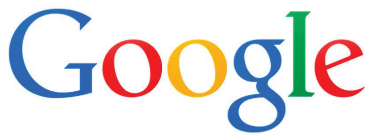
Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities

Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim
Department of Computer Science and Engineering
POSTECH, Korea
{*sangho2, elixir, jangwoo, jkim*}*@postech.ac.kr*

***Abstract*—Graphics processing units (GPUs) are important components of modern computing devices for not only graphics rendering, but also efficient parallel computations. However, their security problems are ignored despite their importance and popularity. In this paper, we first perform an in-depth security analysis on GPUs to detect security vulnerabilities. We observe that contemporary, widely-used GPUs, both NVIDIA's and AMD's, do not initialize newly allocated GPU memory pages which may contain sensitive user data. By exploiting such vulnerabilities, we propose attack methods for revealing a victim program's data kept in GPU memory both during its execution and right after its termination. We further show the high applicability of the proposed attacks by applying them to the Chromium and Firefox web browsers which use GPUs for accelerating webpage rendering. We detect that both browsers leave rendered webpage textures in GPU memory, so that we can infer which webpages a victim user has visited by analyzing the remaining textures. The accuracy of our advanced inference attack that uses both pixel sequence matching and RGB histogram matching is up to 95.4%.**


(a) Google logo image.

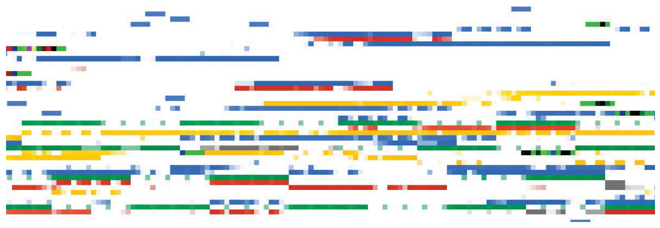

(b) Partial dump of Google webpage textures.

Figure 1. Google logo and partial dump of Google webpage textures extracted from the Chromium web browser with an NVIDIA GPU.

I. Introduction

This work considers how attackers can disclose sensitive data kept in graphics processing unit (GPU) memory. We aim to obtain *rendered webpage textures* to uncover webpages a victim user has visited. We successfully reveal such data from modern GPUs (e.g., NVIDIA and AMD GPUs) when we enable GPU-accelerated webpage rendering of recent web browsers (e.g., Chromium and Firefox). For example, Figure 1 shows the Google logo image of http://google.com and a partial dump of rendered webpage textures extracted from an NVIDIA GPU used by the Chromium web browser. Although the GPU has rearranged the textures according to its undocumented hardware characteristics, we can infer that the dump originates from the webpage because their color patterns are similar. Especially, our combined matching attack can successfully infer up to 95.4% of randomly visited 100 front pages of Alexa Top 1000 websites when a victim uses the Chromium web browser with an NVIDIA GPU (details are in Section V.)

GPUs are important and powerful components of contemporary computing devices. Personal computing devices, including desktops, laptops, and smartphones, use GPUs for supporting various graphics applications, e.g., graphical user interface (GUI), multimedia players, and video games. Large-scale computing devices, including workstations, servers, and clusters, also use GPUs for energy-efficient, massive parallel computations. GPUs utilize a large number of processing cores and a large amount of independent memory for efficiently processing graphics operations and computational workloads. For example, an NVIDIA Kepler GPU can have up to 2880 cores and 6 GB of memory, and its floating-point operation performance is nine times better than that of the recent CPUs [1], [2].

Programmers can use two types of application programming interfaces (APIs) to access GPUs: graphics APIs (e.g., DirectX [3] and OpenGL [4]) and computing APIs (e.g., CUDA [2] and OpenCL [5]). First, the graphics APIs provide functions for graphics operations, such as projection, shading, and texture mapping. Second, the computing APIs provide functions for non-graphics applications, such as financial, medical, or weather data analyses [6], database query optimizations [7], [8], packet routing [9], intrusion detection systems [10], [11], and cryptographic engines [12]–[17].

The most significant differences between the graphics APIs and the computing APIs are sharing and memory manageability. First, the computing APIs allow different users to share the same GPU, whereas the graphics APIs only support a single user. A number of users can share the same GPU using the computing APIs in a time-sharing fashion, as (1) the computing APIs demand no dedicated screens and (2) current GPUs only support sequential execution of different GPU processes [18]. Although some techniques (e.g., VirtualGL [19]) allow remote users to share the same GPU when using the graphics APIs, they warn users of potential security problems (e.g., logging keystrokes and reading back images through an X server).

Second, while GPU drivers manage GPU memory with the graphics APIs, programmers can manually manage GPU memory with the computing APIs, including allocations, CPU-GPU data transfers, and deallocations. GPUs have several types of memory (e.g., global, local, and private memories), and programmers can control them using the computing APIs except some graphics-related memories (e.g., framebuffer and z-buffer). In contrast, the graphics APIs provide no functions to manage such memories while providing a set of optimized functions to perform memory-efficient graphics operations.

Unfortunately, the sharing and high memory manageability of the computing APIs may incur critical security threats because GPUs do not initialize newly-allocated memory buffers [20]. Although numerous studies consider such an *uninitialized memory problem* in operating systems [21]–[24], no study deals with the uninitialized GPU memory problem[1]. If similar security threats exist with the computing APIs, the threats have much larger impact as multiple users share the same GPU.

In this paper, we first perform an in-depth security analysis on GPUs regarding their architectures and computing APIs to reveal any potential security threats. We identify that the computing APIs have a serious uninitialized memory problem because they (1) do not clear newly allocated memory pages, (2) have memory types that programmers cannot delete, and (3) have in-core memory without security mechanisms.

Second, we develop effective security attacks on GPUs applicable to the most widely used GPUs, NVIDIA and AMD GPUs, by exploiting the revealed security threats. Our attacks can disclose sensitive data kept in GPU memory of a victim program both during its execution and right after its termination.

Third, we demonstrate the high applicability of our attacks by inferring browsing history of the two most widely used web browsers, the Chromium and Firefox web browsers. Both browsers support GPU-accelerated webpage rendering acceleration, which uploads webpage textures to GPU memory to increase rendering speed. Our attacks can extract rearranged webpage textures of both browsers from NVIDIA and AMD GPUs.

Lastly, we propose advanced inference attacks which can correctly infer the original webpage of rearranged webpage textures with up to 95.4% accuracy. The proposed inference attacks compare the textures with either known textures or known webpage snapshots to identify the original webpage using three matching methods: (1) pixel sequence matching, (2) RGB histogram matching, and (3) combined matching.

To the best of our knowledge, our work is the first security analysis and attacks targeting GPUs. In summary, our work makes the following contributions:

- **Novel and crucial attack target.** Our work expands the scope of security research to a novel attack target, GPUs. Despite their popularity and importance, there is no in-depth and comprehensive study of their security problems before this work.
- **Complete in-depth study.** We present a complete in-depth security analysis on GPUs regarding not only their architectures, but also their computing APIs. We identify that all kinds of GPU memories accessible by the computing APIs has security problems.
- **Strong and widely-applicable attacks.** We demonstrate our attacks using the most widely-used GPUs and GPU-assisted applications: NVIDIA and AMD GPUs, and the Chromium and Firefox web browsers. Especially, our attacks accurately infer browsing history by up to 95.4%.

The remainder of this paper is organized as follows. In Section II we give an in-depth security analysis on GPUs and motivate our work. In Section III we explain the threat model. In Section IV we explain our attacks to disclose sensitive data remaining in GPU memory. In Section V we propose our inference attacks that identify browsing history of web browsers using GPUs. In Section VI we discuss possible countermeasures against the proposed attacks. In Section VII we summarize related work of this paper. Lastly, we conclude the paper in Section VIII.

II. Background and Security Concerns

In this section, we first give a brief overview on GPUs in terms of their architectures and programming models. Then, we motivate our work by presenting *inevitable* security concerns rising from the nature of GPUs. We use the OpenCL terminology [5] throughout the paper.

A. GPU Architecture

A GPU consists of (1) a *compute device* for executing instructions and (2) a *compute device memory* for storing data used by the compute device. Figure 2 shows the high-level diagram of a typical OpenCL-capable GPU architecture. The compute device consists of several *compute units (CUs)* and a *global/constant memory data cache* shared by all CUs. Each CU consists of *processing elements (PEs)*, also known as GPU cores, and per-CU *local memory* shared by the PEs. Each PE also has per-PE *private memory*.

