This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2944624, IEEE Transactions on Dependable and Secure Computing

Side Channel Attacks on GPUs

Hoda Naghibijouybari, *Student Member, IEEE,* Ajaya Neupane, *Member,IEEE,* Zhiyun Qian, *Member,IEEE,* and Nael Abu-Ghazaleh, *Senior Member, IEEE*

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 by implementing three end-to-end attacks. We show that an OpenGL or CUDA based spy can fingerprint websites accurately (attack I), track user activities within the website, and even infer the keystroke timings for a password text box (attack II) with high accuracy. The third attack demonstrates how a CUDA spy application can derive the internal parameters of a neural network model being used by another CUDA application 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.

Index Terms—GPU, side channels, website fingerprinting, keystroke timing attack.

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 [1]. Both these classes of applications can operate on sensitive data [2], [3] which can be compromised by security vulnerabilities.

Although the security of GPUs is only starting to be explored, several vulnerabilities have already been demonstrated [4], [5], [6], [7], [8], [9]. 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 run-time 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. Jiang et al. [10] assume that the attacker needs to launch the encryption kernel on GPU and measure the whole kernel execution time on its own process (on CPU side), which is a different threat model than ours which investigates side channel between two concurrent apps on the GPU. 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 [11]) on a GPU may be constructed [8], [12]. Neither of these papers demonstrates a general side channel attack. This paper extends our prior work [13] which showed that indeed side channels are present and exploitable, and demonstrated attacks on a range of Nvidia GPUs and for both graphics and computational software stacks and applications.

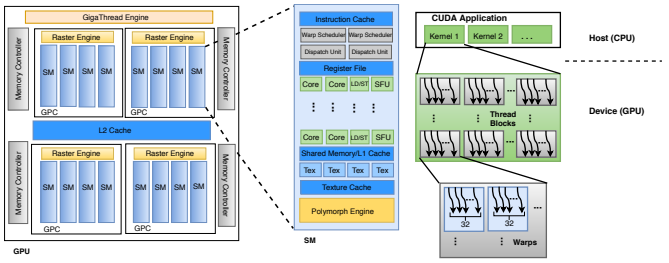
We systematically characterize the situations where a spy can co-locate and measure side channel behavior of a victim in both the graphics and the computational stacks of the Nvidia family of GPUs. For OpenGL workloads, we discover that kernels (shader programs) can be concurrently scheduled provided there are sufficient resources to support them on the GPU. 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. Additionally, we study possibility of attacks on integrated GPUs and also different operating systems.

To illustrate attacks on graphics applications, we implement a web-fingerprinting attack that first detects the browser window size and then identifies user browsing websites with high accuracy. We show an extension to

• *Computer Science and Engineering Department, University of California, Riverside. E-mail: naelag@ucr.edu*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2944624, IEEE Transactions on Dependable and Secure Computing



(a) (b)

Fig. 1: GPU overview (a) Architecture; (b) Application

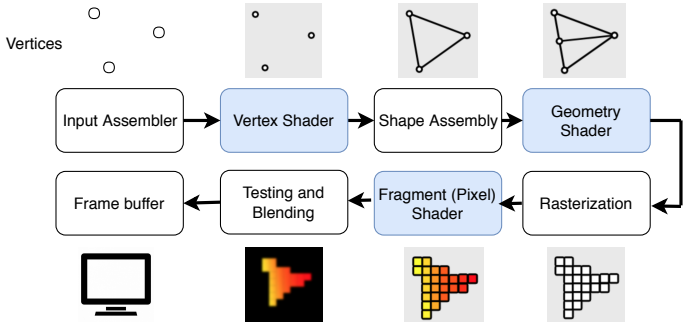


Fig. 2: Graphics processing pipeline

this attack that tracks user activity on a website, and captures keystroke timing. We generalize the attack to work without assuming a specific browser window size. 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.

Disclosure: We have reported all of our findings to Nvidia, who published a security advisory and applied for a vulnerability CVE string (CVE−2018−6260). 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 Interfaces and Architecture

This section overviews GPU programming interfaces and GPU architecture to explain how they are programmed, and how contention arises within them.

