Rendered Insecure: GPU Side Channel Attacks are Practical

Hoda Naghibijouybari
University of California, Riverside
hnagh001@ucr.edu

Ajaya Neupane
University of California, Riverside
ajaya@ucr.edu

Zhiyun Qian
University of California, Riverside
zhiyunq@cs.ucr.edu

Nael Abu-Ghazaleh
University of California, Riverside
nael@cs.ucr.edu

ABSTRACT

Graphics Processing Units (GPUs) are commonly integrated with computing devices to enhance the performance and capabilities of graphical workloads. In addition, they are increasingly being integrated in data centers and clouds such that they can be used to accelerate data intensive workloads. Under a number of scenarios the GPU can be shared between multiple applications at a fine granularity allowing a spy application to monitor side channels and attempt to infer the behavior of the victim. For example, OpenGL and WebGL send workloads to the GPU at the granularity of a frame, allowing an attacker to interleave the use of the GPU to measure the side-effects of the victim computation through performance counters or other resource tracking APIs. We demonstrate the vulnerability using two applications. First, we show that an OpenGL based spy can fingerprint websites accurately, track user activities within the website, and even infer the keystroke timings for a password text box with high accuracy. The second application demonstrates how a CUDA spy application can derive the internal parameters of a neural network model being used by another CUDA application, illustrating these threats on the cloud. To counter these attacks, the paper suggests mitigations based on limiting the rate of the calls, or limiting the granularity of the returned information.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**;

KEYWORDS

GPU, side channel, website fingerprinting, keystroke timing attack

ACM Reference Format:
Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. 2018. Rendered Insecure: GPU Side Channel Attacks are Practical. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3243734.3243831

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10.
https://doi.org/10.1145/3243734.3243831

1 INTRODUCTION

Graphics Processing Units (GPUs) are integral components to most modern computing devices, used to optimize the performance of today's graphics and multi-media heavy workloads. They are also increasingly integrated on computing servers to accelerate a range of applications from domains including security, computer vision, computational finance, bio-informatics and many others [52]. Both these classes of applications can operate on sensitive data [25, 31, 57] which can be compromised by security vulnerabilities in the GPU stack.

Although the security of GPUs is only starting to be explored, several vulnerabilities have already been demonstrated [46, 49, 55, 58, 63, 71, 74]. Most related to this paper, Luo et al. demonstrated a timing channel from the CPU side timing a GPU operation. In particular, they assume that the GPU is running an encryption library, and time encryption of chosen text blocks. The encryption runtime varies depending on the encryption key: the memory access patterns are key-dependent causing timing differences due to GPU memory coalescing effects enabling a timing side channel attack on the key [40]. In [40], the attacker needs to launch the encryption kernel on GPU and measure the whole kernel execution time on its own process (on CPU side), totally different than our threat model that investigates side channel between two concurrent apps on the GPU. More recently, Naghibijouybari et al. showed that covert channels between colluding concurrently running CUDA applications (CUDA is Nvidia's programming language for general purpose workloads on the GPU [4]) on a GPU may be constructed [54, 55]. Neither of these papers demonstrates a general side channel attack, which is the focus of this paper.

This paper explores 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). If indeed possible, 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.

A pre-requisite of architectural side channel attack is the co-location of the attacker and the victim in the resource space such that the attacker can create and measure contention. We systematically characterize the situations where a spy can co-locate and

measure side channel behavior of a victim in both the graphics and the computational stacks of the Nvidia family of GPUs. In the case of OpenGL workloads, we discover that kernels (shader programs) can be concurrently scheduled provided there are sufficient resources to support them. We also verify that the same is true for competing CUDA workloads. Finally, when workloads originate from both CUDA and OpenGL, they interleave the use of the GPU at a lower concurrency granularity (interleaving at the computational kernel granularity). We discuss co-location possibilities for each type of the attack.

Armed with the co-location knowledge, we demonstrate a family of attacks where the spy can interleave execution with the victim to extract side channel information. We explore using (1) Memory allocation APIs; (2) GPU performance counters; and (3) Time measurement as possible sources of leakage. We show that all three sources leak side channel information regarding the behavior of the victim. We successfully build three practical and dangerous end-to-end attacks on several generations of Nvidia GPUs.

To illustrate attacks on graphics applications we implement a web-fingerprinting attack that identifies user browsing websites with high accuracy. We show an extension to this attack that tracks user activity on a website, and captures keystroke timing. We also illustrate attacks on computational workloads showing that a spy can reconstruct the internal structure of a neural network with high accuracy by collecting side channel information through the performance counters on the GPU.

We explore possible mitigation to this type of attack. Preventing or containing contention on GPUs by allocating them exclusively or changing their design could limit the leakage, but is likely impractical. Thus, we focus on limiting the attacker's ability to measure the leakage. We show that solutions that interfere with the measurement accuracy can substantially limit the leakage and interfere with the attacker's ability to extract sensitive information.

In summary, the contributions of this paper are:

- We investigate the feasibility of GPU side channel attacks due to resource contention within the GPU.
- We reverse engineer a range of Nvidia GPU models and extract internal scheduling parameters and measurement APIs that can be used to carry out GPU side channel attacks.
- We demonstrate practical attacks on both graphics and computational GPU workloads, as well as across them. We believe these are the first reported general side channel attacks on GPUs.
- We discuss and evaluate possible mitigations based on limiting the rate or precision of the calls to the vulnerable APIs.

Disclosure: We have reported all of our findings to Nvidia. We understand that they intend to publish a patch that offers system administrators the option to disable access to performance counters from user processes. We also shared a draft of the paper with the AMD and Intel security teams to enable them to evaluate their GPUs with respect to such vulnerabilities.

2 GPU PROGRAMMING INTERFACES AND ARCHITECTURE

This section first overviews GPU programming interfaces and GPU architecture to provide an idea about how they are programmed, and how 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 [17], or WebGL [20] 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 making all attacks practical on a desktop. In theory, a JavaScript application may launch the attack using WebGL, but we found that current versions of WebGL do not expose measurement APIs that allow leakage.

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 [4], Vulkan [19] and OpenCL [13]. 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 [6]. 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 [3] and are increasingly moving towards GPU concurrent multiprogramming.

