University of Amsterdam

OS3/SNE – CyberCrime and Forensics

Portable and forensically sound VRAMFS forensics on NVIDIA GPUs

June 11, 2024

Students:
Vlad Arsene
vlad.arsene@os3.nl

Yassir Laaouissi
yassir.laaouissi@os3.nl

Ferran Tufan
ferran.tufan@os3.nl

Course:
CyberCrime and Forensics

Course code:
5384CYFO6Y

Abstract

Graphics Processing Units (GPUs) have their own type of Video Random Access Memory (VRAM) for low-latency memory operations in order to allow high-performance graphics rendering. The presence of this VRAM also opens up the possibility of hiding data in a peripheral device that may or may not be considered of interest to forensics researchers.

In this paper, we aim to research new and existing methods to extract VRAM in a forensically sound manner to uphold the legal validity of the evidence, shall it be taken to a court case. We will perform tests on different generations of GPUs.

Our results show that VRAM dumping software based on the 'memcpy' functions in OpenCL and CUDA work on an older-generation GPU, but the same software does not work on newer generations of GPUs. The validity of core dumping in CUDA has been evaluated, and it shows this method does not work on the newer GPUs either. Memory address maps present in the Advanced Configuration and Power Interface (ACPI) may be used for writing new dumping software executing raw input/output instructions on GPUs, but no working software has been written yet.

Contents

List of Tables

List of Figures

Acronyms

ACPI Advanced Configuration and Power Interface

FUSE File System in User Space

GPU Graphics Processing Unit

RAM Random Access Memory

VRAM Video Random Access Memory

VRAMFS Video Random Access Memory File System

1 Introduction

GPUs are either peripheral or integrated devices to accelerate the rendering of video frames for computer systems. Due to the integration of special hardware components and instruction set architectures, GPUs can be used for high-performance calculations in computing [7].

GPUs are powered by VRAM, internal memory allocated with lower access times than the standard Random Access Memory (RAM) [8]. Over time, the amount of VRAM in GPUs has increased, due to more performance demands from end-users of GPUs. Consequently, initiatives started exploring the ability to store files in said VRAM. An example of such an initiative is Video Random Access Memory File System (VRAMFS). VRAMFS is based on the library called File System in User Space (FUSE).

Digital Forensics consists of various fields, one of which is Incident Response. The transformation of VRAM to a persistent storage media using VRAMFS opens possibilities for adversaries to store malicious payloads [1] in unorthodox locations. With the logical consequence is that Incident Response analysts want to triage these capabilities as fast as possible, with minimal constraints. This study delves into the publicly available parsing of VRAMFS.

2 Research questions

2.1 Main question

How can acquisition be applied to VRAMFS on different NVIDIA GPUs, such that minimal data integrity compromises are present?

2.2 Sub questions

- How do existing VRAMFS parsers acquire/interpret the file system, and what effect do they have on the data integrity?
- What challenges are faced with parsing VRAMFS from untested NVIDIA GPUs using existing tools?

3 Related work

Previous research into VRAMFS forensics reveals insight into the structure of the file system and ways to capture the VRAMFS. The related work of this study is described below:

- **VRAMGrabber**: a tool that utilizes OpenCL 1.2 and up to fetch the full VRAM of all available GPU devices. The fetched VRAM is stored in the local home directory (cross-platform), where the size of the raw VRAM data files is identical to the total VRAM available in every individual device. This tool was the product of a study conducted by former OS3 students [2]. Note that the previously conducted study formed the basis of our research, as described in this paper. The tool made in the previous study was not verified on modern GPU devices. Hence the scope of our research, and the utilized context from the previous study.