The compute device memory consists of two memory types: *global memory* and *constant memory*. The read-write global memory is for storing GPU computation results. The

[1]We found unpublished [25] and concurrent [26] studies dealing with similar problems during the camera-ready period.

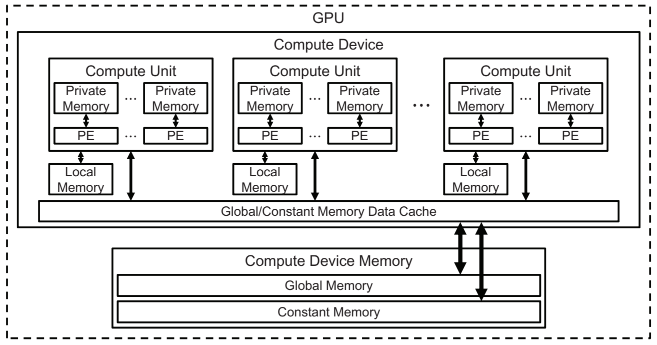


Figure 2. High-level architecture of an OpenCL-capable GPU.

read-only constant memory is for storing codes executed by the compute device and read-only data.

B. GPU Computing Model

In a typical GPU computing model, GPUs run the programmer-defined GPU *contexts*, similar to CPUs. Programmers construct a GPU context by writing *kernels*, *memory management*, and *command queues* using the computing APIs. Kernels are the functions written to run on GPUs. Memory management includes GPU memory allocations, data transfers between CPU and GPU memories, and GPU memory deallocations. Command queues hold the commands to be executed on GPUs. Both kernels and memory management are the commands. A typical execution of a GPU context is as follows: programmers queue (1) GPU memory allocation commands, (2) CPU to GPU data transfer commands, (3) kernel execution commands, (4) GPU to CPU data transfer commands, and (5) GPU memory deallocation commands in a command queue. When programmers queue a kernel execution command to the command queue, a GPU driver passes the pointers of allocated GPU memory accessible by the kernel to the GPUs through the kernel arguments. The GPUs execute all commands in a command queue in the first-in, first-out (FIFO) manner.

C. OpenGL and Textures

We briefly explain OpenGL, a popular graphics APIs widely supported by various operating systems, such as Windows, Linux, Android, and iOS. Among its several functions, we focus on texture functions as textures are sensitive image data of victims.

Texture mapping is a technique to make objects look realistic by mapping images or colors to 2D/3D objects. OpenGL provides a set of functions for texture mapping, such as `glGenTextures()` for generating a texture object, `glBindTexture()` for loading the texture object, and `glTexImage2D()` for specifying an image array of the texture object. Programmers either load an image file or generate some image data for `glTexImage2D()`. Once the texture object is uploaded to GPU memory, programmers can invoke `glTexCoord()` to coordinate the texture object while drawing objects.

Textures reside in the global memory, and programmers can delete textures no longer used by calling `glDeleteTextures()` to increase available global memory size. This function, however, does not initialize the corresponding memory blocks. Accordingly, an attacker can read the uninitialized textures which remain in the global memory after `glDeleteTextures()`.

D. Security Concerns

We now present three major security concerns of GPUs based on our analysis on GPU architectures and APIs. We take advantage of the major security concerns to perform our attacks presented in Section IV.

1) Lack of Memory Page Initialization Upon New Allocation: We identify a crucial security problem of GPU memory—GPUs do not initialize the contents of newly allocated memory pages that possibly contain sensitive data. The new owner can access the sensitive data remaining in the memory pages if the previous owner does not clear it. Modern operating systems have suffered from similar problems, but they solve the problems by filling new memory pages with zeros before delivering them to user space processes [27]. However, we detect that GPUs do not provide such a countermeasure. Therefore, we define the lack of memory page initialization upon new allocation as the first major security concern of GPUs.

2) Unerasable Memory: We identify that a portion of GPU memory is neither erasable by users nor automatically erased by GPUs. As explained in Section II-A, GPUs have several types of memory. While programmers can delete the contents of most types of memory, GPUs prevent programmers from erasing the contents of a few types of memory containing sensitive data (e.g., constant data, kernel codes, and call-by-value arguments). Moreover, we detect that GPUs do not automatically delete such contents even when they are no longer necessary. Thus, protecting sensitive data kept in the unerasable memory becomes the second major security concern of GPUs.

3) Programmer-managed Memory: We identify that threads originating from a kernel running on a CU can access the contents of other kernels, stored in the local and private memories of the CU. GPU computing models allow programmers to manually manage the local and private memories to optimize performance [2], [5]. For security, GPUs should disallow threads of a kernel to access the contents stored in the local and private memories, written by threads of other kernels. But we detect that current GPUs provide no isolation mechanism for the local and private memories. Regarding that GPU-accelerated applications use the local and private memories for storing sensitive data (e.g., secret keys with `libgpucrypto` [12]), lack of such a prevention becomes the third major security concern of GPUs.

III. System and Attack Models

We assume a system that equips a GPU for graphics operations and computations. The system is a multi-user system so that a number of users can share the equipped GPU. A *victim* is a normal user of the system who often executes programs using the GPU, such as 3D rendering software, web browsers, and financial data analysis tools. The victim occupies screens to locally use the graphics APIs. An *attacker* is another normal user of the system (a *local attacker*) who cannot attack the victim at the operating system level due to insufficient privilege as many attacks dealing with multi-user systems assume [28], [29]. The attacker, however, can freely access the GPU via the computing APIs, as any user of the system can use the APIs. Consequently, the attacker attempts to exploit the GPU to disclose the victim's sensitive data possibly remaining in GPU memory.

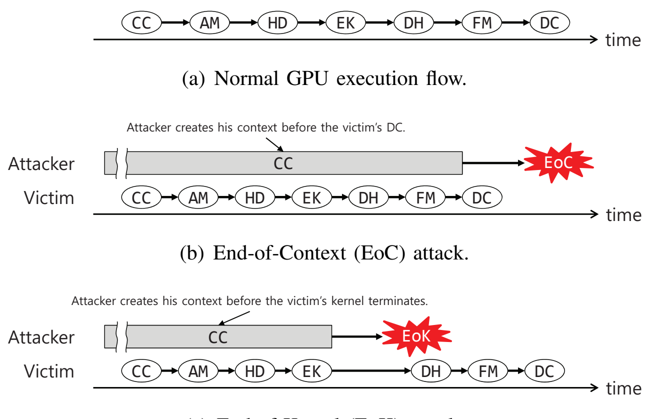
In addition, we consider a remote GPU system using VirtualGL [19], which is basically the same as the preceding system. Here, a victim has a right to use VirtualGL [19] to remotely use the graphics APIs, whereas an attacker has no right to use VirtualGL. Therefore, the attacker also need to exploit the GPU to attack the victim's data kept in GPU memory. As the attack methods for both systems are the same, we do not distinguish them in this work.

IV. Disclosing GPU Memory

In this section, we explain our attacks to disclose sensitive data in GPU memory exploiting the uninitialized memory problem. We propose two attacks to steal sensitive data of a victim program both at the right after its termination and during its execution. We further discuss real attacks on security-sensitive GPU programs in the later section.

A. Basic Attack

We identify that current GPUs have uninitialized memory problems by performing a basic attack that attempts to read the entire global memory after a victim GPU context terminates. First, we execute a simple victim program that writes 1 GB of `0x11111111` on the 3 GB of global memory of an NVIDIA GeForce GTX 780 GPU. Right after the victim program terminates, we execute a simple attack program that reads the entire global memory. When no active GPU program exists, the memory dump read by the attack program contains not the victim's data but dummy data. We expect that a GPU driver turns GPUs to a sleep mode for saving power, so that the data written in the global memory disappear. In contrast, when at least one active GPU program (e.g., a Gnome desktop) exists, the memory dump contains not only the 1 GB data written by the victim program, but also other important data, such as a kernel code. We



CC
AM
HD
EK
DH
FM
DC
time
Attacker creates his context before the victim's DC.
Attacker
Victim
EoC
Attacker creates his context before the victim's kernel terminates.
EoK

(a) Normal GPU execution flow.

(b) End-of-Context (EoC) attack.

(c) End-of-Kernel (EoK) attack.

Figure 3. Overview of proposed attacks.

expect that uncleared data remain in GPU memory, as the GPU driver does not automatically delete them to avoid possible performance degradation [20]. We perform a similar attack on the local and private memories, and detect that GPUs also do not automatically clear them. Accordingly, we conclude that current GPUs ignore the uninitialized memory problems.

