



Hidden storage in Digital Forensics: Storing and retrieving files using VRAM on a high performance graphics card

March 31, 2019

Ivar Slotboom

MSc. Security and Network Engineering
University of Amsterdam
ivar.slotboom@os3.nl

Mike Slotboom

MSc. Security and Network Engineering
University of Amsterdam
mike.slotboom@os3.nl

Abstract—Graphics cards currently contain a Video RAM (VRAM) of gigabytes in order to cache high resolution textures to provide extra detail in video games. Investigating this VRAM in forensic research is a relatively new field. Using APIs CUDA and OpenCL, direct interaction with the VRAM is possible and with utilizing VRAMFS, the VRAM could be turned into a mountable 'hidden' file system. This is because the VRAM is likely to be overlooked during forensic acquisition.

In this research, a forensically sound method based on two APIs are introduced and tested using a newly developed tool: VRAMGrabber. Three scenarios were found and researched, based on whether the system is on or off and whether or not VRAMFS is still mounted to the system.

It is shown possible to utilize the VRAM as file system and to retrieve written data from it when analyzing the VRAM memory dump, which was created by the tool in the test setup. Four file types (.txt, .png, .jpg and .zip) were tested using Scalpel with two different sizes (large and small), resulting in eight files per test. The extracted files were compared with the original test files using SHA256 hashes.

The accuracy of this tool was 0.5 based on the results of Scalpel, where the small files could be found and reconstructed from the dump. Larger test files (i.e. the first being 76.6 KB) could not be recovered presumably due to file scattering on the VRAM when using VRAMFS. If this appears to be correct, the accuracy of the VRAMGrabber is higher, as parts of the files could still be found in the dump using HexDump.

Keywords— Graphics Card, GPU, VRAM, VRAMFS, GPGPU, OpenCL, CUDA, Forensics, VRAMGrabber

I. INTRODUCTION

With the increasing use of digital technology, the field of digital forensics also developed increasingly. Extracting and investigating artifacts from storage and Random Access Memory (RAM) is mostly used to provide evidence in court, but using data from graphics cards of a computer could also be utilized for the same purpose [2]. This is a relatively new field, where graphics cards currently contain Video RAM (VRAM)

with gigabytes of volatile storage in order to (e.g.) cache high resolution textures to provide extra detail to a video game.

In this research, the use of this VRAM is investigated. The VRAM can be used to store volatile data that likely remains hidden if only the data from storage devices and RAM is extracted and investigated during a forensic acquisition on a system [7].

II. BACKGROUND

A. Interacting with the GPU

Usually, a graphics processing unit (GPU) is utilized for the computation of the graphics on a computer. However, when using General-Purpose Computing on Graphics Processing Units (GPGPU), the GPU is used to compute tasks a CPU would perform [3]. When this device is used in this role, interaction with the GPU is possible using application programming interfaces (APIs).

Open Computing Language (OpenCL) is a framework for platform-independent parallel programming based on C/C++ APIs [6]. OpenCL provides an abstraction to a variety of processors (e.g. CPU and GPU) as "devices". These devices contain one or more "compute cores", where instructions are executed by "processing elements" (PEs). This way, a homogeneous approach can be used to program instruction to these types of processors [12].

In OpenCL, which originated from Apple Inc., four types of memory are defined, which are depicted in Figure 1 [12]:

- 1) Large high-latency "global" memory
- 2) Small low-latency read-only "constant" memory
- 3) Shared "local" memory accessible from multiple PEs (work items) within the same compute unit (workgroup)
- 4) "Private" memory or device registers accessible within each PE (work item)

These types of memory can be used in OpenCL programs, where first a context has to be formed by one or more devices. Allocating the memory will namely be associated with the context instead of the separate devices. The OpenCL programs are compiled at runtime, where kernel functions can be activated on the devices [12].

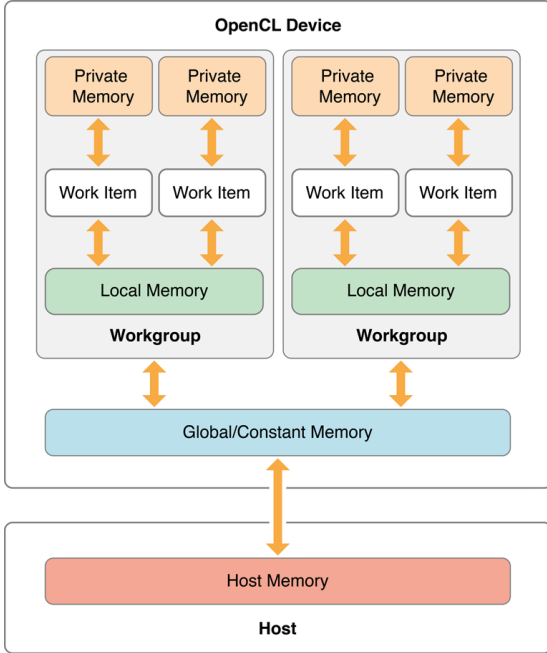


Fig. 1. OpenCL architecture [5]

Instead of using OpenCL to interact with the processors, NVIDIA Compute Unified Device Architecture (CUDA) could be used, which is limited to only NVIDIA GeForce, Quadro and Tesla graphics cards [1]. CUDA uses programs based on C/C++ to interact directly with the GPU and use this device as a GPGPU [8].