4 Methods

In order to conduct forensics on VRAMFS, we needed to set up an environment that supports VRAMFS and thus has an integrated and/or dedicated GPU. To establish verification properties required in forensic research, we have opted to a multitude of devices:

- **Device:** HP Pavilion Gaming 15-cx0630nd

  - **Dedicated GPU:** NVIDIA GeForce GTX 1060 3GB
- **Device:** Dell XPS 17 9720
  - **Dedicated GPU:** NVIDIA GeForce RTX 3050 4GB
- **Device:** Desktop
  - **Dedicated GPU:** MSI GeForce GTX 660

Using ranging device generations allowed us to find out if our results differ per GPU. The used "Desktop" is the same machine that was used in the research of Mike Slotboom [4], and therefore allows for a side-by-side comparison.
In order to use VRAMFS on this device, the following software requirements are present:

- Linux with kernel 2.6+
- FUSE development files
- A GPU with support for OpenCL 1.2

Because the GTX 660 is an older GPU (released in 2012), we have decided to install Ubuntu 20.04 LTS on the GTX 1060 and GTX 660 systems to reduce the likelihood of NVIDIA driver incompatibilities. The machine running the RTX 3050 GPU uses Ubuntu 22.04.4 LTS with the latest NVIDIA drivers.
Due to the OpenCL backward compatibility features and the dedicated GPU being able to support OpenCL 3.0, one can assume that our setup is compatible with running VRAMFS [3].

Using the aforementioned environments. A series of experiments have been conducted. The goal of these experiments is to measure the abilities of VRAMFS forensics, independent of which version/generation NVIDIA GPU is used to use VRAMFS. Below a description is given of the various experiments that were conducted:

- **Using VRAMGrabber to dump the VRAM**: does VRAMGrabber work on the newer generations of GPUs ($\leq$ six years old) as well? If data cannot be acquired, would different driver versions alter the results?
- **Acquisition using CUDA**: even though VRAMGrabber was written in OpenCL, [4] described how CUDA can be used to program a CUDA-based version of VRAMGrabber. Are the CUDA dumps different from the OpenCL ones?
- **Forcing a CUDA GPU core dump**: newer NVIDIA devices support the generation of core dumps, induced by terminating the CUDA kernel. What information can be gathered from those core dumps? How are the results different when compared to the OpenCL-based and CUDA-based versions of VRAMGrabber?

5 Results

5.1 Using VRAMGrabber to dump the VRAM

VRAMGrabber is written in OpenCL 1.2. Mike Slotboom [4] explained how CUDA could be used instead of OpenCL for the VRAMGrabber. The researchers had two reasons to use OpenCL instead of CUDA. First of all, vendor lock-in was a concern; CUDA is a proprietary product from NVIDIA, and it only supports NVIDIA GPUs. OpenCL, however, supports GPUs from different vendors. Finally, OpenCL allows copying over VRAM that is already allocated to a program (i.e. VRAMFS), whereas CUDA does not allow this.

VRAMGrabber has been tested on both the GTX 1060 laptop and the GTX 660 desktop. In our OpenCL 3.0 setup, to preserve compatibility with the OpenCL 1.2 code, the

`CL_HPP_ENABLE_PROGRAM_CONSTRUCTION_FROM_ARRAY_COMPATIBILITY` flag has been added to the VRAMGrabber code. Given that this desktop was also used in the study conducted in 2019, it was apparent to the researchers this tool should dump the VRAM without any issues. By replicating the *Scenario 2* of Mike Slotboom [4], where VRAMFS has recently been unmounted, and hence its VRAM is in unallocated state until claimed by a different program, we were able to dump the contents of the VRAM on the GTX 660 desktop, and hence the contents of the VRAMFS.

On the GTX 1060 laptop, we were not able to dump the VRAM, however. First of all, VRAMGrabber only allocated buffers for unallocated parts of the VRAM, instead of 100% of the available VRAM. In the *Scenario 2*, this should not be an issue, as long as the previously allocated VRAM states are unallocated untill VRAMGrabber finishes.

Suspecting a theoretical issue where read-write actions are needed to gain access to the VRAM, we have patched VRAMGrabber to overwrite parts the unallocated VRAM with a random string. VRAMGrabber did manage to dump those strings, but all other data was padded out. We suspect the newer GPUs and/or newer NVIDIA drivers constrain the VRAM view to the originating process, as a means of ensuring VRAM security and confidentiality.

To rule out a difference caused by variance in the NVIDIA driver versions, versions 475 and 515 of the NVIDIA drivers have been tested. None of these drivers resulted in a successful dump on the GTX 1060 laptop.