B. Overview of Advanced Attacks

We introduce attacks to acquire data stored in the global, local, and private memories of GPUs. We consider two attack points: the end of the GPU context and the end of the GPU kernel. Figure 3 provides an overview of the proposed attacks. Normal GPU execution flow creates a GPU context (CC), allocates GPU memory (AM), copies data from CPU to GPU (HD), executes a kernel (EK), copies back data from GPU to CPU (DH), frees GPU memory (FM), and destroys context (DC) (Figure 3a). In the case of graphics applications, copying results to video frame buffer replaces DH. When the victim follows the same execution flow, the End-of-Context (EoC) attack and the End-of-Kernel (EoK) attack steal data at different moments. The EoC attack dumps all GPU memory after the victim frees its memory (Figure 3b). The EoK attack steals the local and private memories of the victim's kernel right after the victim's kernel terminates (Figure 3c). We mainly explain the proposed attacks using NVIDIA GPUs, and discuss differences between NVIDIA and AMD GPUs in terms of performing the proposed attacks.

C. End-of-Context (EoC) Attack

1) Attack Description: The EoC attack aims to obtain data released after the destruction of a victim program's GPU context. A GPU program can either explicitly destroy its GPU context by calling API functions (e.g., `clReleaseContext()` and `cudaDeviceReset()`) or implicitly destroy its GPU context by its termination. The

Algorithm 1 End-of-Context Attack

```
Input: $own \leftarrow$ the size of memory occupied by attacker
 1: $context \leftarrow createGPUContext()$
 2: $total \leftarrow getTotalMemoryInfo()$
 3: $available \leftarrow getAvailableMemoryInfo()$
 4: while $available + own = total$ do    // no victim exists
 5:     $sleep()$
 6:     $available \leftarrow getAvailableMemoryInfo()$
 7: end while
 8: $available \leftarrow getAvailableMemoryInfo()$
 9: while $available + own \neq total$ do    // victim works
10:     $sleep()$
11:     $available \leftarrow getAvailableMemoryInfo()$
12: end while
13: $alloc \leftarrow allocateMemory(total - own)$    // victim exits
14: $memoryCopyDeviceToHost(alloc)$
```

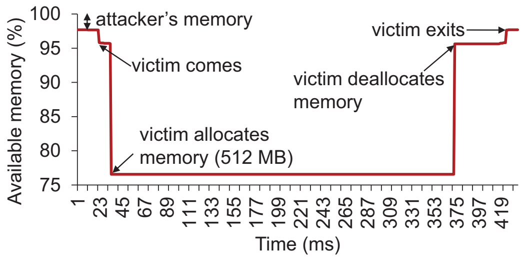


Figure 4. Changes of the available global memory size according to a victim program's activities. The total size of global memory is 2687 MB (NVIDIA Tesla C2050 GPU).

main target of this attack is the results of kernel computations, such as decrypted plaintext and rendered images. If a victim program does not clear its global memory before releasing its GPU context, an attacker can easily obtain the computation results using this attack. Furthermore, we observe that the GPU context destruction also releases other important data, such as *kernel code*, *constant data*, and *call-by-value arguments of kernels*. Current GPUs provide no methods to delete such data.

2) Attack Procedure: Algorithm 1 shows the EoC attack. NVIDIA GPUs provide functions to examine the available and total GPU memory size of the GPUs, such as `cudaMemGetInfo()`. By continuously examining the changes in the available memory size using such functions, an attacker can discern whether a victim program destroyed its GPU context. Figure 4 shows the changes in available memory size according to the victim's activities. The attacker can learn when the victim starts to use GPUs, allocates and deallocates global memory, and exits from the GPUs by leveraging the memory usage history. If the victim no longer uses GPUs, the attacker tries to dump the entire global memory of the GPUs.

3) Reducing Analysis Space: We suggest a technique to reduce the analysis space in memory dumps because (1) usual victim programs only use a small portion of the global memory and (2) investigating several gigabytes of the global memory requires unnecessary efforts. Our technique is to fill the global memory with sufficiently long binary sequences before a victim comes and ignore the sequences when analyzing dumps of the global memory. We modify the attack program to fill the global memory of a GPU with a predefined 1024-bit integer before a victim program arrives. When analyzing memory dump files, we ignore the 1024-bit integer in the files. The probability that a victim program has the predefined 1024-bit integer in its memory is negligible. We test this technique with simple victim programs allocating 64, 128, 256, 512, and 1024 MB of global memory, respectively. On average, the size of analysis space is only 1.3 MB larger than the allocated memory for storing kernel codes and constant data.

Algorithm 2 End-of-Context Attack on Multiple Victims

```
Input: $own \leftarrow$ the size of memory occupied by attacker
 1: $context \leftarrow createGPUContext()$
 2: $total \leftarrow getTotalMemoryInfo()$
 3: $available \leftarrow getAvailableMemoryInfo()$
 4: while $available + own = total$ do    // no victim exists
 5:     $sleep()$
 6:     $available \leftarrow getAvailableMemoryInfo()$
 7: end while
 8: $available \leftarrow getAvailableMemoryInfo()$
 9: while $available + own \neq total$ do    // victims work
10:     $sleep()$
11:     $avail\_new \leftarrow getAvailableMemoryInfo()$
12:     if $avail\_new > available$ then    // victims deallocate memory
13:         $alloc \leftarrow allocateMemory(avail\_new)$
14:         if $kernelDetectInstruction(alloc) = true$ then
15:             $memoryCopyDeviceToHost(alloc)$    // only copy memory with code
16:         end if
17:         $fillMemory(alloc)$
18:         $deallocateMemory(alloc)$
19:     end if
20:     $available \leftarrow getAvailableMemoryInfo()$
21: end while
22: $alloc \leftarrow allocateMemory(total - own)$    // victims exit
23: $memoryCopyDeviceToHost(alloc)$
```

4) Multiple Victims: The EoC attack in Algorithm 1 does not work when multiple victims are consecutively using the GPUs. To solve the problem, we modify the attack to deal with multiple victims (Algorithm 2). Whenever the size of available GPU memory increases, this algorithm attempts to allocate all the available memory to obtain the recently deallocated memory. However, dumping all the available memory whenever deallocations occur requires much storage and transmission overhead. The algorithm avoids the overhead by launching a kernel to verify whether the recent memory deallocation is due to the destruction of a GPU context. It is possible by checking whether the deallocated memory includes instructions (e.g., `0x85800000001c3c02` which is the `NOP` instruction of NVIDIA Kepler GPUs). Lastly, the algorithm copies memory containing instructions to the host

Algorithm 3 End-of-Kernel Attack

```
Input: $own \leftarrow$ the size of memory occupied by attacker
 1: $context \leftarrow createGPUContext()$
 2: $total \leftarrow getTotalMemoryInfo()$
 3: $available \leftarrow getAvailableMemoryInfo()$
 4: while $available + own = total$ do    // no victim exists
 5:     $sleep()$
 6:     $available \leftarrow getAvailableMemoryInfo()$
 7: end while
 8: $available \leftarrow getAvailableMemoryInfo()$
 9: while $available + own \neq total$ do    // victim works
10:     $local\_priv \leftarrow kernelReadLocalPrivMem()$
11:     $memoryCopyDeviceToHost(local\_priv)$
12:     $avail\_new \leftarrow getAvailableMemoryInfo()$
13:     if $avail\_new > available$ and $avail\_new + own \neq total$
    then    // victim releases some global memory
14:         $alloc \leftarrow allocateMemory(avail\_new)$
15:         $memoryCopyDeviceToHost(alloc)$
16:         $deallocateMemory(alloc)$
17:     end if
18:     $available \leftarrow avail\_new$
19: end while
```

and fills the memory with a dummy value before releasing it, to avoid redundant detections of the same instructions.

D. End-of-Kernel (EoK) Attack

1) Attack Description: The EoK attack aims to obtain intermediate data generated when GPU kernels of a victim program are executing. GPU computing models discourage long-running GPU kernels because current GPUs do not support preemptive scheduling. Long-running GPU programs thereby use either several kernels or the same kernel repeatedly and process the intermediate results. The main targets of this attack are frequently accessed data stored in the per-CU local memory and the per-PE private memory. For example, `libgpucrypto`, a cryptography library of SSLShader [12], loads secret keys, AES S-box, and the p and q values of RSA into the local and private memories in order to increase performance. If a victim program does not clear the local and private memories at the end of each kernel execution, an attacker can easily read the data.