We did not evaluate attacks on Android devices that incorporate Nvidia GPUs (few phones use the Nvidia Tegra GPU), but plan to do so in our future research. On Android devices, a modified version of OpenGL is available called OpenGL-ES. It offers access to performance counters (which are used in some of our attacks) but not to the memory API. Additional leakage is available on Android through detailed profiling of rendering behavior, but requires the application to obtain a developer permission.

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 [4], and there are already thousands of applications available for it on desktops, and mobile devices. With its performance and power advantages for a range of useful applications, even though it is not available by default, it is likely that many desktops and mobile devices will install CUDA (or OpenCL) enabling attacks that originate from CUDA on graphics workloads.

2.2 GPU Architecture Overview

We briefly review the GPU architecture to provide some insight into its execution model and how contention arises. The architecture also illustrates that due to the high parallelism and small shared caches, conventional CPU attacks such as prime-probe cache attacks are virtually impossible; it is difficult to correlate accesses to the potential thousands of concurrently executing threads that may have generated them.

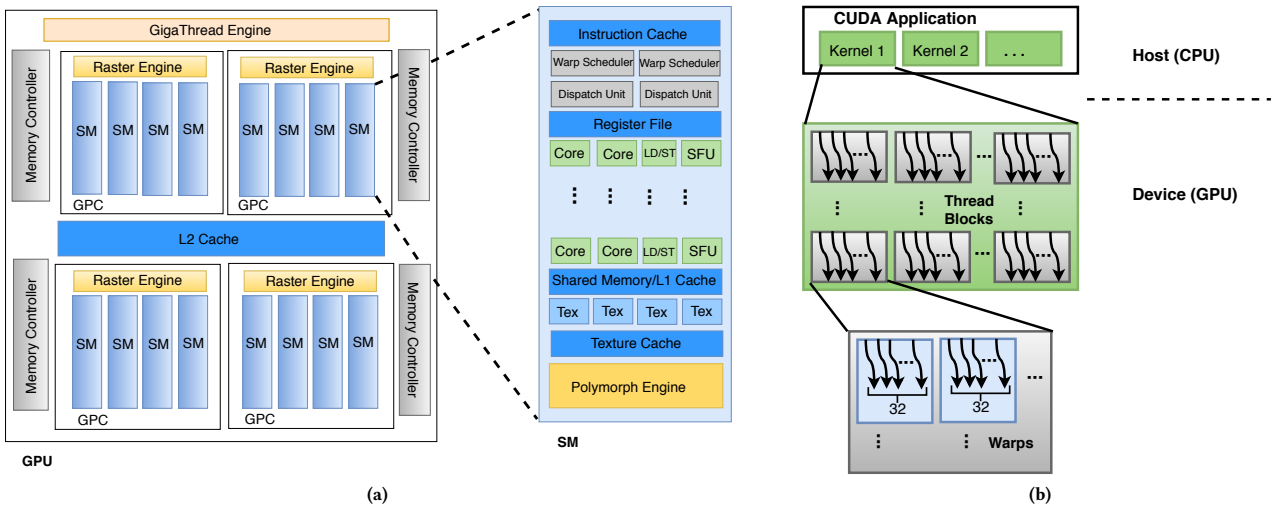

Figure 1: GPU overview (a) Architecture; (b) Application

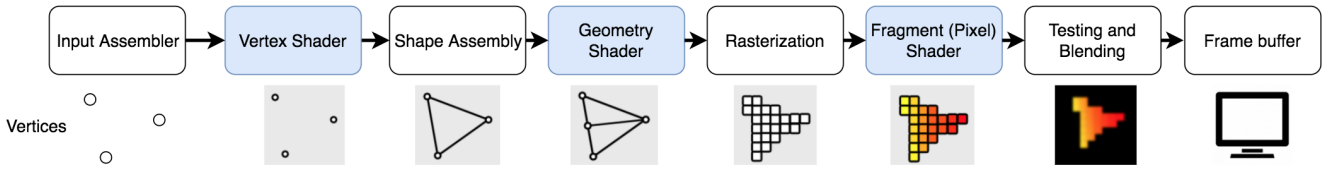

Figure 2: Graphics processing pipeline

Figure 1a presents an architecture overview of a typical 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, includes 15 SMs [11]. 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 1b, 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 parallel matrix multiplication in a computation kernel. Each kernel is decomposed into blocks of threads 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.

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 1a), resulting in bandwidths of several hundred gigabytes per second, but with a high latency. The impact of the latency is 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, making it difficult to correlate fine-grained side channel leakage (e.g., cache miss on a cache set) to a particular computation source.

Graphics Pipeline: 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 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. When measuring performance counters or memory usage, these values leak information about the graphics workload being rendered by the GPU.

3 ATTACK SCENARIOS AND LEAKAGE SOURCES

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

3.1 Attack Scenarios

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 3, 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 on a CUDA victim: attacks from a CUDA spy on a CUDA workload typically on the cloud (Figure 3, middle) where CUDA libraries and drivers are installed; and
- CUDA spy and graphics victim (Cross-Stack): on user systems where CUDA is installed, attacks from CUDA to graphics applications are possible (Figure 3, right).

In the first attack, we assume that the attacker exploits the graphics stack using APIs such as OpenGL or WebGL. In attacks 2 and 3, we assume that a GPU is accessible to the attacker using CUDA or OpenCL. We reverse engineer the co-location properties for each attack vector and identify the leakage sources available in each scenario.

3.2 Available Leakage Vectors on GPUs

Prior work has shown 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 [55]. However, such fine-grained leakage is more difficult to exploit for a side channel attack for a number of reasons:

- The large number of active threads, and the relatively small cache structures make it difficult to carry out high-precision prime-probe or similar attacks on data caches [42, 43, 48, 62].
- The SIMT computational model limits leakage due to data dependent control flow (e.g., if statements). In particular, on a GPU both the if and else clauses of an if-then-else statement are executed if at least one thread is interested in each clause equalizing the side channel signal. This type of side channel is often the target of timing/power analysis, branch prediction based attacks [33, 34] or instruction cache attacks on CPUs [23], but is unavailable on GPUs.
- True colocation with applications running on the GPU concurrently is not possible in all scenarios (e.g., OpenGL and CUDA kernels do not concurrently execute).

Thus, instead of targeting fine-grained contention behavior, most of our attacks focus on aggregate measures of contention through available resource tracking APIs. 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 both Nvidia [15] and AMD GPUs [14] and 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 [10] and on Linux and Android [8, 18] for graphics applications, while AMD supports Linux and Windows [7]. 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.