5.2 Acquisition using CUDA

Despite the inability of CUDA to acquire allocated VRAM, CUDA could still be useful in our experiments, given that we only have NVIDIA-based GPUs. In the *Scenario 2* of Mike Slotboom [4], a CUDA-based program should be able to copy over the VRAM contents, provided that the previously allocated VRAM has not been overwritten by someone else.

As suggested by Mike Slotboom [4], we have written a program (named `CUDAGrabber`) that uses the `cudaMalloc()`, `cudaMemcpy()`, and `cudaFree()` methods to allocate most unallocated VRAM (with at least 100 megabytes left for the host), copy over the memory from the device to the host (using the `cudaMemcpyDeviceToHost` kind [5], followed by freeing up the VRAM. The source code can be found in Appendix part A.

Despite using the same kernel, operating system, and driver versions, this attempt was successful only on the system with the older GTX 660 GPU. Up to 100% of, but at least a majority of, the VRAMFS was found in the video memory dump. However, the dumps on systems with GTX 1060 and RTX 3050 GPUs contain nothing else than zeros, suggesting that the hardware itself overrides de-initialized memory.

5.3 Forcing a CUDA GPU core dump

For NVIDIA devices with compute preemption support, a user-induced core dump [6] can be generated. The `CUDA_ENABLE_USER_TRIGGERED_COREDUMP=1` environment variable creates a pipe with the following format: `corepipe.cuda.HOSTNAME.PID`. After writing anything to this pipe, the CUDA kernel will be stopped and a video memory dump will be generated.

This experiment was performed on the GTX 1060 and RTX 3050 because the older GTX 660 has no compute preemption support. After unmounting VRAMFS, a CUDA program that allocates most of the available memory was run, and a user-induced core dump was generated. In both cases, the output contains some information about the previously running process, however, the rest of the dump is filled with zeroes. This behavior suggests again that the memory is automatically overwritten by the GPU after deallocation.

6 Discussion

6.1 Padding out VRAM

The performed experiments revealed a behavioral difference between the different generations of NVIDIA GPUs. While the previous work could be reproduced using the exact setup, there is no indication that this method is effective on newer hardware (starting from the GeForce GTX 1060 series onwards).

The modern GPUs exhibit consistent results using both, CUDA and OpenCL, thus neither framework appears to automatically zeroes the memory buffers when allocating or deallocating them. Moreover, when triggering a forced core dump, the result is the same. The previous contents of the VRAM can not be extracted. Since the same driver and OS version were used, this is a strong indication that the newer GPU architectures handle memory management differently without exposing old data to newly spawned processes. No plausible explanation could be found in the change logs concerning the newer GPUs.

6.2 Using ACPI tables for raw I/O instructions on the VRAM

In all conducted experiments, we have not been able to extract the VRAMFS on the GTX 1060 and GTX 3050 devices. No root cause has been found for this.

However, in all experiments, a middleware (OpenCL and CUDA drivers) was used to communicate with the GPU. The middleware is responsible for translating the user space calls to I/O instructions over the PCIe bus. Not only could the middleware be used to block certain acquisitions, the driver code is also relevant for determining to what extent integrity can be assumed for the acquisition. In VRAMGrabber, the sole purpose of either OpenCL and CUDA is to extract the VRAM in raw form. For data acquisition, the closer we are to the data source (master data), the less likely it is an in-between tool is interfering with our results.

In a hypothetical scenario, one could write a kernel mode that directly accesses the VRAM, without needing OpenCL or CUDA. In the various Advanced Configuration and Power Interface tables found on a computer, the `DSDT` and `SSDT` tables might contain memory address maps for translating logical addresses to physical addresses in the VRAM. If the kernel module is able to access either table, the VRAM could be acquired directly by dumping the cores of all GPU kernels.

7 Conclusion

In this section, we will reflect on the results from Section 5, and on our research (sub)questions.

The first sub question in this research was: "How do existing VRAMFS parsers acquire/interpret the file system, and what effect do they have on the data integrity?". The one existing acquisition tool, VRAMGrabber, is written in OpenCL. In reality, this OpenCL tool does not *parse* or *auto-mount* the VRAMFS, this tool is merely for acquiring the VRAM of a GPU. In fact, the tool has its use for any case where some sort of data is hidden in the VRAM.

