Forensic Science International: Digital Investigation 49 (2024) 301760

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS USA 2024 - Selected Papers from the 24th Annual Digital Forensics Research Conference USA

A step in a new direction: NVIDIA GPU kernel driver memory forensics

Christopher J. Bowen [a,b,*], Andrew Case [b,c], Ibrahim Baggili [a,b], Golden G. Richard III [a,b]

[a] School of Electrical Engineering & Computer Science, Louisiana State University, USA
[b] Center for Computation and Technology, Louisiana State University, USA
[c] Volatility Foundation, USA

ARTICLE INFO

Keywords:
NVIDIA
GPU forensics
Memory forensics
NVOC
Linux
Volatility
GPU-Assisted malware

ABSTRACT

In the ever-expanding landscape of computation, graphics processing units have become one of the most essential types of devices for personal and commercial needs. Nearly all modern computers have one or more dedicated GPUs due to advancements in artificial intelligence, high-performance computing, 3D graphics rendering, and the growing demand for enhanced gaming experiences. As the GPU industry continues to grow, forensic investigations will need to incorporate these devices, given that they have large amounts of VRAM, computing power, and are used to process highly sensitive data. Past research has also shown that malware can hide its payloads within these devices and out of the view of traditional memory forensics. While memory forensics research aims to address the critical threat of memory-only malware, no current work focuses on video memory malware and the malicious use of the GPU. Our work investigates the largest GPU manufacturer, NVIDIA, by examining the newly released open-source GPU kernel modules for the development of forensic tool creation. We extend our impact by creating symbol mappings between open and closed-source NVIDIA software that enables researchers to develop tools for both "flavors" of software. We specifically focus our research on artifacts found in RAM, providing the foundational methods to detect and map NVIDIA Object Compiler Structures for forensic investigations. As a part of our analysis and evaluation, we examined the similarities between open-and-closed kernel modules by collecting structure sizes and class IDs to understand the similarities and differences. A standalone tool, NVSYMMAP, and Volatility plugins were created with this foundation to automate this process and provide forensic investigators with knowledge involving processes that utilized the GPU.

1. Introduction

Graphics Processing Units (GPUs) are one of the most essential types of computing technology in both personal and commercial computing, experiencing rapid growth driven by advancements in artificial intelligence (AI), High Performance Computing (HPC), and 3D graphics rendering. Over the past decade, GPUs have become integral parts of personal computers. With this development, more forensic investigations will involve one or more GPUs.

Currently, the GPU market is dominated by three primary manufacturers: NVIDIA, AMD, and Intel. At the time of writing, NVIDIA is currently the largest manufacturer, holding 84 % of the GPU market (Peddie 2023). In 2023, NVIDIA's market capitalization passed one trillion for the first time, making it one of the five trillion-dollar USD companies in the technology market (Apple, Microsoft, Alphabet, Amazon, and NVIDIA) (Reuters 2023). While NVIDIA is one of the world's largest companies, there is little research involving the use of the GPU for malicious intentions and even less for forensics regarding a GPU.

As GPUs continue to become a commodity for customers, forensic concerns arise surrounding the substantial computation power a GPU can provide for specific tasks and the kernel level trust the operating system provides to the device. Advanced malware/rootkits can abuse the GPU and even hide valuable evidence within Video Random-Access Memory (VRAM), avoiding Antivirus (AV). While there is no known "wild" malware that hides within the GPU, nation-state attacks could utilize the GPU to become undetectable. Currently, no one is looking into this possibility, and in our work, we aim to start to address this threat.

Previous research has only scratched the surface of valuable information that can be found in the GPU ecosystem. Our work aims to address this gap by conducting the first peer-reviewed analysis of NVIDIA kernel modules on Linux-based systems. Additionally, we present methods to identify and extract NVIDIA Object Compiler (NVOC)

* Corresponding author.
E-mail addresses: cbowe13@lsu.edu (C.J. Bowen), andrew@dfir.org (A. Case), ibaggili@lsu.edu (I. Baggili), golden@cct.lsu.edu (G.G. Richard).

https://doi.org/10.1016/j.fsidi.2024.301760

Available online 5 July 2024

structures for both open and closed-source modules, offering symbol mappings between drivers to allow for future GPU forensic tools and research.

Our contributions are as follows:

- We present the first peer-reviewed analysis of NVIDIA's kernel modules/drivers on Linux systems.
- We provide methods to identify and extract NVOC structures for **both Open and Closed-source Modules.**
- We provide mappings and memory snippets[1] of NVOC Class Definitions structures between open and closed-source NVIDIA drivers to allow for future memory forensic tools and works focused on GPU Forensics.
- We created multiple open-source plugins[2] for Volatility to parse important artifacts out of memory for an investigation.
- We created NVSYMMAP,[3] a Python 3.0 tool, to automate the entire process of mapping NVOC Class Definitions structures between new Open and Closed Source Modules.