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 [15] 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". Note that AMD also provides the similar APIs to query the available GPU memory on both OpenCL

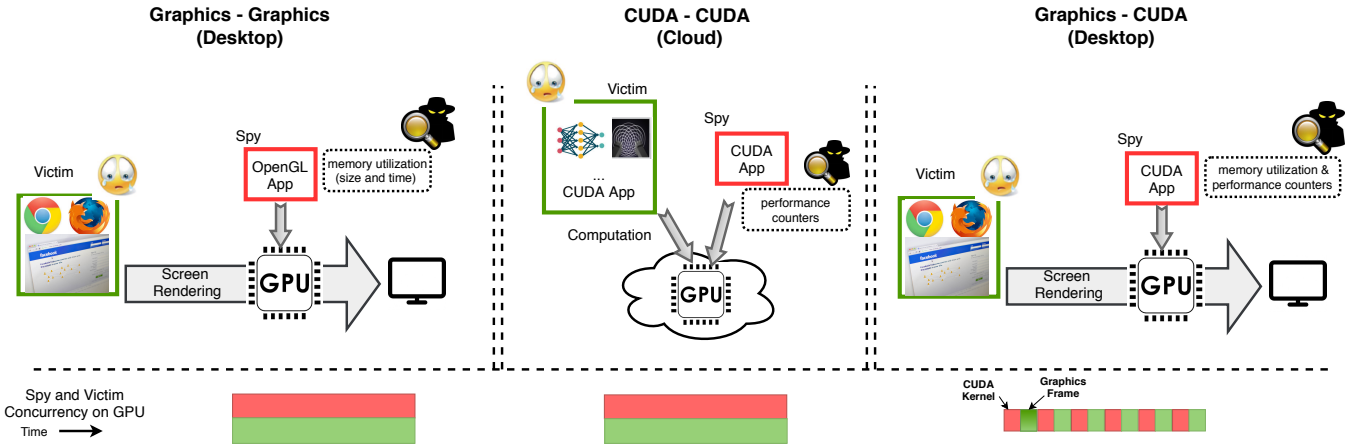

Figure 3: Threat Scenarios

Table 1: GPU performance counters

Category	Event/Metric
Memory	Device memory read/write throughput/transactions Global/local/shared memory load/store throughput/transactions L2 read/write transactions, Device memory utilization
Instruction	Control flow, integer, floating point (single/double) instructions Instruction executed/issued, Issued/executed IPC Issued load/store instructions Issue stall reasons (data request, execution dependency, texture, ...)
Multiprocessor	Single/Double-precision function unit utilization Special function unit utilization Texture function unit utilization, Control-flow function unit utilization
Cache	L2 hit rate (texture read/write) L2 throughput/transaction (Reads/writes, Texture read/writes)
Texture	Unified cache hit rate/throughput/utilization

and OpenGL applications through "cl_amd_device_attribute_query" [12] and "ATI_meminfo" [14] extensions respectively, making our attacks possible on AMD GPUs as well.

Measuring Performance Counters: We use the Nvidia profiling tools [10] to monitor the GPU performance counters from a CUDA spy [8]. Table 1 summarizes some of the important events/metrics tracked by the GPU categorized into five general groups: memory, instruction, multiprocessor, cache and texture. 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.

Measuring Timing: It is also possible to measure the time of individual operation in scenarios where the spy and the victim are concurrently running to detect contention. In scenarios where the victim and spy interleave, measuring timing also discloses the execution time of the victim computation kernel.

Experimental Setup: We verified the existence of all the reported vulnerabilities in this paper 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 in this paper. 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.

4 THREAT SCENARIO 1: GRAPHICS SPY AND GRAPHICS VICTIM

We consider the first threat scenario where an application uses a graphics API such as OpenGL to spy on another application that uses the GPU graphics pipeline (Figure 3, left). First, we have to reverse engineer how concurrent graphics workloads from two applications share the use of the GPU (we call this co-location).

Reverse Engineering Co-location: To understand how two concurrent applications share the GPU, we carry out a number of experiments and use their results 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. Each thread is responsible for assigning color to each pixel in fragment shader. We need to know about SM-ID and timing information of each thread on each OpenGL workload to evaluate contention. OpenGL developers (Khronos group) provide two extensions: "NV_shader_thread_group " [16] which enable programmers to query the ThreadID, the WarpID and the SM-ID in OpenGL shader codes and "ARB_shader_clock " [2] 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.

4.1 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 [20]. Based on WebGL statistics [22] 98% of visitors to a series of websites used WebGL enabled browsers and [71] report that at least 53% of the top-100 sites, and 16.4% of the top-10,000 sites use WebGL. 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 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. GPU accelerated rendering allows for more seamless animations and better performance, while freeing up the CPU. As a result, the Chromium Project's GPU Architecture Roadmap [5] 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 6.

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. For website fingerprinting, we leverage machine learning algorithms to classify the traces to infer which websites the victim is likely to have visited. The machine learning classifier is trained on similar traces obtained for a set of the top ranked websites according to Alexa [1].

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. The spy can reliably obtain all allocation events. To illustrate the side channel signal, Figure 4 shows the GPU memory

[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.

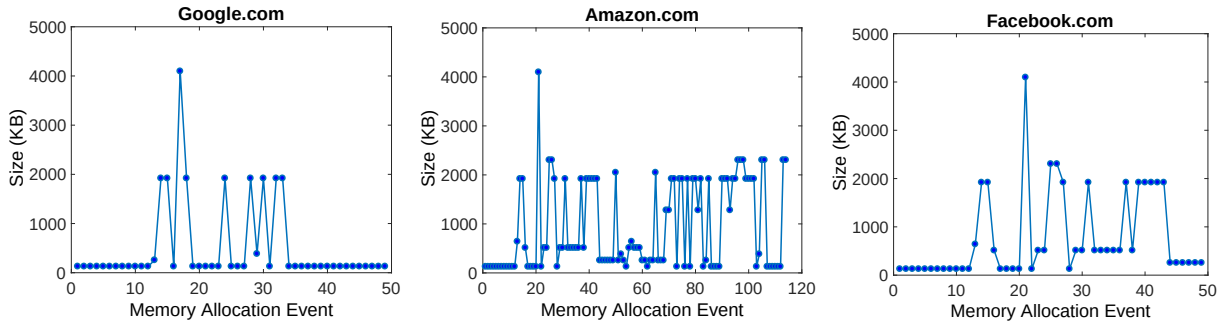

Figure 4: Website memory allocation on GPU (a) Google; (b) Amazon; (c) Facebook

allocation trace when Google, Amazon and Facebook 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 [1]. 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.