2) Attack Procedure: Algorithm 3 shows the EoK attack. In this algorithm, we execute kernels that attempt to read the local and private memories of a GPU, and copy the results to CPU memory. We also check the differences in available memory size to determine whether a victim program dynamically releases some of its global memory. When we detect such memory release, we also attempt to dump it. The loop for reading the local and private memories terminates when the victim program exits from the GPU, and we lastly perform the EoC attack.

3) L1 Data Cache of NVIDIA GPUs: The EoK attack can also acquire sensitive data kept in the L1 data cache of NVIDIA GPUs. NVIDIA GPUs utilize a portion of their per-CU local memory as an L1 data cache of the global memory. Programmers can flexibly configure the size of the local memory and the L1 data cache: 16 KB for the local memory and 48 KB for the L1 data cache or vice versa. However, this configuration allows attackers to read the 32 KB overlapped region used by the L1 data cache of a victim program.

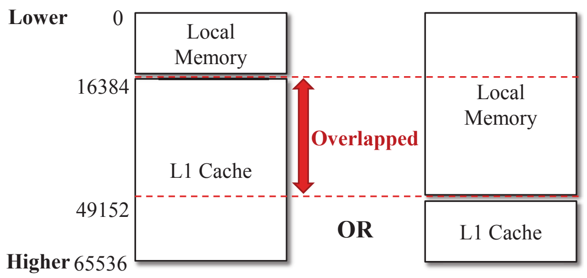


Figure 5. Memory layouts of configurable local memory and L1 data cache (NVIDIA GPUs).

```
// prepare global memory filled with zeros
int *zero_mem;
cudaMalloc((void**)&zero_mem, 49152);
cudaMemset(zero_mem, 0, 49152);
…

__device__ void flushL1(int *zero_mem) {
  for (int i=0; i < 49152/sizeof(int); ++i) {
    zero_mem[i] = zero_mem[49152/sizeof(int)-(i+1)];
  }
}
```

Figure 6. A sample kernel to flush L1 data cache.

Figure 5 shows the layouts of the local memory and the L1 data cache, verified by conducting the following experiment. We first execute a victim program that writes zeros into its 16 KB of local memory and reads 512 MB of global memory filled with `0x11111111`. We then execute an attack program that reads and dumps its 48 KB of local memory. We detect that the lower 16 KB of the local memory is filled with zeros and the upper 32 KB of the local memory is filled with `0x11111111`. Therefore, attackers can obtain the lower 2/3 of the L1 data cache from a victim GPU program if the victim uses a 48 KB L1 data cache.

A GPU program can clear cached data by reading the contiguous 48 KB global memory block filled with a dummy value because the L1 data cache of NVIDIA GPUs is a set-associative, write-evict cache [2], [30]. Figure 6 shows an example of a CUDA code that flushes the L1 data cache. It prepares a 48 KB array filled with zeros in the global memory in advance and reads zeros from the array to clear the L1 data cache.

4) Multiple Victims: The limitation of the EoK attack is that it can only read the local and private memories of a victim kernel which uses the GPU just before an attack kernel. This implies that when multiple victim programs compete to use a GPU, the attack kernel can only see one

Table I
PLATFORMS WE TEST THE PROPOSED ATTACKS.

GPU (Generation)	GPU Mem.	Driver	OS	Kernel	CPU	CPU Mem.
NVIDIA						
GeForce 210 (GT200)	0.5 GB	319.37	Ubuntu 12.04	3.5.0	Intel Pentium Dual-Core E6300	4 GB
Tesla C2050 (Fermi)	2.6 GB	304.108	CentOS 6.3	2.6.32	Intel Xeon X5650*2	24 GB
GeForce GTX 780 (Kepler)	3.0 GB	325.15	Ubuntu 12.04	3.5.0	Intel Core i7-2600	8 GB
AMD						
Radeon HD 7850 (Pitcairn)	1.8 GB	13.1	CentOS 6.4	2.6.32	Intel Xeon E5430*2	8 GB
FirePro W9000 (Tahiti)	6.0 GB	12.104.2	CentOS 6.4	2.6.32	Intel Xeon E5430*2	8 GB

of their data kept in local and private memories.

E. Attacks on AMD GPU

1) Differences and Increased Vulnerability: The attacks on AMD GPUs slightly differ from the attacks on NVIDIA GPUs due to dynamic memory management of AMD GPUs and OpenCL[2]. Unlike NVIDIA GPUs, AMD GPUs and OpenCL provide no APIs to check available memory size as they dynamically manage the global memory. When a new GPU program requests large memory blocks exceeding the available global memory size, while an old GPU program occupying a portion of global memory is inactive, AMD GPUs automatically move the old GPU program's data back to CPU memory to fulfill the new program's requirements. NVIDIA GPUs do not provide this functionality, although they support OpenCL.

However, we detect that *the AMD GPU driver does not nullify GPU memory for the new program*. An attacker can read the global memory of a victim program when he or she requests memory *before* the victim program deletes and deallocates its GPU memory.

Instead of checking the available GPU memory sizes, we use the changes in kernel execution timing for determining whether a victim program uses GPUs. GPU kernels of different programs share a GPU in a time-sharing fashion, so that the execution time of a GPU kernel varies according to other kernels using the GPU.

2) Attack Procedure: Algorithm 4 shows the EoK attack on AMD GPUs. We execute a dummy kernel and measure its execution time to know whether a victim program uses the GPU. If a victim program exists, the execution time of the dummy kernel certainly increases because current GPUs cannot concurrently execute different GPU programs [18]. When an attacker detects a victim, the attacker executes kernels for reading the local and private memories. Furthermore, the attacker can also acquire the entire global memory because of the dynamic memory management of AMD GPUs. Consequently, the EoC attack is unnecessary when attacking AMD GPUs.

[2]http://devgurus.amd.com/message/1296453

Algorithm 4 End-of-Kernel Attack on AMD GPUs

```
Input: $own \leftarrow$ the size of memory occupied by attacker
1: $context \leftarrow createGPUContext()$
2: $time \leftarrow kernelDummy()$
3: while $time < threshold$ do    // no victim exists
4:     $sleep()$
5:     $time \leftarrow kernelDummy()$
6: end while
7: while $time \geq threshold$ do    // victim works
8:     $local\_priv \leftarrow kernelReadLocalPrivMem()$
9:     $memoryCopyDeviceToHost(local\_priv)$
10:    $alloc \leftarrow allocateMemory(total - own)$
11:    $memoryCopyDeviceToHost(alloc)$
12:    $sleep()$
13:    $time \leftarrow kernelReadData()$
14: end while
```

F. Test Platforms

We test five different platforms to check the coverage of the proposed attacks (Table I). The test platforms include NVIDIA GeForce 210, NVIDIA Tesla C2050, NVIDIA GeForce GTX 780, AMD Radeon HD 7850, and AMD FirePro W9000 GPUs on Linux operating systems with various driver versions. We verify that our attacks succeed in the test platforms without errors.

G. Attacks on "Real" Programs

So far, we explain our attacks to disclose GPU memory used by an experimental GPU program we made that only handles meaningless data. To show that our attacks are not only highly applicable, but also crucial threats to both GPUs and their users, we have to attack a *real program* that (1) uses GPU APIs, (2) deals with sensitive data, and (3) is popular. Recent web browsers, such as Chromium and Firefox, fulfill all the requirements: (1) they use graphics APIs for efficient webpage rendering, (2) they handle a user's private data such as browsing history, and (3) they are extremely popular. Therefore, we choose them as our attack targets and discuss the results in the next section.

V. INFERRING WEB BROWSING HISTORY FROM GPUS

In this section, we explain our attacks on web browsers to infer web browsing history of a victim user using data extracted from GPUs by leveraging the attacks explained in Section IV. Recent web browsers, such as Chromium and

Firefox, support GPU-accelerated webpage rendering so we expect that *rendered webpage textures may remain in GPU memory*. Our inference attacks match the GPU memory dump with either known webpage dumps or known webpage snapshots to infer which webpages a victim user has visited.

We identify that webpage textures remain not in the local and private memories, but in the global memory. The content of the local and private memories does not change according to which webpages a victim has visited, but changes according to which web browsers a victim uses. Hence, we focus on attacking the global memory to infer web browsing history.

A. Web Browsers and Configurations

We use the Chromium web browser version 30 and Firefox web browser version 25 in this case study. For the Chromium web browser, we enable the "GPU compositing of all page" option to use GPU-accelerated webpage composition. For the Firefox web browser, we enable the layers.offmainthreadcomposition.enabled and layers.acceleration.force-enabled options, and disable the layers.use-deprecated-textures option. We execute the web browsers on three Linux systems with NVIDIA GeForce 210, NVIDA GeForce GTX 780, and AMD FirePro W9000 GPUs (Table I). The Linux systems with the NVIDIA GPUs use Xfce 4.8, and another Linux system with the AMD GPU uses Gnome 2.28.2.