2.1 GPU Programming Interfaces

GPUs were originally designed to accelerate graphics workloads. They are programmed using application programming interfaces such as OpenGL for 2D/3D graphics [14], or WebGL [15] 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 [11] and OpenCL [16]. 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 [17]. 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 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 [11], and there are already thousands of applications available for it on desktops, and mobile devices.

2.2 GPU Architecture Overview

Figure 1a presents an architecture overview of a typical 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). There is a globally shared L2 cache to provide faster access to memory.

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, threads are grouped into *warps* of typically 32 threads that are scheduled together using the Single Instruction Multiple Thread (SIMT) 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,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2944624, IEEE Transactions on Dependable and Secure Computing

load/store cores and special functional units. The cores are heavily pipelined making it possible to continue to issue new instructions to them in different cycles.

The GPU memory is shared across all the SMs and is connected to the chip using several high speed channels, 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 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.

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

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 [8]. 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 [18].

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 Nvidia GPUs [19] 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 [20] and on Linux and Android [21], [22] 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2944624, IEEE Transactions on Dependable and Secure Computing

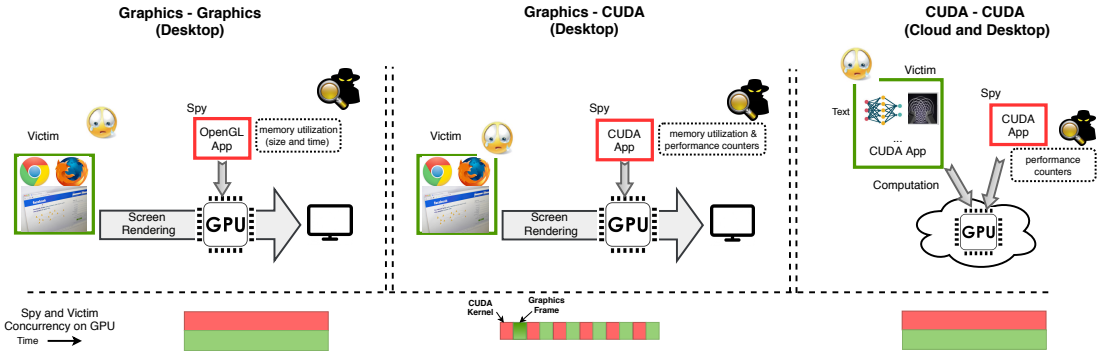


Fig. 3: Threat Models

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 [19] 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 [20] to monitor the GPU performance counters from a CUDA spy. Table 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 work 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" [23] 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 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.

TABLE 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

4 Attack Models 1 and 2: Graphics/CUDA Spy and 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 3, 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2944624, IEEE Transactions on Dependable and Secure Computing

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" [19] which enable programmers to query the ThreadID, the WarpID and the SM-ID in OpenGL shader codes and "ARB_shader_clock" [19] 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 [15]. 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. As a result, the Chromium Project's GPU Architecture Roadmap [24] 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 later in this section.

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 4 shows the GPU memory allocation trace when Google and

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.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2944624, IEEE Transactions on Dependable and Secure Computing

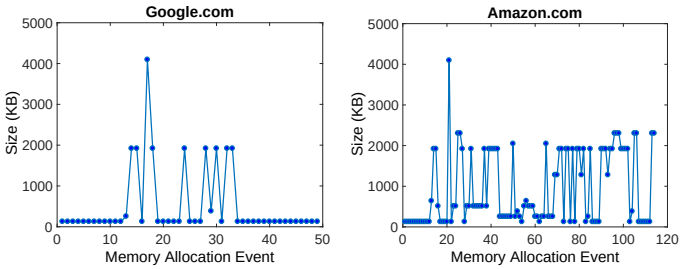


Fig. 4: Website memory allocation on GPU (a) Google; (b) Amazon

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 [25]. 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 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 [26]. 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 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)

Table2 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%.