Classification: 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 [56]. We computed these features separately for the first and the second half of the timeseries 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 [44] (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 2: Memory API based website fingerprinting performance: 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 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%.

Generalization across screen sizes: We considered full screen browser window in our experiments. Given that different users may have different size browser windows, we wanted to check if the attack generalizes across window sizes. We discover that changing the window size results in a similar signal with different amplitude for most websites, and for 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 smaller window). By training the ML model using full screen data and testing with window size 1280*810 and 800*600 measurements, we still see average accuracy of 82% and 63% respectively. We believe performance can also be improved by training with measurements at different window sizes.

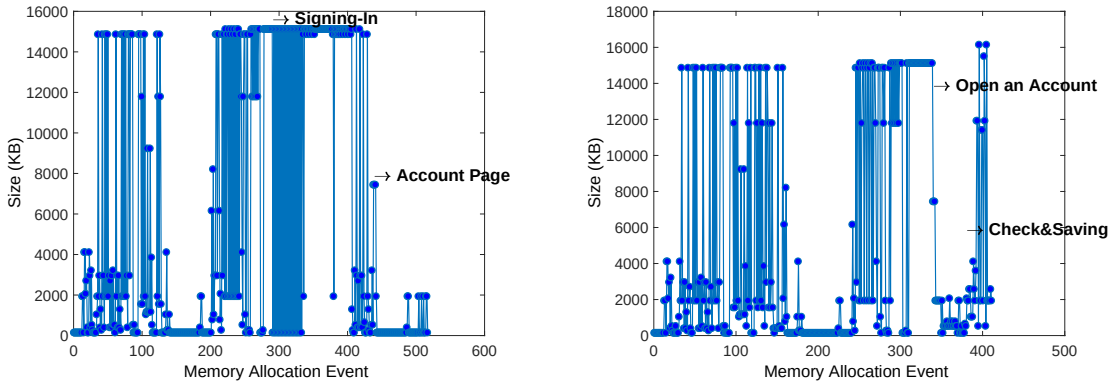

Figure 5: User activity memory trace on Bank of America website (a) sign-in ; (b) Open account: Checking & Saving

4.2 Attack II: User Activity Tracking and Keystroke Monitoring

We follow up on the website fingerprinting attack with a user activity tracking and keystroke monitoring attack. The first component of the attack uses 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 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 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 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)

Password Textbox Identification and Keystroke Timing Inference: 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 or similar structure animating the characters). More specifically, from the same signal, it contains (1) the size of memory allocation by the victim (with granularity of 128KB), 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 (presumably due to the blinking cursor). However, if a new character is typed, the box is immediately re-rendered (resulting in a smaller interval). This effect is shown in Figure 6, 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. For instance, as seen in Figure 6, at allocation event 8 (X-axis), there is the first keystroke, as the allocation happened faster than the regular 600ms interval. At event 9, 600ms passes without user input. Next, at event 10, the second keystroke occurred,

after another 200ms or so. So the inter-arrival time between the first and second keystroke is then 600 + 200 = 800ms.

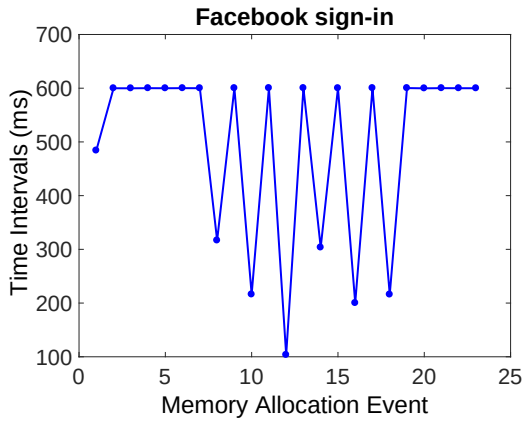

Figure 6: Timing GPU memory allocations for a 6-character password

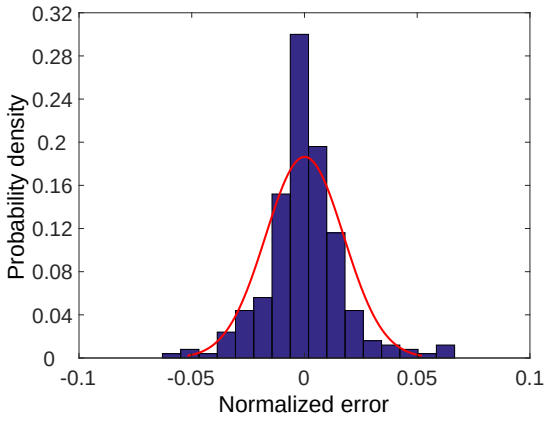

Figure 7: Measurement error distribution of inter-keystroke time

Prior work has shown that inter-arrival times of keystrokes can leak information about the actually characters being typed by the user [65]. 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 first observe that there is a constant bias in the value corresponding the overhead of measurement of around 10ms from a separate experiment of several hundred key presses; we adjust all the samples in the testing data by removing that constant amount. We then compute the normalized error as the difference between the GPU measured interval and the ground truth measured on the CPU side. Figure 7 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 8 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.

It is worth noting that the inter-keystroke time we extract can be closely tied to specific textboxes and specific webpages, representing a practical and serious threat to user privacy in browsing the web.

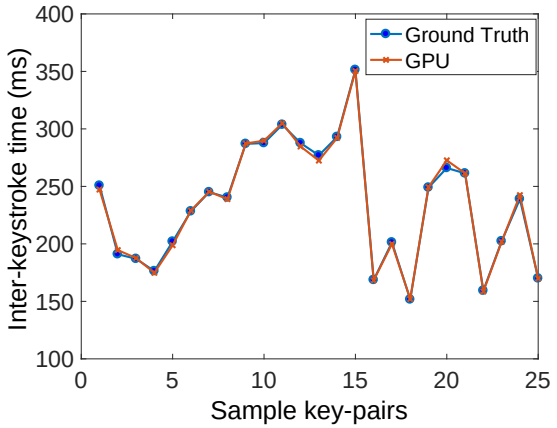

Figure 8: Keystroke timing: Ground Truth vs. GPU

5 THREAT SCENARIO 2: CUDA SPY AND 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 needs to be supported. Modern GPUs support multiprogramming through multiple hardware streams with multi-kernel execution either within the same process, or using a multi-process service (MPS) [9], which allows execution of concurrent kernels from different processes on the GPU. MPS is already supported on GPUs with hardware queues such as the Hyper-Q support available on 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 newest Volta architecture provides hardware support for 32-concurrent address spaces/page tables on the GPU [32]. In fact, all three GPUs we tested for the paper support MPS.