B. GPU Memory Dump and Texture Rearrangement

When we examine a GPU memory dump of `google.com` extracted from the Chromium web browser with an NVIDIA GeForce GTX 780 GPU obtained by performing the EoC attack, we find a number of 32-bit values that seem to represent colors, such as `0x00ffffff`, `0x00404040`, `0x00e85947`, and `0x00da3d29`. Starting from the most significant bit, we treat each two bytes as blank, red, green, and blue color values, respectively. We construct Figure 1b by judging each value with the rule while ignoring black (zeros).

As Figure 1 shows, GPUs store textures on GPU memory in a rearranged form so that we need solutions to recognize them. However, recovering the original textures from the rearranged textures is difficult because (1) GPU vendors document nothing about the hardware-level texture management, (2) GPUs have virtualized and paged memory, and (3) GPU memory dumps also contain other non-color data. Therefore, instead of trying to recover the original textures, we strive to design methods for inferring the visited webpages from the rearranged textures.

C. Overview of Attack Scenarios

We consider three attack scenarios to know how attackers can infer browsing history of victims by leveraging GPUs in various situations. First, we assume an attacker who prepares *GPU memory dumps of known webpages* extracted from the same GPU a victim uses and tries to compare them with a GPU memory dump of an unknown webpage. We confirm that this attack can correctly infer up to 95.4% of randomly selected 100 front pages of Alexa Top 1000 websites.

Second, we assume an attacker who prepares *image snapshots of known webpages* and tries to compare them with a GPU memory dump of an unknown webpage. Unlike the first attack, this attack does not require that the attacker and victims use the same GPU. This attack correctly infers ∼50% of the randomly selected 100 front pages.

Third, we assume an attacker trying to attack a victim who simultaneously opens multiple webpages using either *multiple tabs or windows*. We observe that the attacker can accurately infer the webpages of the front tab or the lastly rendered window.

D. Attack using Known GPU Memory Dump

In this attack, an attacker prepares GPU memory dumps of famous webpages to compare them with a new GPU memory dump of a victim web browser. We choose front pages of Alexa Top 1000 websites as our dataset. We visit each of them using the Chromium and Firefox web browsers with NVIDIA and AMD GPUs, respectively, and close the browsers 60 second later while recording GPU memory dumps using the EoC attack. We repeat these procedures 10 times to average out the results.

We use three matching methods for comparing GPU memory dumps: pixel sequence matching, RGB histogram matching, and a combination of them.

1) Pixel Sequence Matching: The pixel sequence matching compares non-black and non-white contiguous pixel sequence sets extracted from two GPU memory dumps using *Jaccard Index (JI)*. We ignore (1) black pixels because we cannot distinguish them with zero in memory and (2) white pixels because most webpages have a large number of white pixels.

For example, if a GPU memory dump contains the following pixels

$$c_1, c_2, c_b, c_3, c_b, c_4, c_w, c_w, c_5, c_6, c_b,$$

where each c_i is a tuple of red, blue, and green color values (0–255), c_b is black (0,0,0), and c_w is white (255,255,255), the pixel sequence set is

$$P = \{(c_1, c_2), (c_3), (c_4), (c_5, c_6)\}.$$

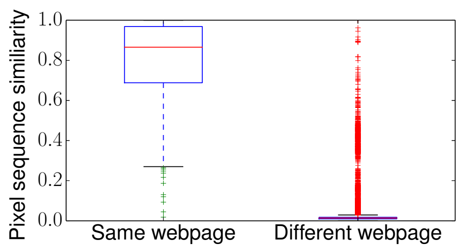
Moreover, the JI of the following two pixel sequence sets

$$P_1 = \{(c_1, c_2), (c_3), (c_4), (c_5, c_6)\},$$

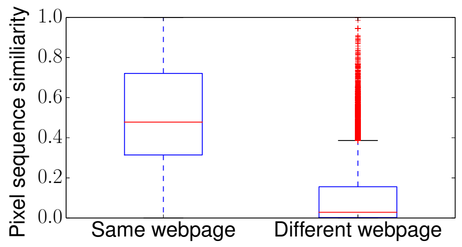
$$P_2 = \{(c_1, c_2, c_3), (c_4), (c_5, c_6)\},$$

are

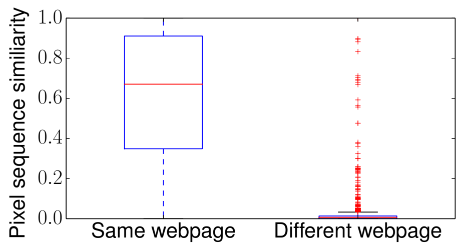
$$\frac{|P_1 \cap P_2|}{|P_1 \cup P_2|} = \frac{|(c_4), (c_5, c_6)|}{|(c_1, c_2), (c_3), (c_4), (c_5, c_6), (c_1, c_2, c_3)|} = \frac{2}{5}.$$



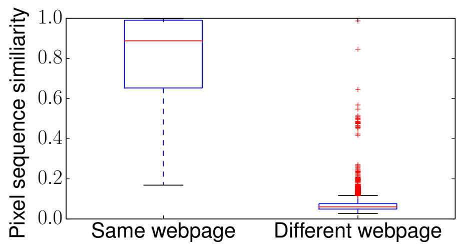
(a) Chromium with GTX 780.



(b) Firefox with GTX 780.



(c) Chromium with W9000.



(d) Firefox with W9000.

Figure 7. Pixel sequence similarity between webpage dumps extracted from NVIDIA and AMD GPUs.

Figure 7 shows boxplots of pixel sequence similarity of the same and different webpages with NVIDIA and AMD GPUs. We observe that the pixel sequence similarity between the same webpage is fairly higher than that of different webpages in all cases. The median similarity between the same webpage is 0.865 whereas that of the different webpage is 0.014 in the Chromium browser with the NVIDIA GPU, those of the Firefox browser with the NVIDIA GPU are 0.478 and 0.029, those of the Chromium browser with the AMD GPU are 0.671 and 0.007, and those of the Firefox browser with the AMD GPU are 0.888 and 0.060. When computing the cross similarity between different webpages, we use the *centroid* pixel sequence set of each webpage, whose average similarity between pixel sequence sets of the same webpage is the largest.

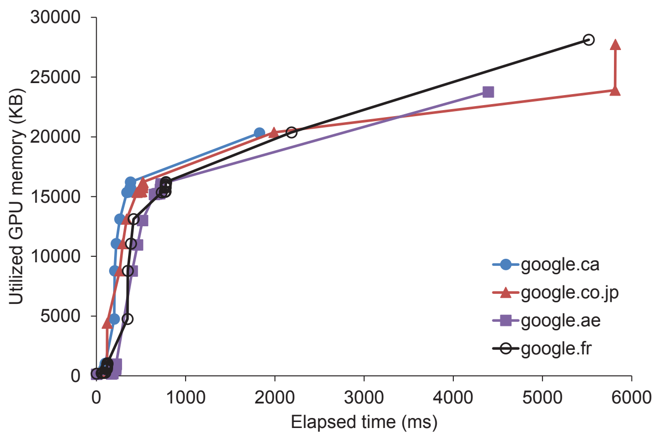


Figure 8. The amount of GPU memory the Chromium web browser utilizes for rendering four different Google webpages with an NVIDIA GTX 780 GPU.

We further inspect (1) the same webpages having low pixel sequence similarity and (2) different webpages having high pixel sequence similarity. First, most of the same webpages with the low similarity are dynamic webpages showing different images at each visit, such as `apple.com` and `tumblr.com`. If attackers prepare a number of GPU memory dumps corresponding to the dynamic contents, they may overcome this limitation.

Second, most of the different webpages with high similarity are either similar or the same webpages having different domain names, such as (`google.com`, `google.co.uk`) and (`facebook.com`, `fbcdn.net`). We are certain that distinguishing them is less meaningful because attackers can infer a victim's preferences using one of the similar webpages. Furthermore, we can distinguish them if we monitor changes in GPU memory utilization by the browsers as Memento [28] does. Figure 8 shows that the Chromium web browser has different GPU memory usage patterns when rendering four different Google webpages. Such a monitoring, however, only works with NVIDIA GPUs because AMD GPUs provide no APIs to check the available global memory size.

