato Pajarola. Interactive sph simulation and rendering on the gpu. In *Proceedings of the International Symposium on Computer Animation (SCA)*, July 2010.

[92] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. In *arXiv preprint arXiv:1706.02677*, 2017.

[93] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. Aslr on the line: Practical cache attacks on the mmu. In *NDSS*, 2017.

[94] Joseph Gravellier, Jean-Max Dutertre, Yannick Teglia, Philippe Loubet Moundi, and Francis Olivier. Remote side-channel attacks on heterogeneous soc. In Sonia Belaïd and Tim Güneysu, editors, *Smart Card Research and Advanced Applications*, pages 109–125, Cham, 2020. Springer International Publishing.

[95] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron. Fine-grained resource sharing for concurrent GPGPU kernels. In *Proceedings of the USENIX Conference on Hot Topics in Parallelism (HotPar)*, June 2012.

[96] Daniel Gruss, David Bidner, and Stefan Mangard. Practical memory deduplication attacks in sandboxed javascript. In Günther Pernul, Peter Y A Ryan, and Edgar Weippl, editors, *Computer Security – ESORICS 2015*, pages 108–122, Cham, 2015. Springer International Publishing.

[97] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In *13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'16, pages 279–299. June, 2016.

[98] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912, Washington, D.C., August 2015. USENIX Association.

[99] Berk Gulmezoglu, Andreas Zankl, Thomas Eisenbarth, and Berk Sunar. Perfweb: How to violate web privacy with hardware performance events. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *Computer Security – ESORICS 2017*, pages 80–97, Cham, 2017. Springer International Publishing.

[100] Akhila Gundu, Gita Sreekumar, Ali Shafiee, Seth Pugsley, Hardik Jain, Rajeev Balasubramonian, and Mohit Tiwari. Memory bandwidth reservation in the cloud to avoid information leakage in the memory controller. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, June 2014.

[101] Marc Hamilton. Keynote: The GPU accelerated data center. In *GPU Technology Conference*, Aug. 2015.

[102] Ari B. Hayes, Lingda Li, Mohammad Hedayati, Jiahuan He, Eddy Z. Zhang, and Kai Shen. GPU taint tracking. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 209–220, Santa Clara, CA, July 2017. USENIX Association.

[103] Jamie Hayes and George Danezis. k-fingerprinting: A robust scalable website fingerprinting technique. In *USENIX Security Symposium*, pages 1187–1203, 2016.

[104] W. HE, W. Zhang, S. Sinha, and S. Das. igpu leak: An information leakage vulnerability on intel integrated gpu. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 56–61, 2020.

[105] HSA Foundation. Heterogeneous system architecture (HSA): Architecture and algorithms. In *Proceedings of the International Symposium on Computer Architecture tutorial (ISCA)*, June 2014.

[106] Casen Hunger, Mikhail Kazdagli, Ankit Rawat, Alex Dimakis, Sriram Vishwanath, and Mohit Tiwari. Understanding contention-based channels and using them for defense. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[107] Intel. Opencl 2.0 shared virtual memory overview, 2014.

[108] Intel. Intel processor graphics gen11 architecture, 2019.

[109] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel Emer. Achieving non-inclusive cache performance with inclusive caches - temporal locality aware (TLA) cache management policies. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2010.

[110] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2010.

[111] Suman Jana and Vitaly Shmatikov. Memento: Learning secrets from process footprints. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 143–157, Washington, DC, USA, 2012. IEEE Computer Society.

[112] Hyeran Jeon and M. Annavaram. Warped-DMR: Light-weight error detection for GPGPU. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, Dec. 2012.

[113] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. MRPB: Memory request prioritization for massively parallel processors. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014.

[114] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A complete key recovery timing attack on a gpu. In *IEEE International Symposium on High Performance Computer Architecture*, HPCA'16, pages 394–405, Barcelona Spain, March 2016. IEEE.

[115] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A novel side-channel timing attack on gpus. In *Proceedings of the on Great Lakes Symposium on VLSI*, VLSI'17, pages 167–172, 2017.

[116] Q. Jiao, M. Lu, H. P. Huynh, and T. Mitra. Improving GPGPU energy-efficiency through concurrent kernel execution and DVFS. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, Feb. 2015.