We assume that the two applications are launched to the same GPU. In most existing cloud setting, the GPU is shared among applications on the same physical node, via I/O pass-through. Co-location of attacker and victim VMs on the same cloud node is an orthogonal problem investigated in prior works [24, 64]. 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. We make sure that we have at least one thread block on each SM to enable measurement of the performance counters specific to each SM. We also have different threads (or warps) utilize different hardware resources (like functional units, different caches and global memory) in parallel to create contention in different resource spaces.

5.1 Attack III: Neural Network Model Recovery

In this attack, our goal is to demonstrate side channels that can be exploited between different computational applications. In this attack scenario, a spy 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 [28]; in such an application, the internal structure of the neural network can be a critical trade secret and the target of model extraction attacks [67].

We use prior results of reverse engineering the hardware schedulers on GPUs [55] 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. Thus, the spy runs and collects hardware performance counter values over time. 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.

Table 4 summarizes the most top ranked features selected in the classification. 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).

Table 4: Top ranked counters for classification

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

Table 5: 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 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.

6 THREAT SCENARIO III: CUDA SPY ON A GRAPHICS VICTIM

Finally, we demonstrate threat scenario III 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.

6.1 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 4. 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

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

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.

6.2 Attack IV: 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 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.

Table7 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 6: Top ranked performance counter features

GPU Performance Counter	Features
Fb_subp0/1_read_sectors[4]	slope
Device memory read transactions	slope, mean
L2_subp0/1/2/3_read_sector_misses[5]	slope, sd
L2_subp0/1/2/3_total_read_sector_queries[6]	slope, sd
Instruction issued	skew, kurtosis

Table 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)

7 ATTACK MITIGATION

The attack may be mitigated completely by removing the shared resource APIs such as the memory API and the performance counters. However, legitimate applications require the use of these interfaces for uses such as tuning their performance. Thus, our goal is to weaken the signal that the attacker gets, making it harder to infer sensitive information.

We evaluate reducing the leakage by:

- Rate limiting: limit the rate an application can call the memory API. In the memory based attack, we can query the API every 60 us (about 16000 times per second), leading to precise identification of memory events corresponding to every object uploaded to the GPU. As we reduce the allowable rate, individual allocation and deallocation events start to be missed, reducing the information in the signal.
- Precision limiting: limit the granularity of the measurement. For example, for the memory API, we set a granularity (e.g., 4Mbytes) and report the memory allocation rounded up to this granularity. Small size allocations can be missed, and the precise size of allocations becomes unknown, weakening the signal.

To evaluate the defenses, we collect data for the top 50 Alexa websites and retrain the machine learning model with the defenses in place. The classification precision decreases with rate limiting

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

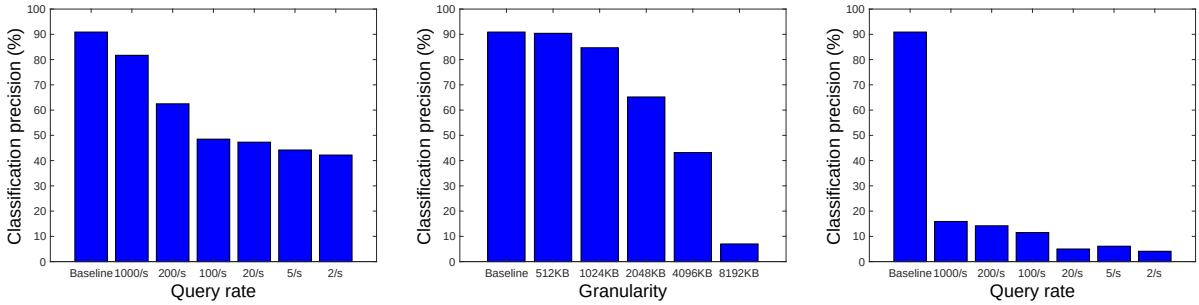

Figure 9: Classification precision with (a) Rate limiting; (b) Granularity limiting; (c) Rate limiting at 4MB granularity

defense as shown in Figure 9a, but its success is limited (as 40% precision can still be achieved under aggressive rate limit). We repeat the experiments approach as we reduce the precision of the reported available memory. In our experiments, for full screen window size, the smallest and most memory allocations on GPU for different websites are 128 KB. If we decrease the granularity and bin all memory allocations to 512KB, 1024KB, 2048KB and 4096KB, the precision of retrained model will be decreased as shown in Figure 9b. By 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 9c. While reducing precision, we believe these mitigations retain some information for legitimate applications to measure their performance, while preventing side channel leakage across applications.

To further evaluate the rate limiting mitigation, we abuse the memory API to implement a covert channel and investigate performance of covert channel with and without the mitigation. To construct the covert channel, a spy application queries the memory API repeatedly to probe the available memory on GPU and detect new memory allocations. On the other side, a trojan application allocates a constant amount of memory on GPU to send "1" and does nothing to send "0". Without any mitigation in place, the achievable covert channel bandwidth is as high as 4Kbps, while rate limiting mitigation decreases the bandwidth to about 910bps, 98bps and 2bps for 1000/s, 100/s and 2/s cases respectively.

We believe similar defenses (limiting the rate, or granularity of measurement) can mitigate performance counter side channels, but we did not experimentally evaluate them. 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.

8 RELATED WORK

We organize the discussion of related work into two different groups: (1) Related work to our attacks; and (2) Covert and side channel attacks on GPUs.

8.1 Related Work to Our Attacks

Different attack vectors have been proposed for website fingerprinting. Panchenko et al. [60] and Hayes et al. [38] capture traffic generated via loading monitored web pages and feed the traces to machine learning based classification. Felten and Schneider [35] utilize browser caching and construct a timing channel to infer the victim visited websites. Jana and Shmatikov [39] 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. [70] presented a user interaction attack (victim's action on website leaks its browsing history) and a timing side channel attack for browser history sniffing.