B. VRAMFS

The VRAM could be used to store files in using a file system structure. An application of this file system is called VRAMFS, which is a GitHub project from Alexander Overvoorde using at least OpenCL 1.2 [9]. In order to make the file system possible, the File System in Userspace (FUSE) library is used [9]. This library is an interface to export a user defined file system to the Linux kernel [10]. FUSE consists of the FUSE kernel module as an interface to the kernel and the libfuse library running in user mode to interact with this kernel module [10]. The overview of the implementation of VRAMFS is shown in Figure 2.

Using VRAMFS makes it possible to investigate the concept of retrieving data from the VRAM while using the structure of a user space file system.

C. Memory forensics tools

In order to investigate the memory dump, open source tools are available. Scalpel is an open source tool for extracting

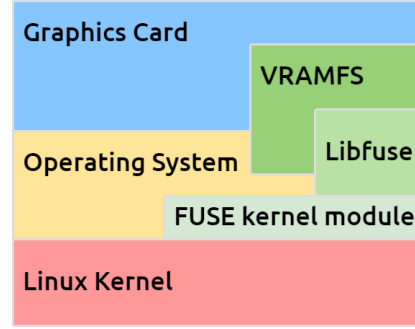


Fig. 2. VRAMFS overview

artifacts from memory based on pattern matching [11]. This makes Scalpel a suitable tool to recover files from the VRAM when a memory dump is performed and the files are not split into parts. In the configuration file, the headers and footers from the file types can be defined to search for these artifacts in a dump.

III. RELATED WORK

Boldyrev and Tykushin [2] investigated the extraction of graphic resources from the memory in a Linux system with investigation using Volatility. This research focused on reconstructing images using a Python algorithm that iterates through the memory and guesses where these images were stored. They claim to have restored a number of images from the frame buffer. This research is valuable for the project, because the possibilities of investigating the VRAM were elaborated.

A research from Albabtain and Yang [1] about retrieving images from the VRAM used a scala of types and sizes. To retrieve this files, they used the OpenCL framework. Using a forensic method, Albabtain and Yang managed to retrieve images and web pages partially from the VRAM, but this could be optimized [1]. This research shows that OpenCL could be used in a forensic setting to interact with the graphics cards and is therefore also valuable in this project.

A similar research from Lee et al. [7] was about the retrieval of images from the GPU using CUDA on NVIDIA graphic cards. In this work a four step method was introduced to perform this retrieval in a forensically sound manner. Zhang et al. [13] continued on the work of Lee et al. regarding the proposed method. In this research the forensically soundness of the method is discussed and tested using hashes of the retrieved images. The method has been verified when calculating the hashes of the data, but because the CPU and memory was used during this retrieval, it is recommended that the system memory forensics is performed first.

The concept of VRAMFS makes it possible to store files in the VRAM and not only images. During the search for related work regarding the investigation of VRAM to store 'hidden' files, no research was found to be available. However, the work of retrieving images from the GPU is useful to work upon.

IV. RESEARCH QUESTIONS

In this research, the possibility of using VRAM as a hidden storage device is examined, including the introduction of a forensic method to investigate the data on this device.

The following main research question is used to support this research: **"Is it possible to use the VRAM of a high performance GPU as a 'hidden' storage device and which forensic method can be used to investigate this?"**. Three sub questions were derived from this question:

- 1) How can files be written and read to and from the VRAM of a graphics card and is this data persistent?
- 2) Which forensically sound method can be used to extract the data from the VRAM during an acquisition?
- 3) How can the extracted data be recovered and analyzed after it has been acquired from the VRAM?

V. METHODOLOGY

The methodology consists of two parts: the approach and test setup.

A. Approach

In this research project, first the reading and writing to and from the VRAM was investigated to make 'hidden' storage possible. This investigation was supported by a test setup, using a PC that includes a NVIDIA graphics card that supports OpenCL 1.2.

The approach was structured by using VRAMFS as amounted filesystem. Another aspect in this research is investigating how persistent the data is that is stored on the VRAM. This was done by extracting the data from the VRAM using tools based on CUDA and OpenCL in the test setup. Moreover, a method was defined to read and write data to and from the VRAM of a GPU. The defined method was verified in the test setup.

When extracting the data from the VRAM, it is necessary to directly communicate with the graphics card. In order to do this, a C++ application was written named `VRAMGrabber` that uses CUDA to allocate VRAM of the device in order to copy it over to the hard disk. When testing this application, it seemed to be more efficient to use OpenCL instead, as it provided more functionality and accessibility of the data.