Website fingerprinting from CUDA Spy using performance counters We also demonstrate a threat model 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. In our prior work [13], we reported a detailed explanation of our reverse engineering method to identify co-location of Spy and Victim and implementing the attack. In this subsection, we briefly discuss the attack and the results. 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. 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. The average precision of the random forest classifier model on correctly classifying the websites is 93.0% (f-measure of 92.7%), which represents excellent accuracy in website fingerprinting.

4.2 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-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2944624, IEEE Transactions on Dependable and Secure Computing

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, 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.

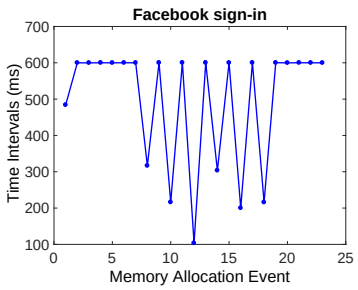


Fig. 5: Timing memory allocations: 6-character password

Prior work has shown that inter-arrival times of keystrokes can leak information about the actually characters being typed by the user [27]. 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 6 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 7 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.

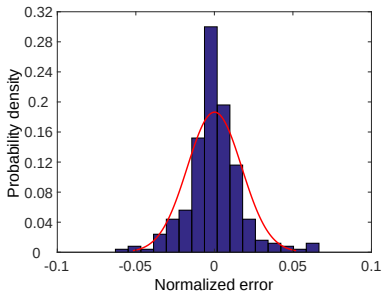


Fig. 6: Error distribution of inter-keystroke time

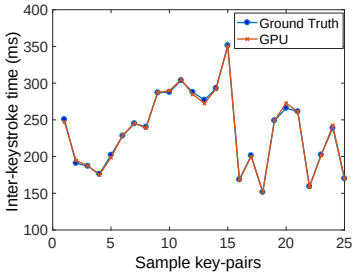


Fig. 7: Keystroke timing: Ground Truth vs. GPU

5 Attack Model 3: CUDA Spy and 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. Modern GPUs support multiprogramming through multiple hardware streams with multi-kernel execution using a multi-process service (MPS) [28], 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2944624, IEEE Transactions on Dependable and Secure Computing

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 [29], [30]. 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.

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 [31]; 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 [8] 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 3 summarizes the most top ranked features selected in the classification and Figure 8 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 3: 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

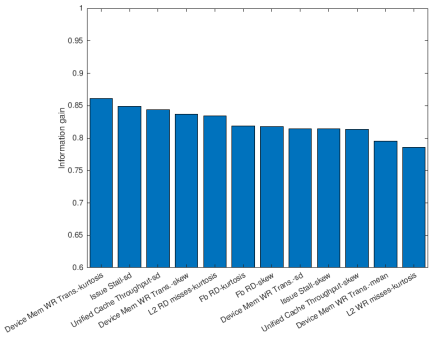


Fig. 8: Information gain of top features

TABLE 4: 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 4 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.

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2944624, IEEE Transactions on Dependable and Secure Computing

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.

6 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. 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: 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 6. With this improvement, we believe that the attacks become robust to changes in window size.

TABLE 6: 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)

7 Attack 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 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. Xu et al. [32] proposed a GPU-specific intra-SM partitioning scheme to isolate con-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2944624, IEEE Transactions on Dependable and Secure Computing

tention 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.

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 9a, and with reducing the granularity in Figure 9b. 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 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.

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.

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.

Related Work to Our Attacks: Different attack vectors have been proposed for website fingerprinting. Panchenko et al. [33] and Hayes et al. [39] capture traffic generated via loading monitored web pages. Felten and Schneider [40] utilize browser caching and construct a timing channel to infer the victim visited websites. Jana and Shmatikov [34] 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. [41] presented a user interaction attack (victim's action on website leaks its browsing history) and a timing side channel attack for browser history sniffing. Table 7 compares the classification accuracy of our memory based and performance counter based attacks to other previously published website-fringerprinting attacks. All attacks, other than [4] 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.