[117] G. Kadam, D. Zhang, and A. Jog. RCoal: Mitigating GPU timing attack via subwarp-based randomized coalescing techniques. In *Proc. International Symposium on High Performance Computer Architecture (HPCA)*, 2018. Accessed online at `http://adwaitjog.github.io/docs/pdf/rcoal-hpca18.pdf`.

[118] G. Kadam, D. Zhang, and A. Jog. Bcoal: Bucketing-based memory coalescing for efficient and secure gpus. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 570–581, 2020.

[119] Mehmet Kayaalp, Khaled N Khasawneh, Hodjat Asghari Esfeden, Jesse Elwell, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. Ric: Relaxed inclusion caches for mitigating llc side-channel attacks. In *Proceedings of the 54th Annual Design Automation Conference (DAC)*, 2017.

[120] Mehmet Kayaalp, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *Proceedings of the 53th Annual Design Automation Conference*, June 2016.

[121] Mikhail Kazdagli, Vijay Janapa Reddi, and Mohit Tiwari. Quantifying and improving the efficiency of hardware-based mobile malware detectors. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, Oct 2016.

[122] S. K. Khatamifard, L. Wang, S. Köse, and U. R. Karpuzcu. A new class of covert channels exploiting power management vulnerabilities. *IEEE Computer Architecture Letters*, 17(2):201–204, 2018.

[123] Khronos. Webgl timer, 2016. `https://www.khronos.org/registry/webgl/extensions/EXT_disjoint_timer_query/`.

[124] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[125] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 463–480, Austin, TX, August 2016. USENIX Association.

[126] Jingfei Kong, Onur Aciicmez, Jean-Pierre Seifert, and Huiyang Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *Proceedings of the International Symposium on High Performance Comp. Architecture (HPCA)*, February 2009.

[127] Gunjae Koo, Yunho Oh, Won Woo Ro, and Murali Annavaram. Access Pattern-Aware Cache Management for Improving Data Utilization in GPU. In *IProceedings of the International Symposium on Computer Architecture (ISCA)*, June 2017.

[128] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, Baltimore, MD, August 2018. USENIX Association.

[129] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. In *arXiv preprint arXiv:1404.5997v2*, 2014.

[130] Evangelos Ladakis, Lazaros Koromilas, Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. You can type, but you can't hide: A stealthy gpu-based keylogger. 2013.

[131] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. Stealing webpage rendered on your browser by exploiting gpu vulnerabilities. In *IEEE Symposium on Security and Privacy*, SPI'14, pages 19–33, San Jose CA USA, May 2014. IEEE.

[132] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. GPUWattch: Enabling energy optimizations in GPGPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2013.

[133] Ang Li, Gert-Jan van den Braak, Akash Kumar, and Henk Corporaal. Adaptive and transparent cache bypassing for GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2015.

[134] Chao Li, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. Locality-driven dynamic GPU cache bypassing. In *Proceedings of the International Conference on Supercomputing (ICS)*, June 2015.

[135] Qingzheng Li and Amine Bermak. A low-power hardware-friendly binary decision tree classifier for gas identification. In *Journal of Low Power Electronics and Applications*, volume 1, 2011.

[136] Y. Liang, H. P. Huynh, K. Rupnow, R. S. M. Goh, and D. Chen. Efficient GPU spatial-temporal multitasking. *IEEE Transactions On Parallel and Distributed Systems (TPDS)*, 26(3):748–760, 2014.

[137] Zhen Lin, Utkarsh Mathur, and Huiyang Zhou. Scatter-and-gather revisited: High-performance side-channel-resistant aes on gpus. In *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs*, GPGPU '19, page 2–11, New York, NY, USA, 2019. Association for Computing Machinery.

[138] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical keystroke timing attacks in sandboxed javascript. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *Computer Security – ESORICS 2017*, pages 191–209, Cham, 2017. Springer International Publishing.

[139] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, Austin, TX, August 2016. USENIX Association.

[140] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[141] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *IEEE International Symposium on High Performance Computer Architecture*, HPCA'16, pages 406–418, Barcelona, Spain, March 2016.

[142] Fangfei Liu and Ruby B. Lee. Random fill cache architecture. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, Dec. 2014.