To test whether or not the graphics card can be used as a storage device, the following steps were taken: first the VRAM was mounted using VRAMFS, secondly files were copied to it, the VRAM was unmounted and afterwards tests were performed on it using applications written with CUDA and VRAMFS. Once the copy of the VRAM had been stored on a storage device, analysis tools were used to retrieve the files that were put on the mounted VRAM.

Once the manual tests seemed successful, automation was used to verify the consistency of the results. With these automated tests, it was also verified whether the results remained consistent if the graphics card was still mounted, or if the data persists when the computer has been rebooted.

In order to check for the results of the consistency in these automated tests, Scalpel audit logs were used to see what

files it was able to carve out, where a SHA256 hash was used to verify the integrity of the carved files. These hashes were compared to the hashes of the original files to determine false positives and to detect missing files that have been lost, scattered or (partially) overwritten in the VRAM.

B. Test setup

In order to test the interaction with the VRAM and to verify the method, a test setup is built, containing four components:

- 1) Desktop with a PCI-e 3.0 x16 slot on the motherboard
- 2) Graphics card with 2GB of VRAM
- 3) Data to be stored on the VRAM
- 4) Tools for investigation and interaction with the GPU

In Figure 3 these components are depicted in the test setup. A desktop is used containing an Intel(R) Core(TM) i7 CPU 920 @ 2.67GHz, 6 GB System RAM, Hitachi HDS5C302 2 TB hard disk and Ubuntu 18.04.2 LTS (desktop) installed on it.

The graphics card used in this system is a MSI NVIDIA GeForce GTX 660 Twin Frozr (OC edition) with 2GB of GDDR5 VRAM, at the speed of 6,008 GHz. NVIDIA driver version 418.39 for Linux was used in this environment, resulting in support for OpenCL 1.2.

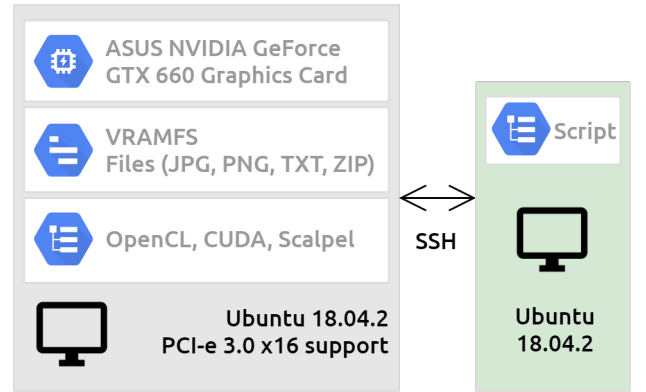


Fig. 3. Test Setup

Because the system with the graphics cards was running an Ubuntu desktop environment, the tests were performed via an SSH connection by using another machine. This way, the screen of the system remained unchanged and the script performed bash commands in the background of the system, without altering the VRAM by locally interacting with the system.

The software used in this research project is limited available on [the project's GitLab repository](#). The VRAMGrabber tool based on OpenCL 1.2 (and above) has made public on [the tool's GitHub page](#).

VI. RESULTS

In this section the results of our research are described.

A. Reading and writing files on GPU

In order to read and write files to and from the VRAM of the GPU, a structured approach is chosen by using VRAMFS to mount the VRAM as a file system. This way, the VRAM of the GPU was usable in the Ubuntu file system and as long as the VRAM stayed 'mounted', interaction with this VRAM (e.g. reading and writing files) was possible. If the VRAM was 'unmounted' or the system was shut down, the data was not available and retrievable from the file system any more.

1) *Persistence of the data on GPU:* When VRAMFS was started again, the offset of this file system in the VRAM had shifted, resulting in the disappearance of the previously used file and directory tree. This shows the limited persistence of this data of using VRAMFS when mounting and unmounting, where the stored data can be directly overwritten after the allocation of the VRAM is released.

Next to using VRAMFS, files could also be read and written using the write buffers of CUDA and OpenCL. This way, the VRAM is not mounted and there is no VRAM reserved to store this data for a longer period, but interaction is still possible. This results in the storing of volatile data, where the offsets are key to find the location of the written files.

When the VRAMGrabber was used in this situation, the data from the dump can contain the files that are written to the VRAM, unless this data is overwritten by another program, due to the lack of allocated VRAM memory space.