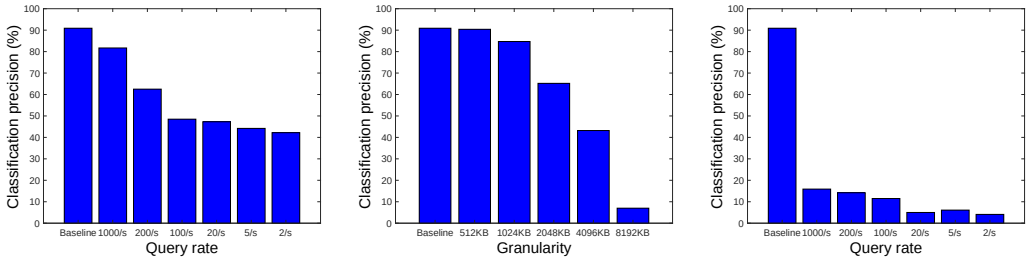
Some of the principles used in our attack have also been leveraged by other researchers. Goethem et al. [42] and Bortz et al. [43] propose cross-site timing attacks on web browsers to estimate the size of cross-origin resources or provide user information leakage from other site. [44] and [45] propose timing channels by measuring the time elapsed between frames using a JavaScript API.These attacks are difficult currently since most browsers have reduced the timer resolution eliminating the timing signal used by the attacks. Gulmezoglu et al. [37] proposed a side channel on per-core/per-process CPU hardware performance counters (which are limited in number).

Leakage through the keystroke timing pattern is also a known effect which has been exploited as a user authentication mechanism [46], [47]. Keystroke timing has also been used to compromise/weaken passwords [27] or compromise user privacy [48]. Lipp et al. [49] 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.

Side Channel Attacks on GPUs: 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. Jiang et al. [10] 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. The same group [50] 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 [10] cannot be used for a side-channel between two concurrent applications. Luo et al. study a power side channel attack on AES encryption executing on a GPU [6]. This attack requires physical access to the GPU to measure the power. Naghibijouybari et al. [8] 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 paper extends our prior work [13] which demonstrated three end-to-end side channel attacks on Nvidia GPUs: Website fingerprinting, keystroke monitoring and, neural network model extraction attack. In this paper, we generalize website fingerprinting attack across different window sizes, so that the attacker utilizes a side channel signal to detect the browser window size, before imple-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2019.2944624, IEEE Transactions on Dependable and Secure Computing



(a) (b) (c)

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

TABLE 7: 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 (94)	200 (100)	Chrome
PC-based attack	side channel (GPU performance counters)	93	200	Chrome
[33]	side channel (traffic analysis)	92.52	100	Tor
[34]	side channel (memory footprint via procfs)	78	100	Chrome
[35]	side channel (LLC)	82.1 (88.6)	8	Safari (Tor)
[36]	side channel (shared event loop)	76.7	500	Chrome
[37]	side channel (CPU performance counters)	84	30	Chrome
[38]	side channel (iOS APIs)	68.5	100	Safari
[4]	leftover memory on GPU	95.4	100	Chrome

menting website fingerprinting attack. We also study the feasibility of attacks on integrated GPUs such as Intel Graphics and portability of attacks to other operating systems. In neural network model extraction attack, we analyze the top features in classification and also discuss possibility of attack in interleaved kernel execution model.

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 models and also identify different ways to measure leakage. We demonstrate a series of end-to-end GPU side channel attacks covering the different threat models on both graphics and computational stacks, as well as across them. 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 work in this paper is supported by the National Science Foundation under Grant No.:CNS-1619450.