[143] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, SP'15, San Jose, CA, USA, May 2015. IEEE.

[144] S. Liu, Y. Wei, J. Chi, F. H. Shezan, and Y. Tian. Side channel attacks in computation offloading systems with gpu virtualization. In *2019 IEEE Security and Privacy Workshops (SPW)*, pages 156–161, 2019.

[145] W. Liu, B. Schmidt, G. Voss, and W. Müller-Wittig. Molecular dynamics simulations on commodity gpus with cuda. In *Proceedings of the International Conference on High Performance Computing (HPC)*, Mar. 2007.

[146] Chao Luo, Yunsi Fei, and David Kaeli. Side-channel timing attack of rsa on a gpu. *ACM Transactions on Architecture and Code Optimization*, 16(3), August 2019.

[147] Chao Luo, Yunsi Fei, Pei Luo, Saoni Mukherjee, and David Kaeli. Side-channel power analysis of a gpu aes implementation. In *33rd IEEE International Conference on Computer Design*, ICCD'15, 2015.

[148] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *39th Annual International Symposium on Computer Architecture*, ISCA'12, pages 118–129, Portland, OR, USA, June 2012.

[149] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. Thermal covert channels on multi-core platforms. In *24th USENIX Security Symposium*, pages 865–880, Washington, D.C., 2015.

[150] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Confidentiality issues on a gpu in a virtualized environment. In *International Conference on Financial Cryptography and Data Security*, pages 119–135, November 2014.

[151] Clémentine Maurice, Manuel Weber, Micheal Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the other side: Ssh over robust cache covert channels in the cloud. In *Network and Distributed System Security Symposium*, NDSS'17. January, 2017.

[152] Wen mei Hwu. *GPU Computing Gems*. Elsevier, 1st. edition, 2011.

[153] Microsoft Azure. GPU-Accelerated Microsoft Azure, 2019. http://www.nvidia.com/object/gpu-accelerated-microsoft-azure.html.

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

[155] Tom M. Mitchell. Machine learning. In *McGraw-Hill Science/Engineering/Math*, 1997.

[156] F. Monrose and A. Rubin. Authentication via keystroke dynamics. In *Proc. ACM International Conference on Computer and Communication Security (CCS)*, 1997.

[157] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *Proc. of USENIX Security Symposium*, Aug. 2007.

[158] Mozilla. Sharedarraybuffer, 2018. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer`.

[159] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu Ghazaleh. Side channel attacks on gpus. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1, 2019.

[160] Hoda Naghibijouybari and Nael Abu-Ghazaleh. Covert Channels on GPGPUs. In *IEEE Computer Architecture Letters*, 2016.

[161] Hoda Naghibijouybari, Khaled Khasawneh, and Nael Abu-Ghazaleh. Constructing and Characterizing Covert Channels on GPGPUs. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2017.

[162] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: Gpu side channel attacks are practical. In *Conference on Computer and Communications Security (CCS)*, pages 2139–2153, 2018.

[163] Alex Nanopoulos, Rob Alcock, and Yannis Manolopoulos. Feature-based classification of time-series data. *International Journal of Computer Research*, 10(3):49–61, 2001.

[164] Naoki Nishikawa, Keisuke Iwai, and Takakazu Kurokawa. High-performance symmetric block ciphers on cuda. In *Second International Conference on Networking and Computing*, ICNC'11, pages 221–227, Osaka Japan, November 2011.

[165] Nvidia. CUDA SDK 2.3. `https://developer.nvidia.com/cuda-toolkit-23-downloads`, 2009.

[166] NVIDIA. Gpu cloud computing, 2017.

[167] NVIDIA. Multi-process service, 2017.

[168] NVIDIA. Nvlink and nvswitch, 2020.

[169] Mike O'Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W. Keckler, and William J. Dally. Fine-grained dram: Energy-efficient dram for extreme bandwidth systems. In *Proceedings of the International Symposium on Microarchitecture*, pages 41–54, 2017.

[170] Lena E. Olson, Jason Power, Mark D. Hill, and David A. Wood. Border control: Sandboxing accelerators. In *48th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO'15, pages 470–481, Waikiki HI USA, December 2015.