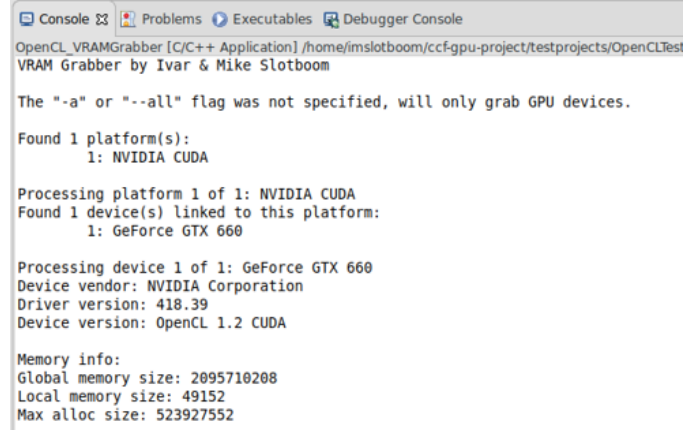
2) *Retrieving data from VRAM with VRAMGrabber:* With the VRAMGrabber tool that was created during this research, it was possible to allocate VRAM and copy the raw data that currently resides on the VRAM. This process was done by using small blocks to ease the allocation process and copying. After the data has been copied into a temporary buffer, this data could be written to a file on the hard disk.

When the VRAMGrabber was using CUDA, this process was applied where this tool was able to copy over the raw existing data from the VRAM, but this was only possible for unallocated memory spaces of the VRAM. Any space that was already allocated by other applications or the operating system could not be captured, which could lead to loss of evidence.

Another key point to note is that CUDA could only provide support for NVIDIA devices, so if a wider range of different device vendors is used, OpenCL had to be used instead of CUDA.

When using OpenCL instead for the VRAMGrabber application, the tool was able to grab the data of the allocated memory spaces in a slower, but consistent way due to fencing and global synchronization mechanics. Because it was possible to copy over allocated memory spaces, no evidence can be obstructed afterwards by other applications and the operating system.

Another positive aspect was that it was easier to grab both platform and device information, such as the vendor, supported OpenCL version and driver version, which is shown in Figure 4. With this technique it is still required to have OpenCL at version 1.2 or higher.



```
OpenCL_VRAMGrabber [C/C++ Application] /home/imslotboom/ccf-gpu-project/testprojects/OpenCLTest
VRAM Grabber by Ivar & Mike Slotboom

The "-a" or "--all" flag was not specified, will only grab GPU devices.

Found 1 platform(s):
  1: NVIDIA CUDA

Processing platform 1 of 1: NVIDIA CUDA
Found 1 device(s) linked to this platform:
  1: GeForce GTX 660

Processing device 1 of 1: GeForce GTX 660
Device vendor: NVIDIA Corporation
Driver version: 418.39
Device version: OpenCL 1.2 CUDA

Memory info:
Global memory size: 2095710208
Local memory size: 49152
Max alloc size: 523927552
```

Fig. 4. VRAMGrabber using OpenCL

OpenCL did apply the same process as CUDA, where memory spaces were allocated and data was copied over into a temporary buffer. The main difference with OpenCL is that a kernel instruction had to be created to gain access to the graphics device in the first place.

Since it was not needed to utilize the kernel itself, but rather only a read function from the graphics device, the method was still straightforward by performing a queued read buffer that retrieved the raw data.

B. Forensic method for retrieving data from GPU

The data is stored on the GPU is volatile and therefore prone to alteration. Using a forensic method to retrieve this volatile data, the impact on this data has to be brought down to a minimum, resulting in a reliable as possible output, which can be used in court cases.

This method starts with live forensics, when a system is on and an investigator starts the acquisition. First, the investigator has to determine which graphics card is in place in the system and if this card supports CUDA or OpenCL. This will determine if OpenCL or CUDA is used to extract the data from the VRAM.

Using VRAMFS, different scenarios can be identified when applying live forensics, based on the status of the system and VRAMFS, when the system is acquired. The order of acquisition steps in this live forensics approach is based on the volatility of data: System RAM, VRAM and disk [13].

First the System RAM has to be secured. This can be done by RAM dumping tools, which will not be covered in this research. After the acquisition of the System RAM, the next step is dumping the VRAM. The variables in this situation are whether the system is on or off and whether VRAMFS is mounted to this system. This leads to three scenarios:

1) *Scenario 1: VRAMFS is mounted and the system is on:* If the system is on, the procedure for live forensics will be started by the investigator. Instead of only applying memory forensics in this situation, the methods for disk forensics can

be used by dumping the entire file system to an image, which can be used to investigate later in the process.

When an image is made from an existing disk, the mounted 'VRAM disk' is included, supposing that the system stays on and VRAMFS is mounted.

In Figure 5 the flow of this procedure is depicted. Because the acquisition of the VRAM depends on programs, which are run using the CPU and System RAM, the acquisition of the System RAM has to be performed first [13]. After this acquisition is finished, the file systems on the disks will be imaged.

It is necessary that this happens using live forensics when the system is running, because otherwise, the mounted VRAM 'disk' will not be imaged. This happens when the system is off and the disk is imaged using 'dead forensics' [4].