The limitation of the pixel sequence matching is that pixel sequences heavily depend on which GPU and web browser a victim uses (Figure 9). For this reason, attackers should prepare the web browser and the GPU that are equivalent to those of a victim to perform this attack.

2) RGB Histogram Matching: The RGB histogram matching compares non-black and non-white RGB histograms derived from two GPU memory dumps using *Euclidean distance*. An RGB histogram is a tuple of 256 values

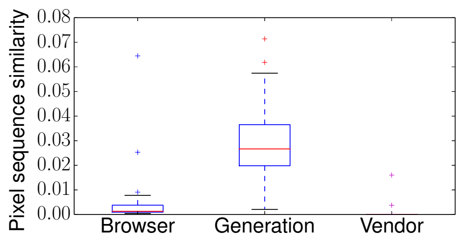


Figure 9. Pixel sequence similarity between the same webpage dumps extracted from different browsers (on an NVIDIA GeForce GTX 780), different generation of GPUs (NVIDIA GeForce 210 vs. GTX 780 with Chromium), and different vendors (NVIDIA GTX 780 vs. AMD W9000) with Chromium.

for red, blue, and green channels as follow:

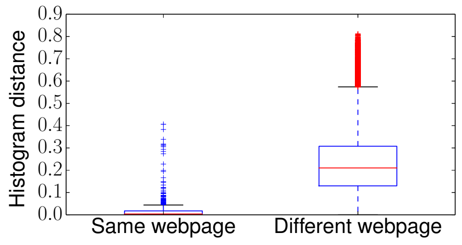
$$H = (r_0, r_1, \ldots, r_{255}, g_0, g_1, \ldots, g_{255}, b_0, b_1, \ldots, b_{255}).$$

For example, a pixel (128,64,32) contributes one to the 128-th, the 320-th, and the 544-th values of a tuple, respectively. We check dissimilarity of RGB histogram tuples by dividing them with the sum of all 768 values (*normalization*) and computing Euclidean distance, while using a random projection method for dimensionality reduction [31].

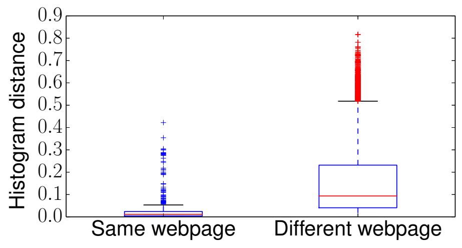
Figure 10 shows boxplots of RGB histogram distance of the same and different webpages with NVIDIA and AMD GPUs. We identify that histogram distance between the same webpage is shorter than that of the different webpages in all cases. The median distance between the same webpage is 0.004 whereas that of the different webpage is 0.210 in the Chromium browser with the NVIDIA GPU, those of the Firefox browser with the NVIDIA GPU are 0.011 and 0.093, those of the Chromium browser with the AMD GPU are 0.003 and 0.196, and those of the Firefox browser with the AMD GPU are 0.004 and 0.150. Again, dynamic webpages and similar or the same webpages with different domain names dominate errors, like the pixel sequence matching.

Since the RGB histogram has weaker correlation with GPUs than the pixel sequences has, preparing the same GPU that a victim uses is optional for performing this attack. Figure 11 shows that the RGB histogram distance between the same webpage dumps extracted from different GPUs is shorter than those of the different webpages when we use the same web browser. In contrast, dumps came from the different browsers quite differ, so that attackers should prepare different dump sets for different browsers.

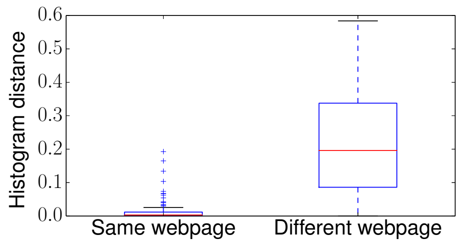
3) Inference Accuracy and Combined Matching: We evaluate the inference accuracy of the proposed matching methods. We randomly choose 100 front pages from the Alexa Top 1000 websites and visit them while recording GPU memory dumps. We then compare each of the new dumps with the known dumps to infer the corresponding webpages. When we detect a known dump having either the largest similarity or shortest distance with a new dump (a *nearest neighbor*), we treat both correspond to the same webpage. In addition, if the proposed matching methods decide the same or similar webpages having different domain names are the same, we treat the methods are correct because visiting whether `google.com` or `google.fr` incurs negligible difference in inferring a victim's preferences. We perform this procedure 10 times and finally compute their average.



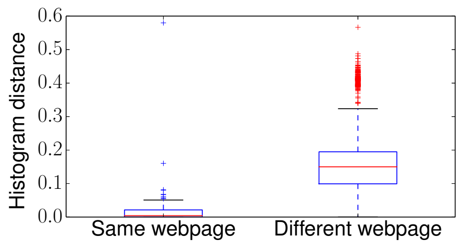
(a) Chromium with GTX 780.



(b) Firefox with GTX 780.



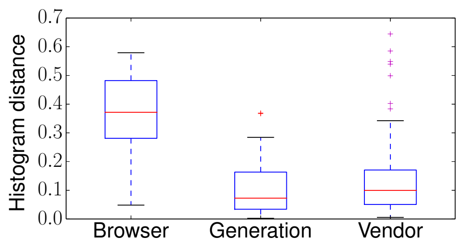
(c) Chromium with W9000.



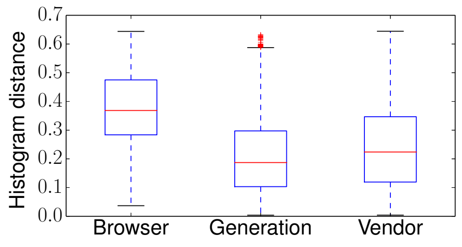
(d) Firefox with W9000.

Figure 10. RGB histogram distance between webpage dumps extracted from NVIDIA and AMD GPUs.

Figure 12 shows the evaluated inference accuracy of the proposed methods. On average, the pixel sequence matching can infer 69.4% of the randomly selected 100 webpages and the RGB histogram matching can infer 60.9% of them. We identify that the Chromium web browser with the NVIDIA



(a) Same webpage.



(b) Different webpage.

Figure 11. RGB histogram distance between dumps extracted from different browsers (on an NVIDIA GeForce GTX 780), different generation GPUs (NVIDIA GeForce 210 vs. GTX 780 with Chromium), and different vendors (NVIDIA GTX 780 vs. AMD W9000) with Chromium.

GPU and the Firefox web browser with the AMD GPU are more vulnerable.

Lastly, we *simultaneously apply the pixel sequence matching and the RGB histogram matching* when comparing dumps to further increase the inference accuracy. The procedure is as follows. First, we use the pixel sequence matching for detecting the top-k known dumps similar to a new dump (k-*nearest neighbors*). Next, we use the RGB histogram matching for selecting one of the top-k known dumps having the smallest distance with a new dump. We treat the selected known dump as the corresponding webpage of the new dump. As shown in Figure 12, the combined matching achieves the highest accuracy: it correctly infers 84.6% of the randomly selected 100 webpages on average. Particularly, victims who use the Chromium web browser with the NVIDIA GPU are in danger because the combined matching attack can infer the webpages they have visited within 95.4% accuracy.

4) Efficiency: We lastly check the efficiency of our matching methods. For the measurements, we use the test platform that has Intel Core i7-2600 CPU and 8 GB of main memory. On average, it takes ∼0.451 s and ∼0.002 s to perform the pixel sequence matching and the RGB histogram matching between two GPU memory dumps, respectively. The average execution time of the combined matching is the sum of the two average values. Since we can conduct each comparison between two GPU memory dumps in parallel, we can further accelerate the performance of the matching methods using data parallel systems (e.g., multi-core processors or GPUs' computing APIs). We omit details

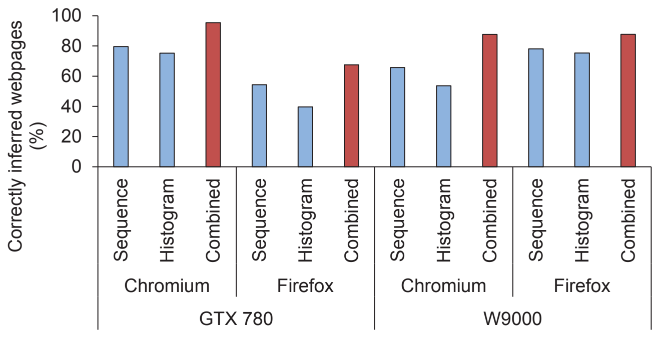


Figure 12. Portion of correctly inferred webpages according to GPUs, browsers, and matching methods.

of such accelerations in this work, because reducing attack costs is not our primary goal.