[171] Lena E. Olson, Simha Sethumadhavan, and Mark D. Hill. Security implication of third-party accelerator. *IEEE Computer Architecture Letters*, 15(1):50–53, 2015.

[172] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1406–1418, New York, NY, USA, 2015. ACM.

[173] Meltem Ozsoy, Caleb Donovick, Iakov Gorelik, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Malware-aware processors: A framework for efficient online malware detection. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015.

[174] D. Page. Partitioned cache architecture as a side-channel defense mechanism. In *Crypt. ePrint Arch.*, 2005.

[175] S. Pai, R. Govindarajan, and M. J. Thazhuthaveetil. Preemptive thread block scheduling with online structural runtime prediction for concurrent GPGPU kernels. In *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT)*, pages 483–484, 2014.

[176] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU concurrency with elastic kernels. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2013.

[177] Andriy panchenko, Fabian Lanze, Andreas Zinnen, Martin Henze, Jan Pennekamp, Klaus Wehrle, and Thomas Engel. Website fingerprinting at internet scale. In *23rd Internet Society (ISOC) Network and Distributed System Security Symposium (NDSS 2016)*, 2016.

[178] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Chimera: Collaborative preemption for multitasking on a shared GPU. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2015.

[179] A. Peacock, X. Ke, and M. Wilkerson. Typing patterns: A key to user identification. *IEEE Security and Privacy*, 2:40–47, 2004.

[180] Antonio J Peña, Carlos Reaño, Federico Silla, Rafael Mayo, Enrique S Quintana-Ortí, and José Duato. A complete and efficient cuda-sharing solution for hpc clusters. *Parallel Computing*, 40(10):574–588, 2014.

[181] Colin Percival. Cache missing for fun and profit. In *BSDCan*, 2005.

[182] Benoit Baudry Gildas Avoine Pierre Laperdrix, Nataliia Bielova. Browser fingerprinting: A survey. In *arXiv*, May 2019.

[183] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. Cuda leaks: Information leakage in gpu architecture. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(1), 2016.

[184] J.R. Quinlan. Induction of decision trees. In *Machine Learning*, 1986.

[185] Moinuddin K. Qureshi and Yale N. Patt. Utility-based partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, Dec. 2006.

[186] Vignesh T. Ravi, Michela Becchi, Gagan Agrawal, and Srimat Chakradhar. Supporting gpu sharing in cloud environments with a transparent runtime consolidation framework. In *20th international symposium on High performance distributed computing*, HPDC'11, pages 217–228, San Jose, CA, USA, June 2011.

[187] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proc. ACM conference on Computer and communications security*, CCS'09, pages 199–212, Chicago, Illinois, USA, November 2009.

[188] Timothy G. Rogers, Daniel R. Johnson, Mike O'Connor, and Stephen W. Keckler. A variable warp size architecture. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2015.

[189] John Russel. Today's outlook: GPU-accelerated weather forecasting. https://www.hpcwire.com/2015/09/15/todays-outlook-gpu-accelerated-weather-forecasting/.

[190] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript. In Aggelos Kiayias, editor, *Financial Cryptography and Data Security*, pages 247–267, Cham, 2017. Springer International Publishing.

[191] Ali Shafiee, Akhila Gundu, Manjunath Shevgoor, Rajeev Balasubramonian, and Mohit Tiwari. Avoiding information leakage in the memory controller with fixed service policies. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, Dec. 2015.

[192] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on SSH. In *Proc. USENIX Security Symposium*, 2001.

[193] Manish Reddy Sparsh Mittal, S. B. Abhinaya and Irfan Ali. A survey of techniques for improving security of gpus. *Journal of Hardware and Systems Security*, 2:266–285, 2018.

[194] M. Stephenson, S. K. S. Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, and S. W. Keckler. Flexible software profiling of GPU architectures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2015.

[195] J.E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W.R. Davis, P.D. Franzon, M. Bucher, S. Basavarajaiah, Julie Oh, and R. Jenkal. FreePDK: An open-source variation-aware design kit. In *Proceedings of the International Conference on Microelectronic Systems Education*, June 2007.