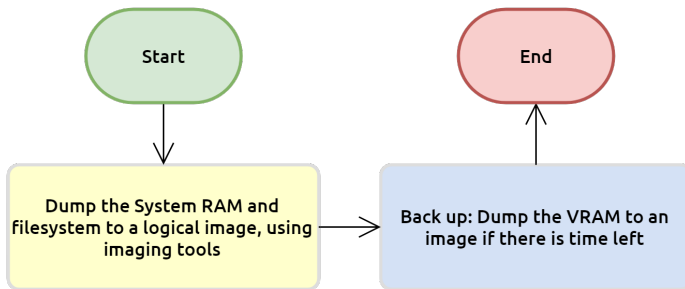


Fig. 5. Flow 1

Because in this scenario the creation of a VRAM dump is not necessary, the steps in this method are not detailed. In the following scenarios, the steps for imaging the VRAM are discussed further.

2) *Scenario 2: The VRAMFS is (recently) unmounted and the system is on:* If the system is on and VRAMFS is not mounted anymore, the previous approach will not suffice. In this situation, memory forensics is needed to dump the VRAM to an image. Using this method to secure the data, the volatility of this 'disk' has to be taken into account [13]. Because it is possible to interact with the VRAM of the GPU using OpenCL and CUDA, applying live memory forensics to this VRAM should be possible. The approach of Lee et al. and Zheng et al. makes use of CUDA to retrieve data from the VRAM, consisting of four steps [7][13]:

- 1) Get GPU memory space information using `cudaMemGetInfo()`
- 2) Get access to the data of closed programs and the location it was freed from by allocating the entire memory space using `cudaMalloc()`
- 3) Copy all the data in GPU global memory using the `cudaMemcpy()`
- 4) Free the memory space allocated by using the `cudaFree()`

Instead of CUDA, OpenCL can be used in this method, which has a slightly different approach. When using OpenCL, the devices and the context have to be determined first, to

be able to run code on the devices. It is then possible to dump the contents of the VRAM to a file using a C++ script and a OpenCL enabled kernel. This offers a broader range of graphics cards, because not only NVIDIA cards have OpenCL enabled, but also AMD and Intel cards. If a driver for this language is installed, the acquisition of the data can be started. The following steps apply when OpenCL is used:

- 1) Get platforms and select GPU device as the context using `clGetDevicePlatforms()`, `clGetDevices()` and `clContext()`
- 2) Create a buffer to interact with the data on the device using `clBuffer()`
- 3) Copy all the data in GPU global memory using `clEnqueueReadBuffer()`
- 4) Free the memory space allocated by finishing the OpenCL program

This live memory forensics approach was applied into the procedure, which is shown in Figure 6. After the System RAM is secured in an image, the contents of the VRAM are extracted using the four step method. Afterwards, the disk can be imaged.

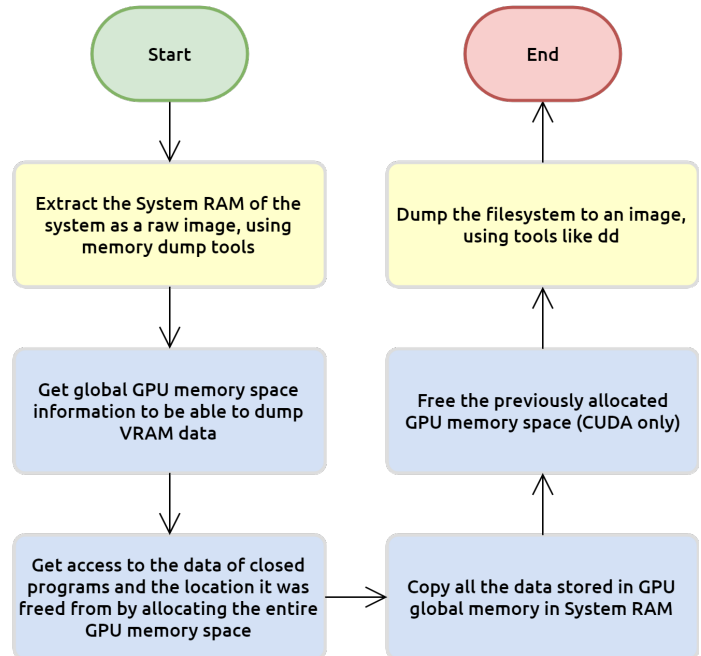


Fig. 6. Flow 2

3) *Scenario 3: The system is off, hence the VRAMFS is unmounted:* If the system is off, it means that the VRAM disk is also not mounted to the file system. In this scenario, it is hard to perform a proper dump of the VRAM, because this data is not persistent on the VRAM and therefore the quality of the data cannot be guaranteed.

If the system is booted up, the VRAM will be used to create the desktop environment and therefore the stored data on the VRAM can be easily overwritten, which results in the loss of the data or only parts are still available.