Our work aims to reveal how valuable artifacts can be found within a system's Random-Access Memory (RAM) for NVIDIA GPUs and provide industry tools for both open and closed-source environments.

2. Motivations and goals

Until recently, NVIDIA's code was primarily closed-source, making the creation of forensic tools nearly impossible because of the enormous amount of reverse engineering required to understand how the software operates. However, in May 2022, NVIDIA released open-source GPU Kernel modules under dual GPL/MIT licenses that allow users to opt into (Cherukuri et al., 2023). This change is a pivotal step toward enhancing the utilization and security of NVIDIA GPUs on Linux. However, despite this progress, a critical limiting factor still exists: most users will still utilize closed-source drivers.

To address this limitation, our work aims to understand the inner-workings of both kernel modules, how structures are laid out in memory, and what type of memory to look in – RAM or VRAM. If we can parse vital information to determine if a process used the GPU maliciously and what it was trying to accomplish, then investigators will have a greater understanding of what occurred during an incident.

We know GPUs will commonly transfer information between RAM and VRAM. By examining the drivers of the system's GPU, we can begin to understand how memory management and translation occur and leverage this to find forensic evidence. We can examine NVIDIA's kernel module, stored in RAM, to extract the necessary system information regarding the GPU for an investigation.

While past research has focused on examining VRAM, we believe by examining the contents of RAM, we can start to develop forensic tools to detect GPU-assisted malware and standalone GPU malware. Our research aims to provide the foundation for comprehensive forensic methods and tools capable of extracting artifacts from RAM for any version of NVIDIA Linux drivers.

3. Background

This section provides background knowledge for the rest of the paper, including an introduction to Linux Kernel Modules, NVIDIA Kernel Modules, and NVOC Structures.

3.1. Linux kernel modules and Kallsyms

Linux kernel modules are executables that can be dynamically loaded and unloaded into kernel space when the system runs. These modules can extend the kernel's functionality by implementing interfaces for devices as drivers. Each module serves a specific purpose and can export symbols through Kallsyms.

Kallsyms, the Linux kernel symbol table, is a data structure that contains information about code within kernel space, such as the address of functions and structures in memory. Kallsyms displays the dynamically loaded address of each symbol, which can be utilized to locate essential structures in the kernel memory space and parse associated data. Kallsyms is exported to userspace via **/proc/kallsyms**.

3.2. NVIDIA kernel modules

NVIDIA currently provides two distinct "flavors" of kernel drivers for Linux-based operating systems – open source and closed source. Each version of the drivers helps provide the kernel with an interface to access and utilize the GPU. When an NVIDIA driver is installed on a Linux-based system, four distinct kernel modules are loaded into kernel space:

- **nvidia:** The main NVIDIA Kernel module we investigate in this work.
- **nvidia_modeset:** The NVIDIA Kernel module that handles the mode setting of the GPU.
- **nvidia_drm:** The NVIDIA kernel module that handles the Direct Rendering Manager.
- **nvidia_uvm:** The NVIDIA kernel module that handles Unified Virtual Memory.

These modules implement interfaces provided by the Direct Rendering Manager (DRM), drm_kms_helper, and Video kernel modules. They also provide interfaces to userland processes for accessing the GPU. To list these modules, users can run lsmod and grep for "nvidia". In this work, we exclusively examine Nvidia's 525 drivers; however, our methods extend to future versions of the drivers.

3.3. NVIDIA object compiler

NVIDIA's kernel modules use NVOC for a large portion of their driver code base. NVOC is a preprocessor that allows NVIDIA to add specific metadata to the headers of structures to allow for lookups, feature toggle flags, and specific chip behaviors. NVIDIA uses NVOC in both their open and closed-source kernel modules for Linux and Windows drivers. NVOC code generator is a fork of Clang 3.X and is currently a closed-source tool used within NVIDIA (Tijanic 2022). NVOC follows the general structure of C++, implementing a Run-Time Type Information (RTTI) structure for each object. Within each NVOC_RTTI structure (Listing 3) is a pointer to a Class Definition structure, which can be used to map symbols between open and closed source modules.

In the open-source kernel modules, NVOC files are found in **/src/nvidia/generated/**. Files with the endings **_nvoc.c** and **_nvoc.h** were pre-compiled using NVOC. These files contain important information for creating memory forensics tools relating to GPUs and can be used to understand NVIDIA's ecosystem. In Source Code Analysis and Method Creation, we expand upon this background knowledge to explain how NVOC is implemented and can be used to locate and map open-to-closed source structures.

4. Methodology

This section describes our methodology for examining NVIDIA's source code and creating forensic tools. We expand on our work by explaining our methods to locate and parse NVOC structures for both open and closed-source Nvidia drivers.

Our methodology follows:

[1] https://github.com/LSUACL/GPU-Forensics/tree/main/memory-snippets.
[2] https://github.com/LSUACL/GPU-Forensics/tree/main/plugins.
[3] https://github.com/LSUACL/GPU-Forensics/tree/main/NVSYMMAP.