[196] Paul Stone. Pixel Perfect Timing Attacks with HTML5, 2013. `https://www.contextis.com/media/downloads/Pixel_Perfect_\Timing_Attacks_with_HTML5_Whitepaper.pdf`.

[197] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniek Liu, and Wen Mei Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical report, Mar. 2012.

[198] Synopsys. Synopsys design compiler, 2010. `http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/Pages/default.aspx`.

[199] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling preemptive multiprogramming on gpus. In *41st annual international symposium on Computer architecuture*, ISCA'14, pages 193–204, Minneapolis, Minnesota, USA, June 2014.

[200] X. Tang, A. Pattnaik, H. Jiang, O. Kayiran, A. Jog, S. Pai, M. Ibrahim, M. T. Kandemir, and C. R. Das. Controlled kernel launch for dynamic parallelism in GPUs. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017.

[201] Yingying Tian, Sooraj Puthoor, Joseph L. Greathouse, Bradford M. Beckmann, and Daniel A. Jiménez. Adaptive GPU cache bypassing. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, Feb. 2015.

[202] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1382–1393, New York, NY, USA, 2015. ACM.

[203] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A placement vulnerability study in multi-tenant public clouds. In *24th USENIX Security Symposium (USENIX Security)*, 2015.

[204] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. Gpu-assisted malware. In *2010 5th International Conference on Malicious and Unwanted Software*, pages 1–6, 2010.

[205] Giorgos Vasiliadis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Pixelvault: Using gpus for securing cryptographic operations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1131–1142, Scottsdale Arizona USA, November 2014.

[206] Pepe Vila and Boris Kopf. Loophole: Timing attacks on shared event loops in chrome. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 849–864, Vancouver, BC, 2017. USENIX Association.

[207] Pepe Vila, Boris Köpf, and José F Morales. Theory and practice of finding eviction sets. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 39–54. IEEE, 2019.

[208] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili. Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on GPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2015.

[209] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. Laperm: Locality aware scheduler for dynamic parallelism on gpus. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2016.

[210] Xin Wang and Wei Zhang. Cracking randomized coalescing techniques with an efficient profiling-based side-channel attack to gpu. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '19, New York, NY, USA, 2019. Association for Computing Machinery.

[211] Xin Wang and Wei Zhang. An efficient profiling-based side-channel attack on graphics processing units. In Kim-Kwang Raymond Choo, Thomas H. Morris, and Gilbert L. Peterson, editors, *National Cyber Summit (NCS) Research Track*, pages 126–139, Cham, 2020. Springer International Publishing.

[212] Yao Wang and G. Edward Suh. Efficient timing channel protection for on-chip networks. In *Proceedings of the International Symposium on Networks-on-Chip (NOCS)*, May 2012.

[213] Yao Wang and G. Edward Suh. Timing channel protection for a shared memory controller. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2014.

[214] Zhenghong Wang and Ruby B. Lee. New constructive approach to covert channel modeling and channel capacity estimation. In *Proceedings of the International Conference on Information Security (ISC)*, 2005.

[215] Zhenghong Wang and Ruby B. Lee. Covert and side channels due to processor architecture. In *22nd Annual Computer Security Applications Conference*, ACSAC '06, pages 473–482, Miami Beach, FL, USA, December 2006. IEEE.

[216] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2007.

[217] Zhenghong Wang and Ruby B. Lee. A novel cache architecture with enhanced performance and security. In *41st IEEE/ACM International Symposium on Microarchitecture*, MICRO'08, pages 83–93, Lake Como Italy, November 2008. IEEE.

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

[219] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, Mar. 2016.

[220] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. Quality of service support for fine-grained sharing on gpus. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2017.

[221] Junyi Wei, Yicheng Zhangy, Zhe Zhou, Zhou Liy, and Mohammad Abdullah Al Faruque. Leaky dnn: Stealing deep-learning model secret with gpu context-switching side-channel. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020.

[222] Zachary Weinberg, Eric Y. Chen, Pavithra Ramesh Jayaraman, and Collin Jackson. I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 147–161, Washington, DC, USA, 2011. IEEE Computer Society.

[223] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. Jackhammer: Efficient rowhammer on heterogeneous fpga-cpu platforms. In *arXiv:1912.11523*, 2020.