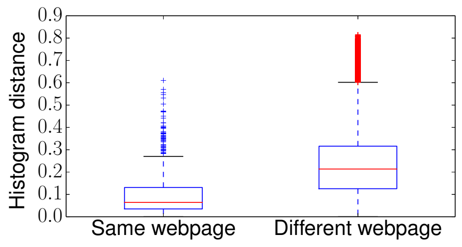
E. Attack using Webpage Snapshot

The attack using known GPU memory dumps demand attackers to prepare a number of GPU memory dumps to perform the attack. Although the RGB history matching does not need the same GPU a victim uses, still, preparing dumps is significant overhead.

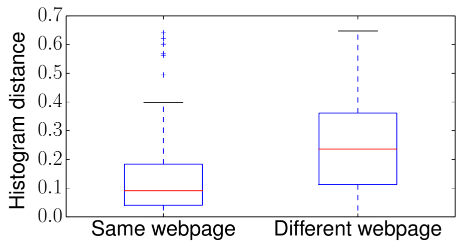
We introduce another method for further reducing attack overhead: using known webpage snapshots instead of known GPU memory dumps. First, we use PhantomJS [32] for loading the front pages of Alexa Top 1000 websites while taking their image snapshots. Next, we compare the snapshots with GPU memory dumps using the RGB histogram matching. Note that we cannot perform the pixel sequence matching using snapshots because of texture rearrangement.

When taking webpage image snapshots, we set `page.clipRect` of PhantomJS to fit to our screen size to create screen-size snapshots. This is because GPU memory dumps only contain a portion of webpages displayed on a screen, whereas webpage screenshots of PhantomJS by default also contain a portion of the webpages not displayed on the screen (full webpage screenshots).

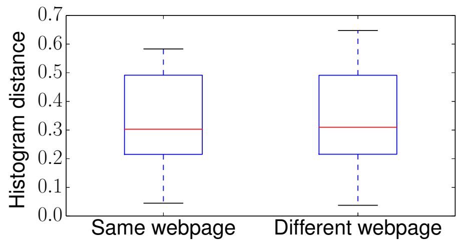
Figure 13 shows boxplots of RGB histogram distance between webpage image snapshots and dumps. We identify that the RGB histogram matching between the webpage snapshots and the dumps extracted from the Chromium browsers with both GPUs work well: the median distance between the same webpage is 0.064 whereas that of the different webpage is 0.214 in the NVIDIA GPU, and the median distance between the same webpage is 0.009 whereas that of the different webpage is 0.225 in the AMD GPU. In contrast, the RGB histogram matching does not work well when we test the Firefox browsers with GPUs: their distance is long even when we compare the same webpages. This is because the dumps extracted from the Firefox web browsers also contain non-texture data, resulting in low inference accuracy.



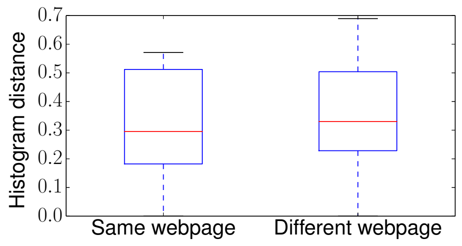
(a) Chromium with GTX 780.



(b) Chromium with W9000.



(c) Firefox with GTX 780.



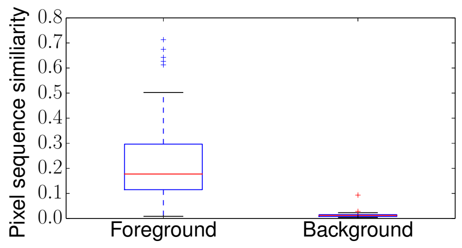
(d) Firefox with W9000.

Figure 13. RGB histogram distance between webpage image snapshots and dumps.

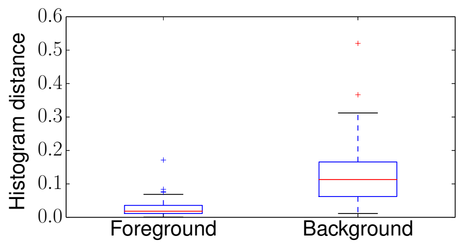
We further check inference accuracy as explained in Section V-D. We correctly infer ∼50% and ∼22% of the randomly selected 100 front pages using the Chromium web browser with NVIDIA and AMD GPUs, respectively, which is worse than that of the attack using known GPU memory dumps, but the results of NVIDIA GPUs are still meaningful.

F. Attack on Victims Browsing Multiple Webpages

Lastly, we identify whether we can attack a victim who simultaneously visits multiple webpages using tabs or separate windows. Using multiple tabs or windows of web browsers is common in a desktop environment so that considering such scenarios is meaningful.



(a) Pixel sequence.



(b) RGB histogram.

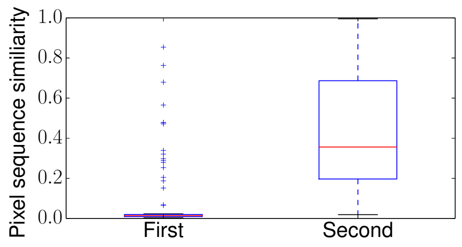
Figure 14. Comparisons between combined dumps and dumps of foreground and background webpages in two tabs (Chromium and GTX 780).

First, we visit each front page of Alexa Top 100 websites along with a randomly selected front page among them using *two tabs* while recording GPU memory dumps (10 times). Next, we check the pixel sequence similarity and the RGB histogram distance between the recorded dumps and known dumps of the foreground and background webpages, respectively. For simplicity, we only use the Chromium web browser and an NVIDIA GTX 780 GPU when performing this attack.

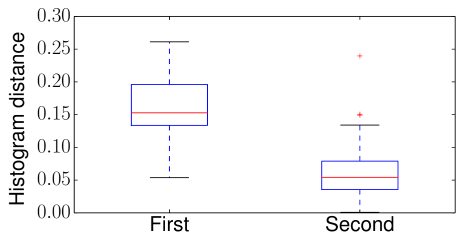
We observe that the recorded memory dumps with two tabs mostly contain the pixel sequences and the RGB histograms of the foreground webpages (Figure 14). We conclude that the Chromium web browser does not use GPUs for rendering background webpages as foreground webpages cloak them.

Second, we visit each front page of Alexa Top 100 websites *right after* visiting a randomly selected front page among them using *two separate windows* while recording GPU memory dumps (10 times). We adjust the size and position of two windows to avoid overlap between them. We then check the pixel sequence similarity and the RGB histogram distance between the recorded dumps and known dumps of the firstly and secondly rendered webpages, respectively.

We detect that the recorded memory dumps mostly contain the pixel sequences and RGB histograms of the secondly rendered webpages (Figure 15). We presume that GPUs write the textures of the secondly rendered webpages to the same buffer of the firstly rendered webpages so that the earlier textures are overwritten.



(a) Pixel sequence.



(b) RGB histogram.

Figure 15. Comparisons between combined dumps and dumps of firstly and secondly rendered webpages in two windows (Chromium and GTX 780).

We lastly check inference accuracy: the combined matching correctly infers 98.6% of webpages loaded in the foreground tabs and 90.8% of webpages loaded in the secondly rendered windows. We presume that the relatively low accuracy of the inference on the secondly rendered windows is due to the unoverwritten textures of the firstly rendered windows. Consequently, when a victim visits a number of webpages using a GPU-accelerated web browser, attackers can accurately infer some of the webpages either loaded in the foreground tab or rendered in the lastly opened window.

VI. Discussion

A simple solution for preventing the proposed attack is clearing newly allocated global memory pages as WebGL [20] does. Moreover, GPUs need to delete the per-CU local memory and per-PE private memory at GPU context switches.

Unfortunately, we expect that GPU vendors are unwilling to embrace such methods because they bring performance degradation. For example, NVIDIA targets to reduce the cost of the GPU context switch below 25 μs [33]. However, it takes 79 μs to delete the entire local and private memories when we execute our optimized GPU memory deletion program in an NVIDIA GeForce GTX 780 GPU, which is approximately three times longer than the context switching time. The GPU vendors may not accept such huge overhead because their main concerns are performance and power efficiency. Consequently, we demand new hardware- and software-level solutions for efficiently clearing GPU memory, which cannot be accomplished without GPU vendors' efforts.

While neither GPU vendors nor researchers provide such solutions, GPU programs dealing with sensitive data should delete global memory pages before deallocating them, and clear the local and private memories before context switches. The graphics APIs, however, provide no functions to clear memory contents. Therefore, graphics programs have to use the computing APIs for manually clearing allocated GPU memory, though it results in performance degradation and high programming complexity.