Because proper live memory forensics can be used to recover parts in memory that are still available, the method used in this situation is the same as depicted in Figure 6. This will not guarantee the retrieval of all the data. 'Dead' disk forensics can be performed to make an image of the disk, except the mounted VRAMFS.

C. Recovering and analysis of files in VRAM

The proposed methods were used in the test setup, which made an image of the VRAM of the NVIDIA graphics card, when utilizing the VRAMGrabber based on OpenCL. Using Scalpel, HexDump and Strings the contents of this file can be extracted and investigated. In the test setup, automated test scripts were used to verify the reliability of the results of the three scenarios: mounted, unmounted and rebooted. The following steps were taken to investigate these scenarios, where each step was timestamped in the script:

- 1) Allocate the VRAM for VRAMFS
- 2) Overwrite the VRAM with zeroes first in order to start the test in a clean manner
- 3) Copy test files to the mounted VRAMFS
- 4) Deallocate the VRAMFS
- 5) Run the OpenCL VRAMGrabber tool
- 6) Analyze the raw VRAM data with Scalpel, Strings and HexDump
- 7) Calculate the SHA256 hash of carved files and compare it to the original files

In order verify the retrieval of files, eight test files have been written to the VRAM in the test setup, which are described in Table I. Four extensions were used in two file sizes Small (i.e. 'S') and Large (i.e. 'L') to test multiple file types: .txt, .png, .jpg and .zip.

TABLE I
TEST FILES USED IN THE TEST SCENARIOS

File Name	File Type	Size
JPG_Small.jpg (S)	.jpg	42.3 KB
JPG_Large.jpg (L)	.jpg	2263.3 KB
PNG_Small.png (S)	.png	125.4 KB
PNG_Large.png (L)	.png	454.4 KB
TXT_Small.txt (S)	.txt	2.2 KB
TXT_Large.txt (L)	.txt	76.6 KB
ZIP_Small.zip (S)	.zip	3.0 KB
ZIP_Large.zip (L)	.zip	503.5 KB

In the configuration file of Scalpel the headers and footers of the file types were altered to only retrieve the mentioned eight files. When running the 50 tests, the retrieved files were automatically verified with the original files, using the calculation of the SHA256 hash of the files.

The results of the three tests are shown per scenario in Figure 7. In this figure True Positives (i.e. SHA256 hash corresponds with hash of original file), False Positives (i.e. SHA256 does not correspond with original) and False Negatives (i.e. file is not recognized by Scalpel) are used. A True Negative is not part of the results of Scalpel.

The blue line shows the maximal amount of True Positives, when running the tests, which corresponds with a 100%

Test results Scalpel (per file type and approach)

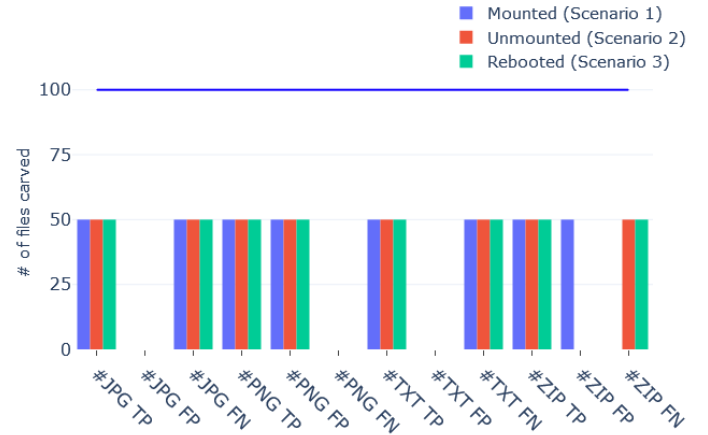


Fig. 7. Test results Scenarios

success score. In this diagram, the results of the three scenarios show a similar pattern, where Scenario 1 is an exception with False Positives when extracting the ZIP-files.

In order to calculate the accuracy of the results in this test, the following formula is used based on True Positives, True Negatives, False Positives and False Negatives:

$$\frac{(TN + TP)}{(TN + TP + FN + FP)}$$

[14]. This formula divides the number of correct test results from Scalpel by the number of all tests. The overall accuracy over the three test was 0.5, because from the 1200 files that could be retrieved, 600 files were reconstruction, 200 files were false positives and 400 files were not found by Scalpel (i.e. False Negatives).

1) *Test results Scenario 1:* In the first scenario, the VRAMFS was mounted when the VRAMGrabber was used to dump the VRAM. The dumping of the System RAM and file system were out of scope in this test. The test was performed 50 times, following the previously defined steps. Only in this test, the deallocation of the VRAMFS was scheduled after running the OpenCL VRAMGrabber tool. This was done to keep the VRAMFS mounted while dumping the VRAM.

The accuracy of the results are shown in Table II. In this table the True Positives (TP) and False Positives (FP) are shown and the accuracy is calculated. The accuracy of this test is: 0.5.