References

[1] W. mei Hwu, *GPU Computing Gems*, 1st ed. Elsevier, 2011.
[2] A. D. Biagio, A. Barenghi, G. Agosta, and G. Pelosi, "Design of a parallel aes for graphic hardware using the cuda framework," in *IEEE International Symposium on Parallel & Distributed Processing*, ser. IPDPS'09. Rome Italy: IEEE, 2009.
[3] R. C. Detomini, R. S. Lobato, R. Spolon, and M. A. Cavenaghi, "Using gpu to exploit parallelism on cryptography," in *6th Iberian Conference on Information Systems and Technologies*, ser. CISTI'11. Chaves Portugal: IEEE, 2011.
[4] S. Lee, Y. Kim, J. Kim, and J. Kim, "Stealing webpage rendered on your browser by exploiting gpu vulnerabilities," in *IEEE Symposium on Security and Privacy*, ser. SPI'14. San Jose CA USA: IEEE, 2014, pp. 19–33.
[5] R. D. Pietro, F. Lombardi, and A. Villani, "Cuda leaks: Information leakage in gpu architecture," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 15, no. 1, 2016.
[6] C. Luo, Y. Fei, P. Luo, S. Mukherjee, and D. Kaeli, "Side-channel power analysis of a gpu aes implementation," in *33rd IEEE International Conference on Computer Design*, ser. ICCD'15, 2015.
[7] Z. Zhu, S. Kim, Y. Rozhanski, Y. Hu, E. Witchel, and M. Silberstein, "Understanding the security of discrete gpus," in *Proceedings of the General Purpose GPUs*, ser. GPGPU'10, 2017.
[8] H. Naghibijouybari, K. Khasawneh, and N. Abu-Ghazaleh, "Constructing and characterizing covert channels on gpus," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2017.
[9] Z. Yao, Z. Ma, A. Sani, and A. Chandramowlishwaran, "Sugar: Secure GPU acceleration in web browsers," in *Proc. International Conference on Architecture Support for Operating Systems and Programming Languages (ASPLOS)*, 2018.
[10] Z. H. Jiang, Y. Fei, and D. Kaeli, "A complete key recovery timing attack on a gpu," in *IEEE International Symposium on High Performance Computer Architecture*, ser. HPCA'16. Barcelona Spain: IEEE, 2016, pp. 394–405.
[11] "CUDA, Nvidia," 2018, https://developer.nvidia.com/cuda-zone/.
[12] H. Naghibijouybari and N. Abu-Ghazaleh, "Covert channels on gpgpus," *IEEE Computer Architecture Letters*, vol. 16, no. 1, pp. 22–25, 2016.

[13] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, "Rendered insecure: Gpu side channel attacks are practical," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, 2018, pp. 2139–2153.

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

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

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

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

[18] M. Kayaalp, D. Ponomarev, N. Abu-Ghazaleh, and A. Jaleel, "A high-resolution side-channel attack on last-level cache," in *53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2016, pp. 1–6.

[19] "OpenGL Extenstion, Khronos Group," 2018, https://www.khronos.org/registry/OpenGL/index_gl.php.

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

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

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

[23] "OpenGL Extension, Intel," 2018, https://www.khronos.org/registry/OpenGL/extensions/INTEL/INTEL_performance_query.txt.

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

[25] "Alexa Top Sites," 2018, https://www.alexa.com/topsites.

[26] A. Nanopoulos, R. Alcock, and Y. Manolopoulos, "Feature-based classification of time-series data," *International Journal of Computer Research*, vol. 10, no. 3, pp. 49–61, 2001.

[27] D. X. Song, D. Wagner, and X. Tian, "Timing analysis of keystrokes and timing attacks on SSH," in *Proc. USENIX Security Symposium*, 2001.

[28] "Multi-Process Service, Nvidia," 2018, https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[29] T. Ristenpart, E. Tromer, H. Shacham, and S. 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*, ser. CCS'09, Chicago, Illinois, USA, 2009, pp. 199–212.

[30] A. O. F. Atya, Z. Qian, S. V. Krishnamurthy, T. L. Porta, P. McDaniel, and L. Marvel, "Malicious co-residency on the cloud: Attacks and defense," in *IEEE Conference on Computer Communications*, ser. INFOCOM'17, 2017, pp. 1–9.

[31] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, ser. IISWC '09, 2009, pp. 44–54.