Some of the principles used in our attack have also been leveraged by other researchers. Goethem et al. [68] and Bortz et al. [26] propose cross-site timing attacks on web browsers to estimate the size of cross-origin resources or provide user information leakage from other site. [66] and [45] propose timing channels by measuring the time elapsed between frames using a JavaScript API. Kotcher et al. [45] found that after applying CSS filter to framed document, its rendering time becomes related to its content. Similarly, Stone et al. [66] 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 attacks. Oren et al. [59] implement a last level cache prime+probe side channel in JavaScript. More recently, Vila and Kopf [69] 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. Gulmezoglu et al. [37] proposed a side channel on per-core/per-process CPU hardware performance counters (which are limited in number). Zhang et al. [72] identify several iOS APIs as attack vectors to implement cross-app information leakage to extract private user information, detecting installed apps, etc. 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.

Table 8 compares the classification accuracy of our memory based and performance counter based attacks to other previously

Table 8: Classification accuracy comparison of website-fingerprinting attacks on Alexa top websites

	Attack Vector	Accuracy (%)	# of Websites	Browser
Mem-based attack	side channel (GPU memory API)	90.4	200	Chrome
PC-based attack	side channel (GPU performance counters)	93	200	Chrome
[60]	side channel (traffic analysis)	92.52	100	Tor
[39]	side channel (memory footprint via procfs)	78	100	Chrome
[59]	side channel (LLC)	82.1 (88.6)	8	Safari (Tor)
[69]	side channel (shared event loop)	76.7	500	Chrome
[37]	side channel (CPU performance counters)	84	30	Chrome
[72]	side channel (iOS APIs)	68.5	100	Safari
[46]	leftover memory on GPU	95.4	100	Chrome

published website-fringerprinting attacks. All the attacks use machine learning methods to classify collected data from Alexa top websites. All attacks, other than [46] which uses leftover memory, exploit side channel leakage. Although there are differences in terms of the number of websites considered, and the browser that is attacked, our attacks are among the most accurate.

Leakage through the keystroke timing pattern is also a known effect which has been exploited as a user authentication mechanism [30, 53, 61]. Keystroke timing has also been used to compromise/weaken passwords [65] or compromise user privacy [27]. Lipp et al. [47] propose a keystroke interrupt-timing attack implemented in JavaScript using a counter as a high resolution timer. Our attack provides an accurate new leakage of keystroke timing to unauthorized users enabling them to implement such attacks. We demonstrated that GPU side channels threaten not only graphics applications, but also computational workloads running on GPU.

A number of attacks have been published targeting GPUs or their software stack on the CPU [21, 71, 74]. [46, 51, 63, 73] propose information leakage due to left-over memory from processes that recently terminated (not side channel). This class of vulnerability can be closed by clearing memory before it gets reallocated to a different application.

8.2 Side Channel Attacks on GPUs

Jiang et al. [40] conduct a timing attack on a CUDA AES implementation. 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. Thus, in a GPU implemented 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 [41] 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 bank conflicts generated by threads within the warp. The attacks require the spy to be able to trigger the launch of the victim kernel. The self-contention exploited in the first attack [40] cannot be used for a side-channel between two concurrent applications. Moreover, these attacks are only used to exploit a CUDA victim. Luo et al. study a power side channel attack on AES encryption executing on a GPU [49]. This attack requires physical access to the GPU to measure the power.

Frigo et al. [36] use WebGL timing APIs to implement rowhammer attack on memory. They use a timing channel to find contiguous area of physical memory rather than extracting application state like our attack, totally different threat model.

Naghibijouybari et al. [55] construct three types of covert channels between two colluding CUDA applications. We use their results for reverse engineering the co-location of CUDA workloads in our third attack on the neural network application. The fine-grained timing information that they use in a covert channel by enforcing regular contention patterns is too noisy to exploit for side-channel attacks due to the large number of active threads.

This work is the first that explores side channels due to contention between two GPU applications. It is also the only GPU attack to compromise graphics applications; prior works consider CUDA applications only. Many side channel attacks have been proposed on CPUs; our memory API attack bears similarity to side channel attacks through `procfs` [29, 39]. Our defense mechanism is in the spirit of other side channel defenses that limit the leakage by interfering with the measurement APIs [50, 72].

9 CONCLUDING REMARKS

In this paper, we explore side channel attacks among applications concurrently using the Graphical Processing Unit (GPU). We 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 from the Rodinia benchmark, 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.

Our attacks demonstrate that side channel vulnerabilities are not restricted to the CPU. Any shared component within a system

can leak information as contention arises between applications that share a resource. Given the wide-spread use of GPUs, we believe that they are an especially important component to secure.

The paper also considered possible defenses. We proposed 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 showed that such defenses substantially reduce the effectiveness of the attack, to the point where the attacks are no longer effective. Finding the right balance between utility and side channel leakage for general applications is an interesting tradeoff to study for this class of mitigations.

ACKNOWLEDGEMENT

The authors would like to thank the anonymous reviewers for their valuable comments and helpful suggestions. The work is supported by the National Science Foundation under Grant No.:CNS-1619450.