TABLE II
ACCURACY TEST RESULTS SCENARIO 1

File Type	TP	FP	FN	Accuracy
.jpg	50 (S)	0	50 (L)	0.5
.png	50 (S)	50 (L)	0	0.5
.txt	50 (S)	0	50 (L)	0.5
.zip	50 (S)	50 (L)	0	0.5
Total	200	100	100	0.5

2) *Test results Scenario 2:* In the second scenario, the defined steps were followed 50 times. This resulted in the values in Table III, where the overall accuracy is: 0.5. This shows a higher accuracy then when the VRAM is still mounted during the acquisition of the data.

TABLE III
ACCURACY TEST RESULTS SCENARIO 2

File Type	TP	FP	FN	Accuracy
.jpg	50 (S)	0	50 (L)	0.5
.png	50 (S)	50 (L)	0	0.5
.txt	50 (S)	0	50 (L)	0.5
.zip	50 (S)	0	50 (L)	0.5
Total	200	50	150	0.5

3) *Test results Scenario 3:* Scenario 3 consisted of re-booting the system before the defined steps were taken. This resulted in the retrieval of data that still existed in the VRAM after the system was booted up again. The results of this test are shown in Table IV and the accuracy of this test was 0.5, which is the same as the previous results.

TABLE IV
ACCURACY TEST RESULTS SCENARIO 3

File Type	TP	FP	FN	Accuracy
.jpg	50 (S)	0	50 (L)	0.5
.png	50 (S)	50 (L)	0	0.5
.txt	50 (S)	0	50 (L)	0.5
.zip	50 (S)	0	50 (L)	0.5
Total	200	50	150	0.5

VII. DISCUSSION

In this research, the reading and writing to the VRAM of a graphics card have been confirmed using VRAMFS. The tool VRAMGrabber was developed to dump the contents of this VRAM after writing to it. In order to image the VRAM with VRAMGrabber, the graphics card has to support either CUDA or OpenCL. This brings requirements to this approach, but if NVIDIA cards are used and the NVIDIA drivers are installed, CUDA is supported on these graphics cards. To use OpenCL, additional drivers may be required, based on the brand of the device (e.g. NVIDIA, AMD and Intel). However, when VRAMFS is used, OpenCL 1.2 is minimal required, which means the GPU has to have support for OpenCL in the first place.

Using the test setup with a single NVIDIA GeForce GTX 660, an accuracy of 0.5 from the Scalpel results was reached. This indicates how well the test files can be recovered from the VRAM dump in one piece and when using a graphics card which is older than 5 years. Based on the support for OpenCL and CUDA on the graphics cards, which are introduced after the card used in the test setup, the VRAMGrabber should work on these graphics cards as well.

In the three scenarios, the small files per file type were successfully found by Scalpel and reconstructed. On the other hand, when trying to extract the larger files, these files could not be found or reconstructed. The 'large' text file of 76.6

KB was not found in the VRAM dump of one of the three scenarios. This shows that file size matters when writing to the VRAM using VRAMFS, because these files could be scattered over the VRAM, which is not recognized by Scalpel.

When the raw data dump from the tests was inspected using HexDump, parts of the 'larger' files were found in the data, which supports this assumption. If this assumption appears to be true, the accuracy of this tool is higher as the scattered files are also available in the VRAM dump.

The development of a method to locally start the VRAM-Grabber on the suspect's system using an auto-run script (e.g. from an investigator's USB drive) was out of scope in this research. Instead, the tests were performed using SSH to minimize the impact of user interaction on the system, which could lead to overwritten data on the VRAM.

When a forensic investigation is started including the retrieval of the VRAM data, the approach on this system should also be taken into account. Moreover, the data, which is dumped by the VRAMGrabber has to be stored in an additional disk or USB key, in order to minimize the impact of running this tool on the disk. If this data is stored on the local disk on the system, this could interfere with the investigation if the disk is imaged afterwards.

If the issues of API support and launching the VRAM-Grabber automatically are resolved, the proposed methods could be accepted as forensically sound, because only the kernel instructions in the APIs are used when dumping the VRAM data. The utilization of the VRAMGrabber will not alter the data on the VRAM itself, which was shown in the test setup. This shows the significance of this research in forensic investigation.

Because the VRAMGrabber based on OpenCL performs a dump of the whole VRAM, this dump can also be used to retrieve pictures, which are shown on the screen. This approach originated from the related research, but this shows also the applicability of the tool. In this tool, the OpenCL supported GPU is selected as device, but also the use of the CPU is possible if this CPU has also support for OpenCL. When the CPU is selected, the System RAM will be dumped, but if this tool is executed, this System RAM will also be used to store the temporary dump in a buffer.

VIII. CONCLUSION

The following research question is answered in this research: *"Is it possible to use the VRAM of a GPU as a 'hidden' storage device and which forensic method can be used to investigate this?"*.