[32] Q. Xu, H. Naghibijouybari, S. Wang, N. Abu-Ghazaleh, and M. Annavaram, "Gpuguard: Mitigating contention based side and covert channel attacks on gpus," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19. New York, NY, USA: ACM, 2019, pp. 497–509. [Online]. Available: http://doi.acm.org/10.1145/3330345.3330389

[33] A. panchenko, F. Lanze, A. Zinnen, M. Henze, J. Pennekamp, K. Wehrle, and T. Engel, "Website fingerprinting at internet scale," in *23rd Internet Society (ISOC) Network and Distributed System Security Symposium (NDSS 2016)*, 2016.

[34] S. Jana and V. Shmatikov, "Memento: Learning secrets from process footprints," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12, 2012, pp. 143–157.

[35] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. 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*, ser. CCS '15, 2015, pp. 1406–1418.

[36] P. Vila and B. Kopf, "Loophole: Timing attacks on shared event loops in chrome," in *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC, 2017, pp. 849–864.

[37] B. Gulmezoglu, A. Zankl, T. Eisenbarth, and B. Sunar, "Perfweb: How to violate web privacy with hardware performance events," in *Computer Security – ESORICS 2017*. Cham: Springer International Publishing, 2017, pp. 80–97.

[38] X. Zhang, X. Wang, X. Bai, Y. Zhang, and X. Wang, "Os-level side channels without procfs: Exploring cross-app information leakage on ios," in *Proceedings of the Symposium on Network and Distributed System Security*, 2018.

[39] J. Hayes and G. Danezis, "k-fingerprinting: A robust scalable website fingerprinting technique," in *USENIX Security Symposium*, 2016, pp. 1187–1203.

[40] E. W. Felten and M. A. Schneider, "Timing attacks on web privacy," in *Proceedings of the 7th ACM Conference on Computer and Communications Security*, ser. CCS '00, 2000, pp. 25–32.

[41] Z. Weinberg, E. Y. Chen, P. R. Jayaraman, and C. 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*, ser. SP '11, 2011, pp. 147–161.

[42] T. Van Goethem, W. Joosen, and N. 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*, ser. CCS '15, 2015, pp. 1382–1393.

[43] A. Bortz and D. Boneh, "Exposing private information by timing web applications," in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW '07, 2007, pp. 621–628.

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

[45] R. Kotcher, Y. Pei, P. Jumde, and C. Jackson, "Cross-origin pixel stealing: timing attacks using css filters," in *ACM Conference on Computer and Communications Security*, 2013, pp. 1055–1062.

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

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

[48] 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*, Oct. 2011, pp. 974–980.

[49] M. Lipp, D. Gruss, M. Schwarz, D. Bidner, C. Maurice, and S. Mangard, "Practical keystroke timing attacks in sandboxed javascript," in *Computer Security – ESORICS 2017*, S. N. Foley, D. Gollmann, and E. Snekkenes, Eds. Cham: Springer International Publishing, 2017, pp. 191–209.

[50] Z. H. Jiang, Y. Fei, and D. Kaeli, "A novel side-channel timing attack on gpus," in *Proceedings of the on Great Lakes Symposium on VLSI*, ser. VLSI'17, 2017, pp. 167–172.

Hoda Naghibijouybari is a PhD student in the Department of Computer Science and Engineering at the University of California, Riverside. Her research interests are in architecture support for security, including GPU security, and side channel attacks and defenses.

Ajaya Neupane is a Postdoctoral Researcher at the University of California Riverside. His interests are in network and system security. He received his PhD in Computer and Information Sciences from the University of Alabama at Birmingham in 2017.

Zhiyun Qian is an associate professor in the Computer Science and Engineering department at the University of California Riverside. His research interest is on system and network security. He received the PhD degree in Computer Science and Engineering from University of Michigan in 2012.

Nael Abu-Ghazaleh is a Professor in the Computer Science and Engineering department and the Electrical and Computer Engineering department at the University of California at Riverside. His research interests are in the areas of computer architecture support for security, parallel discrete event simulation, networking and mobile computing. He received his PhD from the University of Cincinnati in 1997.