REFERENCES

[1] 2018. Alexa Top Sites. https://www.alexa.com/topsites.
[2] 2018. ARB Extenstion, Khronos Group. https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_shader_clock.txt.
[3] 2018. CUDA-enabled GPUs, Nvidia. https://developer.nvidia.com/cuda-gpus.
[4] 2018. CUDA, Nvidia. https://developer.nvidia.com/cuda-zone/.
[5] 2018. GPU Architecture Roadmap, The Chromium Projects. https://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome/gpu-architecture-roadmap.
[6] 2018. GPU Cloud Computing, Nvidia. https://www.nvidia.com/en-us/data-center/gpu-cloud-computing/.
[7] 2018. GPUPerfAPI. https://gpuopen.com/gaming-product/gpuperfapi/.
[8] 2018. Linux Graphics Debugger, Nvidia. https://developer.nvidia.com/linux-graphics-debugger.
[9] 2018. Multi-Process Service, Nvidia. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.
[10] 2018. NVIDIA Profiler User's Guide. http://docs.nvidia.com/cuda/profiler-users-guide/index.html.
[11] 2018. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.
[12] 2018. OpenCL Extenstion, Khronos Group. https://www.khronos.org/registry/OpenCL/extensions/amd/cl_amd_device_attribute_query.txt.
[13] 2018. OpenCL Overview, Khronos Group. https://www.khronos.org/opencl/.
[14] 2018. OpenGL Extension, ATI. https://www.khronos.org/registry/OpenGL/extensions/ATI/ATI_meminfo.txt.
[15] 2018. OpenGL Extenstion, Khronos Group. https://www.khronos.org/registry/OpenGL/extensions/NVX/NVX_gpu_memory_info.txt.
[16] 2018. OpenGL Extenstion, Khronos Group. https://www.khronos.org/registry/OpenGL/extensions/NV/NV_shader_thread_group.txt.
[17] 2018. OpenGL Overview, Khronos Group. https://www.khronos.org/opengl/.
[18] 2018. Tegra Graphics Debugger, Nvidia. https://developer.nvidia.com/tegra-graphics-debugger.
[19] 2018. Vulkan Overview, Khronos Group. https://www.khronos.org/vulkan/.
[20] 2018. WebGL Overview, Khronos Group. https://www.khronos.org/webgl/.
[21] 2018. WebGL Security, Khronos Group. https://www.khronos.org/webgl/security/.
[22] 2018. WebGL Statistics. http://webglstats.com/.
[23] Onur Aciiçmez, Billy Bob Brumley, and Philipp Grabher. 2010. New Results on Instruction Cache Attacks. In *Proceedings of the 12th International Conference on Cryptographic Hardware and Embedded Systems (CHES'10)*. Springer-Verlag, Berlin, Heidelberg, 110–124. http://dl.acm.org/citation.cfm?id=1881511.1881522
[24] Ahmed Osama Fathy Atya, Zhiyun Qian, Srikanth V. Krishnamurthy, Thomas La Porta, Patrick McDaniel, and Lisa Marvel. 2017. Malicious co-residency on the cloud: Attacks and defense. In *IEEE Conference on Computer Communications (INFOCOM'17)*. 1–9. https://doi.org/10.1109/INFOCOM.2017.8056951
[25] Andrea Di Biagio, Alessandro Barenghi, Giovanni Agosta, and Gerardo Pelosi. 2009. Design of a Parallel AES for Graphic Hardware using the CUDA framework. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09)*. IEEE, Rome Italy. https://doi.org/0.1109/IPDPS.2009.5161242
[26] Andrew Bortz and Dan Boneh. 2007. Exposing Private Information by Timing Web Applications. In *Proceedings of the 16th International Conference on World Wide Web (WWW '07)*. ACM, New York, NY, USA, 621–628. https://doi.org/10.1145/1242572.1242656
[27] Prima Chairunnanda, Num Pham, and Urs Hengartner. 2011. Privacy: Gone with the Typing! Identifying Web Users by Their Typing Patterns. In *IEEE International Conference on Privacy, Security, Risk and Trust*. 974–980.
[28] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC '09)*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797
[29] Qi Alfred Chen, Zhiyun Qian, and Zhuoqing Morley Mao. 2014. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *USENIX Security Symposium*. 1037–1052. https://dl.acm.org/citation.cfm?id=2671291
[30] N. L. Clarke and S. M. Furnell. 2006. Authenticating mobile phone users using keystroke analysis. *International Journal on Information Security* 6 (Dec. 2006). https://dl.acm.org/citation.cfm?id=1201686
[31] Renan Correa Detomini, Renata Spolon Lobato, Roberta Spolon, and Marcos Antonio Cavenaghi. 2011. Using GPU to exploit parallelism on cryptography. In *6th Iberian Conference on Information Systems and Technologies (CISTI'11)*. IEEE, Chaves Portugal. http://ieeexplore.ieee.org/document/5974171
[32] Luke Durant. 2017. Inside Volta. Presentation at GPU-Tech. Accessed online Feb. 2018 from http://on-demand.gputechconf.com/gtc/2017/presentation/s7798-luke-durant-inside-volta.pdf.
[33] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking Branch Predictors to Bypass ASLR. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. 1–13.
[34] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 693–707. https://doi.org/10.1145/3173162.3173204
[35] Edward W. Felten and Michael A. Schneider. 2000. Timing Attacks on Web Privacy. In *Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS '00)*. ACM, New York, NY, USA, 25–32. https://doi.org/10.1145/352600.352606
[36] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *Proceedings of IEEE Symposium on Security and Privacy*. 357–372. https://doi.org/10.1109/SP.2018.00022
[37] Berk Gulmezoglu, Andreas Zankl, Thomas Eisenbarth, and Berk Sunar. 2017. PerfWeb: How to Violate Web Privacy with Hardware Performance Events. In *Computer Security – ESORICS 2017*, Simon N. Foley, Dieter Gollmann, and Einar Snekkenes (Eds.). Springer International Publishing, Cham, 80–97.
[38] Jamie Hayes and George Danezis. 2016. k-fingerprinting: A Robust Scalable Website Fingerprinting Technique. In *USENIX Security Symposium*. 1187–1203.
[39] Suman Jana and Vitaly Shmatikov. 2012. Memento: Learning Secrets from Process Footprints. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society, Washington, DC, USA, 143–157. https://doi.org/10.1109/SP.2012.19
[40] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. 2016. A complete key recovery timing attack on a GPU. In *IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. IEEE, Barcelona Spain, 394–405. https://doi.org/10.1109/HPCA.2016.7446081
[41] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. 2017. A Novel Side-Channel Timing Attack on GPUs. In *Proceedings of the on Great Lakes Symposium on VLSI (VLSI'17)*. 167–172. https://doi.org/10.1145/3060403.3060462
[42] Mehmet Kayaalp, Khaled N Khasawneh, Hodjat Asghari Esfeden, Jesse Elwell, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. 2017. RIC: Relaxed Inclusion Caches for Mitigating LLC Side-Channel Attacks. In *Proceedings of the 54th Annual Design Automation Conference (DAC)*. https://dl.acm.org/citation.cfm?id=3062313
[43] Mehmet Kayaalp, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Aamer Jaleel. 2016. A high-resolution side-channel attack on last-level cache. In *53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2897937.2897962
[44] Ron Kohavi. 1995. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *International Joint Conference on Artificial Intelligence (IJCAI)*, Vol. 14. 1137–1145. https://dl.acm.org/citation.cfm?id=1643047
[45] Robert Kotcher, Yutong Pei, Pranjal Jumde, and Collin Jackson. 2013. Cross-origin pixel stealing: timing attacks using CSS filters. In *ACM Conference on Computer and Communications Security*. 1055–1062. https://doi.org/10.1145/2508859.2516712
[46] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. 2014. Stealing Webpage Rendered on your Browser by Exploiting GPU Vulnerabilities. In *IEEE Symposium on Security and Privacy (SP'14)*. IEEE, San Jose CA USA, 19–33. https://doi.org/10.1109/SP.2014.9
[47] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. 2017. Practical Keystroke Timing Attacks in Sandboxed