For the second sub question, "What challenges are faced with parsing VRAMFS from untested NVIDIA GPUs using existing tools?", we can conclude the following: VRAMGrabber was developed using a GPU that was released in 2012. On this GPU, VRAMGrabber works fine, but VRAMGrabber does not work on GPUs released in later years. We suspect the memory management has changed in newer GPU generations. Because we were not able to find the root cause for the behavioral change on the GPUs, we cannot advise on alternative methods to bypass the newly introduced restrictions.

Finally, we can answer the main research question: "How can acquisition be applied to VRAMFS on different NVIDIA GPUs, such that minimal data integrity compromises are present?". On newer GPU generations, we have not been able to acquire the VRAMFS or

VRAM. Replacing OpenCL with CUDA did not make a visible difference in the output. In itself, using said middleware may have impact on the data integrity. However, we were not able to define the exact impact.

8 Future work

An investigation of the possible differences in the GPU versions that lead to the previously mentioned results could be the starting point for further research. Most likely either a hardware or a firmware change is responsible for the observed behavior, thus an interesting result would consist in pinpointing it exactly.

Moreover, as explained in Section 6, we advise researching the capabilities of a custom kernel module, directly accessing the VRAM using the `DSDT` and `SSDT` tables in ACPI. These tables could be used by the module to perform a non-intrusive read operation of the exposed memory space, therefore dumping the cores of all the GPU kernels.

References

[1] eversinc33. *Abusing the GPU for Malware with OpenCL*. https://eversinc33.com/posts/gpu-malware/. [Online]. 2023.

[2] f13rce. *VRAMGrabber*. https://github.com/f13rce/VRAMGrabber. [Online]. 2019.

[3] Kronos group. *Khronos Group Releases OpenCL 3.0*. https://www.khronos.org/news/press/khronos-group-releases-opencl-3.0. [Online]. 2020.

[4] Ivar Slotboom Mike Slotboom. "Hidden storage in Digital Forensics: Storing and retrieving files using VRAM on a high performance graphics card". In: (2019).

[5] NVIDIA. *CUDA Runtime API*. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html. [Online]. 2024.

[6] NVIDIA. *CUDA-GDB*. https://docs.nvidia.com/cuda/cuda-gdb/index.html#gpu-core-dump-support. [Online]. 2024.

[7] John D Owens et al. "GPU computing". In: *Proceedings of the IEEE* 96.5 (2008), pp. 879–899.

[8] Baijian Yang Yazeed Albabtain. "GPU forensics: recovering artifacts from the GPU's global memory using OpenCL". In: *Proceedings of the Third International Conference on Information Security and Digital Forensics*. 2017, pp. 12–20.

Appendix

A CUDAGrabber source code

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

#include <cuda_runtime_api.h>

#define FILE_PATH "VRAM.raw"

__global__ void grabberKernel(void *data) {
    sleep(20);
}

int main() {
    // Disable GPU caching
    setenv("CUDA_CACHE_DISABLE", "1", 1);


    size_t free_mem, total_mem;
    cudaMemGetInfo(&free_mem, &total_mem);
    printf("Free: %ld | Total: %ld\n", free_mem, total_mem);

    size_t allocated = free_mem - 100000000; // free - 0.1GB
    printf("Allocating %ld...\n", allocated);

    // read
    void *h_buf = malloc(allocated), *d_buf;
    memset(h_buf, 0, sizeof(h_buf));
    cudaMalloc((void **)&d_buf, allocated);

    printf("Reading VRAM...\n");
    cudaMemcpy(h_buf, d_buf, allocated, cudaMemcpyDeviceToHost);

    FILE *f = fopen(FILE_PATH, "wb");
    fwrite(h_buf, allocated, 1, f);
    fclose(f);

    cudaFree(d_buf);
    free(h_buf);
    return 0;
}
```

Compile with:

```
# Without debugging symbols
$ nvcc main.c -o cudagrabber

# With debugging symbols
$ nvcc -g -G main.c -o cudagrabber
```