[224] K. Wilcox, D. Akeson, H.R. Fair, J. Farrell, D. Johnson, G. Krishnan, H. Mclntyre, E. McLellan, S. Naffziger, R. Schreiber, S. Sundaram, and J. White. 4.8 A 28nm x86 APU optimized for power and area efficiency. In *International Solid- State Circuits Conference (ISSCC)*, Feb. 2015.

[225] Henry Wong, M. M. Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *IEEE International Symposium on Performance Analysis of Systems & Software*, ISPASS'10, 2010.

[226] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter. Enabling and exploiting flexible task assignment on GPU through sm-centric program transformations. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2015.

[227] Shujiang Wu, Song Li, Yinzhi Cao, and Ningfei Wang. Rendered private: Making GLSL execution uniform to prevent webgl-based browser fingerprinting. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1645–1660, Santa Clara, CA, August 2019. USENIX Association.

[228] Xiaolong Xie, Yun Liang, Yu Wang, Guangyu Sun, and Tao Wang. Coordinated static and dynamic cache bypassing for GPUs. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2015.

[229] Qiumin Xu and Murali Annavaram. PATS: Pattern aware scheduling and power gating for GPGPUs. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Aug. 2014.

[230] 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. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2016.

[231] Qiumin Xu, Hoda Naghibijouybari, Shibo Wang, Nael Abu-Ghazaleh, and Murali Annavaram. Gpuguard: Mitigating contention based side and covert channel attacks on gpus. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, pages 497–509, New York, NY, USA, 2019. ACM.

[232] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*, Los Alamitos, CA, USA, may 2019. IEEE Computer Society.

[233] Mengjia Yan, Yasser Shalabi, and Josep Tolrrellas. Replayconfusion: Detecting cache-based covert channel attacks using record and replay. In *49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO'16, Taipei Taiwan, October 2016. IEEE.

[234] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. Are coherence protocol states vulnerable to information leakage? In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2018.

[235] Zhihao Yao, Zongheng Ma, Ardalan Sani, and Aparna Chandramowlishwaran. Sugar: Secure GPU acceleration in web browsers. In *Proc. International Conference on Architecture Support for Operating Systems and Programming Languages (ASPLOS)*, 2018.

[236] Zhihao Yao, Saeed Mirzamohammadi, Ardalan Amiri Sani, and Mathias Payer. Milkomeda: Safeguarding the mobile gpu interface using webgl security checks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1455–1469, New York, NY, USA, 2018. Association for Computing Machinery.

[237] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association.

[238] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B Lee, and Gernot Heiser. Mapping the intel last-level cache. *IACR Cryptology ePrint Archive*, 2015:905, 2015.

[239] Michael K. Reiter Yinqian Zhang, Ari Juels and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 990–1003, 2014.

[240] Boris Zbarsky. Reduce resolution of performance.now, 2015. `https://hg.mozilla.org/integration/mozilla-inbound/rev/48ae8b5e62ab`.

[241] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 305–316, New York, NY, USA, 2012. Association for Computing Machinery.

[242] M. Zhao and G. E. Suh. Fpga-based remote power side-channel attacks. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 229–244, 2018.

[243] J. Zhong and B. He. Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling. *IEEE Transactions on Parallel and Distributed System (TPDS)*, 25(6):1522–1532, 2014.

[244] Yanqi Zhou, Sameer Wagh, Prateek Mittal, and David Wentzlaff. Camouflage: Memory traffic shaping to mitigate timing attacks. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2017.

[245] Zhe Zhou, Wenrui Diao, Xiangyu Liu, Zhou Li, Kehuan Zhang, and Rui Liu. Vulnerable gpu memory management: Towards recovering raw data from gpu. *Proceedings on Privacy Enhancing Technologies*, 2017:57 – 73, 2017.

[246] Zhiting Zhu, Sangman Kim, Yuri Rozhanski, Yige Hu, Emmett Witchel, and Mark Silberstein. Understanding the security of discrete gpus. In *Proceedings of the General Purpose GPUs*, GPGPU'10, pages 1–11, Austin TX USA, 2017.

[247] P. Zou, A. Li, K. Barker, and R. Ge. Fingerprinting anomalous computation with rnn for gpu-accelerated hpc machines*. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 253–256, 2019.