JavaScript. In *Computer Security – ESORICS 2017*, Simon N. Foley, Dieter Gollmann, and Einar Snekkenes (Eds.). Springer International Publishing, Cham, 191–209. https://link.springer.com/chapter/10.1007/978-3-319-66399-9_11

[48] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy (SP'15)*. IEEE, San Jose, CA, USA. https://doi.org/10.1109/SP.2015.43

[49] Chao Luo, Yunsi Fei, Pei Luo, Saoni Mukherjee, and David Kaeli. 2015. Side-Channel Power Analysis of a GPU AES Implementation. In *33rd IEEE International Conference on Computer Design (ICCD'15)*. https://doi.org/10.1109/ICCD.2015.7357115

[50] Robert Martin, John Demme, and Simha Sethumadhavan. 2012. TimeWarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *39th Annual International Symposium on Computer Architecture (ISCA'12)*. Portland, OR, USA, 118–129. https://doi.org/10.1109/ISCA.2012.6237011

[51] Clémentine Maurice, Christoph Neumann, Olivier Heen, and AurÃlien Francillon. 2014. Confidentiality Issues on a GPU in a Virtualized Environment. In *International Conference on Financial Cryptography and Data Security*. 119–135. https://link.springer.com/chapter/10.1007/978-3-662-45472-5_9

[52] Wen mei Hwu. 2011. *GPU Computing Gems* (1st. ed.). Elsevier.

[53] Fabian Monrose and Aviel Rubin. 1997. Authentication via keystroke dynamics. In *ACM International Conference on Computer and Communication Security (CCS)*.

[54] Hoda Naghibijouybari and Nael Abu-Ghazaleh. 2016. Covert Channels on GPGPUs. *IEEE Computer Architecture Letters* 16, 1 (2016), 22–25. https://ieeexplore.ieee.org/document/7509650/

[55] Hoda Naghibijouybari, Khaled Khasawneh, and Nael Abu-Ghazaleh. 2017. Constructing and Characterizing Covert Channels on GPUs. In *Proc. International Symposium on Microarchitecture (MICRO)*. 354–366.

[56] Alex Nanopoulos, Rob Alcock, and Yannis Manolopoulos. 2001. Feature-based classification of time-series data. *International Journal of Computer Research* 10, 3 (2001), 49–61. https://dl.acm.org/citation.cfm?id=766918

[57] Naoki Nishikawa, Keisuke Iwai, and Takakazu Kurokawa. 2011. High-performance symmetric block ciphers on CUDA. In *Second International Conference on Networking and Computing (ICNC'11)*. Osaka Japan, 221–227. https://doi.org/10.1109/ICNC.2011.40

[58] Lena E. Olson, Jason Power, Mark D. Hill, and David A. Wood. 2015. Border Control: Sandboxing Accelerators. In *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*. Waikiki HI USA, 470–481. https://doi.org/10.1145/2830772.2830819

[59] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. 2015. 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)*. ACM, New York, NY, USA, 1406–1418. https://doi.org/10.1145/2810103.2813708

[60] Andriy panchenko, Fabian Lanze, Andreas Zinnen, Martin Henze, Jan Pennekamp, Klaus Wehrle, and Thomas Engel. 2016. Website Fingerprinting at Internet Scale. In *23rd Internet Society (ISOC) Network and Distributed System Security Symposium (NDSS 2016)*. https://doi.org/10.14722/ndss.2016.23477

[61] Alen Peacock, Xian Ke, and Matthew Wilkerson. 2004. Typing patterns: A key to user identification. *IEEE Security and Privacy* 2 (2004), 40–47.

[62] Colin Percival. 2005. Cache missing for fun and profit. In *BSDCan*. https://doi.org/10.1.1.144.872

[63] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. 2016. CUDA leaks: Information Leakage in GPU Architecture. *ACM Transactions on Embedded Computing Systems (TECS)* 15, 1 (2016). https://doi.org/10.1145/2801153

[64] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. 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)*. Chicago, Illinois, USA, 199–212. https://doi.org/10.1145/1653662.1653687

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

[66] Paul Stone. 2013. Pixel Perfect Timing Attacks with HTML5. https://www.contextis.com/resources/white-papers/pixel-perfect-timing-attacks-with-html5.

[67] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2016. Stealing Machine Learning Models via Prediction APIs. In *25th USENIX Security Symposium (USENIX Security)*. 601–618.

[68] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. 2015. 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)*. ACM, New York, NY, USA, 1382–1393. https://doi.org/10.1145/2810103.2813632

[69] Pepe Vila and Boris Kopf. 2017. Loophole: Timing Attacks on Shared Event Loops in Chrome. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 849–864. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/vila

[70] Zachary Weinberg, Eric Y. Chen, Pavithra Ramesh Jayaraman, and Collin Jackson. 2011. 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)*. IEEE Computer Society, Washington, DC, USA, 147–161. https://doi.org/10.1109/SP.2011.23

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

[72] Xiaokuan Zhang, Xueqiang Wang, Xiaolong Bai, Yinqian Zhang, and XiaoFeng Wang. 2018. OS-level Side Channels without Procfs: Exploring Cross-App Information Leakage on iOS. In *Proceedings of the Symposium on Network and Distributed System Security*.

[73] Zhe Zhou, Wenrui Diao, Xiangyu Liu, Zhou Li, Kehuan Zhang, and Rui Liu. 2017. Vulnerable GPU Memory Management: Towards Recovering Raw Data from GPU. In *Proceedings on Privacy Enhancing Technologies*, Vol. 2017 (2). 57–73. https://doi.org/10.1515/popets-2017-0016

[74] Zhiting Zhu, Sangman Kim, Yuri Rozhanski, Yige Hu, Emmett Witchel, and Mark Silberstein. 2017. Understanding the Security of Discrete GPUs. In *Proceedings of the General Purpose GPUs (GPGPU'10)*. Austin TX USA, 1–11. https://doi.org/0.1145/3038228.3038233