Based on the results, it is possible to use the VRAM as 'hidden' storage using VRAMFS, where the VRAM is mounted to the file system. The reliability of writing to the VRAM has been verified in a test environment. Three scenarios were identified, regarding whether the system was running and whether the VRAMFS was mounted. Based on these scenarios, two forensic methods have been introduced, which places the retrieval of VRAM data after System RAM during live forensics.

It was possible to retrieve the written data, when an image was made from the VRAM using VRAMGrabber. This tool was developed during this research and had two implementations based on two APIs: CUDA and OpenCL. The main difference between the two APIs is that VRAMGrabber using OpenCL was able to dump the VRAM as a whole, where CUDA could only image the unallocated memory space of the VRAM.

After the tool had dumped the VRAM, the copied files could be retrieved and reconstructed using Scalpel, where the SHA256 of the files corresponded with the hash of the original test files. This shows that the data was not altered and artifacts could still be recovered even though the VRAMFS was unmounted. The accuracy of this tool was 0.5 based on the results of Scalpel, where small files could be found and reconstructed from the dump. 'Larger' test files (i.e. the first being 76.6 KB) could not be recovered presumably due to file scattering on the VRAM when using VRAMFS. If this appears to be correct, the accuracy of the VRAMGrabber is higher, as parts of the files could still be found in the dump using HexDump.

IX. FUTURE WORK

Although this research shows that retrieving 'hidden' data from VRAM is possible, there is still some future work left. The VRAMGrabber could be tested in other scenarios, using different brands of GPUs, different file types, different file systems and different operating systems. This way, the presumed file scattering could be further investigated.

The impact of using the proposed methods could be further investigated. This way, the use of this tool in real forensic investigations could be considered, where the forensically soundness is verified. The automatic launch of the VRAM-Grabber on a suspect's system, when a graphics card is identified during acquisition, has to be developed and is therefore future research.

Three scenarios are introduced in this research and this could also be extended with research in the persistence of written data, when the system is off for a longer period of time (e.g. a day or a week). Only 50 tests per scenario were conducted in the test setup, which can be expanded in a test to verify the results.

REFERENCES

- [1] Y. Albabtain and B. Yang. Gpu forensics: Recovering artifacts from the gpus global memory using opencl. In *The Third International Conference on Information Security and Digital Forensics (ISDF2017)*, page 12, 2017.
- [2] M. Boldyrev and A. Tykushin. Graphical processor unit memory acquisition, 2017.
- [3] J. Ghorpade, J. Parande, M. Kulkarni, and A. Bawaskar. Gpgpu processing in cuda architecture. *arXiv preprint arXiv:1202.4347*, 2012.
- [4] M. Grobler and B. Von Solms. Live forensic acquisition as alternative to traditional forensic processes. In *4th International Conference on IT Incident Management & IT Forensics*, 09 2008.
- [5] Apple Inc. Memory objects in os x opencl. https://developer.apple.com/library/archive/documentation/Performance/Conceptual/OpenCL_MacProgGuide/OpenCLLionMemoryObjects/OpenCLLionMemoryObjects.html, 2019. Accessed on March 9th 2019.
- [6] The Khronos Group Inc. Opencl overview - the khronos group inc. <https://www.khronos.org/opencl/>, 2019. Accessed on February 20th 2019.
- [7] S. Lee, Y. Kim, J. Kim, and J. Kim. Stealing webpages rendered on your browser by exploiting gpu vulnerabilities. In *2014 IEEE Symposium on Security and Privacy*, pages 19–33. IEEE, 2014.
- [8] NVIDIA. Programming guide :: Cuda toolkit documentation. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>, 2019. Accessed on March 5th 2019.
- [9] A. Overvoorde. Github - overv/vramfs: Vram based file system for linux. <https://github.com/Overv/vramfs>, 2019. Accessed on February 18th 2019.
- [10] N. Rath. Github - libfuse/libfuse: The reference implementation of the linux fuse (filesystem in userspace) interface. <https://github.com/libfuse/libfuse>, 2019. Accessed on February 20th 2019.
- [11] G. Richard and L. Marziale. Github - sleuthkit/scalpel: Scalpel is an open source data carving tool. <https://github.com/sleuthkit/scalpel>, 2019. Accessed on February 21th 2019.
- [12] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.
- [13] M. Rogers Y. Zhang, B. Yang and R. A. Hanse. Digital forensics and cyber crime: 7th international conference, icdf2c 2015, seoul, south korea, october 6-8, 2015. revised selected papers. In *Digital Forensics and Cyber Crime*, volume 157 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 53–66. Springer International Publishing, 01 2015.
- [14] W. Zhu, N. Zeng, and N. Wang. Sensitivity, specificity, accuracy, associated confidence interval and roc analysis with practical sas implementations. <https://www.lexjansen.com/nesug/nesug10/hl/hl07.pdf>.