VII. Related Work

In this section, we introduce some related studies of this work.

A. *Remote Pixel Stealing in HTML5*

Numerous researchers [34]–[36] consider security attacks exploiting HTML5 CSS filters that allow web developers to apply various graphics effects on webpages using host GPUs. By applying CSS filters to a target webpage loaded in an iframe while measuring the completion time, the CSS filter-based attacks can recognize a user's login status and steal pixels of the target webpage. However, the attacks have restricted coverage because (1) they should deceive victims to visit their malicious webpage and (2) many webpages disallow web browsers to load them in an iframe to avoid security attacks.

B. *Security Attacks using GPUs*

GPU-based cracking against passwords or hash values are well-known security attacks [37]. Some academic studies also utilize GPUs for conducting general security attacks. Vasiliadis *et al.* [38] shows the possibility of malware obfuscation using GPUs. First, they load an encrypted malware on a host's main memory and map its memory address to GPU memory to enable direct access on the memory from GPUs, also known as zero-copy memory [2]. Next, their code decrypts the malware, and the host finally executes it. Ladakis *et al.* [39]'s GPU-based keylogger also relies on memory-mapped IOs. The keylogger first uses a rootkit for mapping the keybuffer in kernel memory to GPU memory. The GPU part of the keylogger then records keystrokes through the mapped keybuffer, and lastly returns the recorded keystrokes to the host.

C. *General Applications of GPUs*

Several researchers conduct various studies to utilize GPUs for solving general and computation-intensive problems. For example, researchers try to use GPUs for increasing the performance of AES [13]–[15] and RSA [16], [17] algorithms. Some researchers also implement high-speed intrusion detection systems (IDSs) [10], [11] and an SSL accelerator [12]. Other researchers also introduce GPU-accelerated routers for IP network [9] and database accelerators [7], [8].

The preceding applications, however, may suffer from serious threats because of the security problems discussed in this work. For example, attackers can extract secret keys and plaintext from GPU-based cryptographic engines without a root privilege. Furthermore, they can capture packets from GPU-based IDSs, SSL accelerators, and routers. Consequently, both GPU programmers and researchers need to be aware of the security problems of GPUs that this work considers.

VIII. Conclusion

GPUs become more powerful and general, and many applications increase their performance using them. However, no in-depth study has considered their security problems. In this paper, we investigated the security vulnerabilities of GPUs, and described attacks that reveal a victim's sensitive data kept in GPUs. We further applied the proposed attacks on popular programs using GPUs: the Chromium and Firefox web browsers utilizing GPUs for faster webpage rendering. We were able to successfully obtain rendered webpage textures remaining in GPU memory and accurately infer their original webpages. Therefore, both GPU vendors and programmers need to know that GPU programs can be in danger, and prepare countermeasures to cope with such threats.

Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful comments and suggestions. This research was supported by the MSIP, Korea, under the ITRC support program supervised by the NIPA through CMEST (NIPA-2013-H0301-13-3002) and NRF Grant funded by the Korean Government (NRF-2012-Global Ph.D. Fellowship Program).

References

[1] NVIDIA, "NVIDIA's next generation CUDA compute architecture: Kepler GK110," 2012.

[2] ——. CUDA C programming guide version 5.5. http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[3] Microsoft, "DirectX graphics and gaming," http://msdn.microsoft.com/en-us/library/windows/desktop/ee663274(v=vs.85).aspx.

[4] D. Shreiner, G. Sellers, J. M. Kessenich, and B. M. Licea-Kane, *OpenGL Programming Guide: The Official Guide to Leaning OpenGL.* Addison Wesley, 2013.

[5] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL.* Morgan Kaufmann, 2012.

[6] NVIDIA, "Tesla supercomputing HPC industrical case studies," http://www.nvidia.com/object/tesla-case-studies.html.

[7] P. Bakkum and K. Skadron, "Accelerating SQL database operations on a GPU with CUDA," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*, 2010.

[8] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili, "Kernel weaver: Automatically fusing database primitives for efficient GPU computation," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

[9] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: a GPU-accelerated software router," in *Proceedings of the ACM SIGCOMM 2010 Conference*, 2010.

[10] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "MIDeA: A multi-parallel intrusion detection architecture," in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, 2011.

[11] M. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park, "Kargus: a highly-scalable software-based intrusion detection system," in *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, 2012.

[12] K. Jang, S. Han, S. Han, S. Moon, and K. Park, "SSLShader: Cheap SSL acceleration with commodity processors," in *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[13] D. L. Cook, J. Ioannidis, A. D. Keromytis, and J. Luck, "CryptoGraphics: Secret key cryptography using graphics cards," in *Proceedings of The Cryptographer's Track at RSA Conference 2005 (CT-RSA)*, 2005.

[14] J. Yang and J. Goodman, "Symmetric key cryptography on modern graphics hardware," in *Proceedings of the 13th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2007.

[15] O. Harrison and J. Waldron, "Practical symmetric key cryptography on modern graphics hardware," in *Proceedings of the 17th USENIX Security Symposium*, 2008.

[16] ——, "Efficient acceleration of asymmetric cryptography on graphics hardware," in *Proceedings of the 2nd International Conference on Cryptology in Africa (AFRICACRYPT)*, 2009.

[17] R. Szerwinski and T. Güneysu, "Exploiting the power of GPUs for asymmetric cryptography," in *Proceedings of the 10th Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2008.

[18] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, "The case for GPGPU spatial multitasking," in *Proceedings of the 18th IEEE International Conference on High Performance Computer Architecture (HPCA)*, 2012.

[19] VirtualGL. The VirtualGL project. http://www.virtualgl.org.

[20] Khronos, "WebGL security," http://www.khronos.org/webgl/security/.

[21] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum, "Shredding your garbage: Reducing data lifetime through secure deallocation," in *Proceedings of the 14th USENIX Security Symposium*, 2005.

[22] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation," in *Proceedings of the 13th USENIX Security Symposium*, 2004.

[23] T. Garfinkel, B. Pfaff, J. Chow, and M. Rosenblum, "Data lifetime is a system problem," in *Proceedings of the 11th ACM SIGOPS European Workshop*, 2004.

[24] A. M. Dunn, M. Z. Lee, S. Jana, S. Kim, M. Silberstein, Y. Xu, V. Shmatikov, and E. Witchel, "Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[25] R. Di Pietro, F. Lombardi, and A. Villani, "CUDA leaks: Information leakage in GPU architectures," *ArXiv e-prints*, 2013.

[26] C. Maurice, C. Neumann, O. Heen, and A. Francillon, "Confidentiality issues on a GPU in a virtualized environment," in *Proceedings of the 18th International Conference on Financial Cryptography and Data Security (FC)*, 2014.

[27] R. Love, *Linux Kernel Development*. Addison Wesley, 2010.

[28] S. Jana and V. Shmatikov, "Memento: Learning secrets from process footprints," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, 2012.

[29] K. Zhang and X. Wang, "Peeping tom in the neighborhood: Keystroke eavesdropping on multi-user systems," in *Proceedings of the 18th USENIX Security Symposium*, 2009.

[30] R. Meltzer, C. Zeng, and C. Cecka, "Micro-benchmarking the C2070," in *GPU Technology Conference*, 2013.

[31] E. Bingham and H. Mannila, "Random projection in dimensionality reduction: Applications to image and text data," in *Proceedings of the 17th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2001.

[32] A. Hidayat. PhantomJS: Headless WebKit with JavaScript API. http://phantomjs.org.

[33] NVIDIA, "NVIDIA's next generation CUDA compute architecture: Fermi," 2009.

[34] R. Kotcher, Y. Pei, P. Jumde, and C. Jackson, "Cross-origin pixel stealing: Timing attack using CSS filters," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, 2013.

[35] P. Stone, "Pixel perfect timing attacks with HTML5," in *Blackhat USA*, 2013.

[36] A. Barth, "Adam barth's proposal," http://www.schemehostport.com/2011/12/timing-attack-on-css-shaders.html.

[37] D. Goodin, "25-GPU cluster cracks every standard Windows password in <6 hours."

[38] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "GPU-assisted malware," in *Proceedings of the 5th International Conference on Malicious and Unwanted Software (Malware)*, 2010.

[39] E. Ladakis, L. Koromilas, G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "You can type, but you can't hide: A stealthy GPU-based keylogger," in *Proceedings of the 6th European Workshop on System Security (EuroSec)*, 2013.