1. Source Code Analysis
2. Memory Acquisition
3. Memory Analysis
4. Method Creation

4.1. Source code analysis

NVIDIA's open-source drivers can be downloaded from their GitHub repository.[4] We manually analyzed the structure of the source code to understand and identify code patterns we could utilize to locate structures in memory. After reviewing the overall architecture of the codebase, we determined a substantial amount of the software could be covered by focusing on the OS-agnostic and auto-generated code.

A significant number of these files and structures utilized NVOC. NVOC structures follow a unique layout that can be utilized to map structures in memory and between each module. Each NVOC structure has a unique CLASSID that can be used to map and identify data structures. An example of a CLASSID declaration from the open-source code can be found in Listing 1.

Listing 1. Example of NVOC ClassID Declaration

```
typedef struct GpuAccounting GpuAccounting;

__nvoc_class_id_GpuAccounting 0x0f1350;
```

Each of these CLASSIDs are held within a unique NVOC_CLASS_DEF structure in the NVOC_CLASS_INFO member (Listing 6 Line 3). These class definition structures are directly exported through **/proc/kallsyms**, allowing the ability to locate them after a memory sample has been collected. In these structures, important information, such as the size of the structure, RTTI provider ID, and name (if the NV_PRINTF_STRINGS_ALLOWED is set), is included. With each class definition symbol mapped, we can use the method described in Reverse NVIDIA Object Lookup to locate any NVOC structure in memory.

Listing 2. NVOC_CLASS Definition Structure

```
struct NVOC_CLASS_DEF{
    // contains classId, size, and name
    const NVOC_CLASS_INFO classInfo;

    const NVOC_DYNAMIC_OBJ_CREATE objCreatefn;
    const struct NVOC_CASTINFO *const
    pCastInfo;
    const struct NVOC_EXPORT_INFO const
    pExportInfo;
};
```

Each NVOC_CLASS_DEF structure also has an associated NVOC_RTTI structure that points to it (Listing 3). This pointer is the first member of the RTTI structure (Listing 3 Line 2). These NVOC_RTTI structures are also unique to each NVOC structure and can be used in mapping NVOC structures.

Listing 3. NVOC_RTTI Definition Structure

```
struct NVOC_RTTI{
    const struct NVOC_CLASS_DEF *const
    pClassDef;
    const NVOC_DYNAMIC_DTOR              dtor;
    const NvU32                          offset;
};
```

In Method Creation, we explain how we use NVOC's structure format to map symbols and structures from open to closed-source modules.

Listing 4. Nvidia Symbols From Open and Closed Source Software

```
# sudo cat /proc/kallsyms | grep [nvidia]
Closed Source:                                  Open Source:
<snip>
      r _nv001945rm  [nvidia]   ----------      r __nvoc_class_def_DispChannel [nvidia]
      r _nv002176rm  [nvidia]   ----------      r __nvoc_class_def_P2PApi      [nvidia]
      r _nv002136rm  [nvidia]   ----------      r __nvoc_class_def_OBJOS       [nvidia]
      r _nv002246rm  [nvidia]   ----------      r __nvoc_class_def_VideoMemory [nvidia]
      r _nv002112rm  [nvidia]   ----------      r __nvoc_class_def_OBJGVASPACE [nvidia]
<snip>
```

4.2. Memory acquisition

To properly assess NVIDIA's GPUs memory footprint, we needed to collect physical memory samples because GPUs are not easily virtualized and, in most cases, are run on physical hardware. In future work, we aim to explore NVIDIA's Virtual GPU Software; however, in this work, all memory samples acquired were with Surge Collect Pro,[5] a physical memory sample acquisition tool.

We created two testing environments that included the same NVIDIA GPU and operating system. We then installed each flavor of the drivers (open and closed) and verified they were in use. After the drivers were loaded into memory, we took physical memory images of the systems so we could inspect each driver for NVOC structures. A detailed apparatus of devices and software for our research is displayed in Table 1.

4.3. Memory analysis

To analyze each of the memory samples, we decided to use Volatility[6] 2.6 because it is open-source and widely available. The Volatility Framework is a collection of volatile memory tools that offer investigators insight into the current state of a machine at acquisition and can be used to extract digital artifacts from volatile memory.

We primarily utilized the Linux volshell plugin to navigate memory dumps to search for NVOC structures. We determined many of these NVOC structures were in use for the open-source drivers and could be found with their associated kallsym. We also determined that the closed-source module followed the NVOC implementation when examining the memory sample. With this information, we started to develop methods to search and parse each NVOC structure for both modules.

[4] https://github.com/NVIDIA/open-gpu-kernel-modules.

[5] https://www.volexity.com/products-overview/surge/.

[6] https://github.com/volatilityfoundation/volatility.

Table 1
Apparatus table depicting the hardware and software utilized throughout the experiment.

Hardware/Software	Use	Company	Software/Model Version
Volatility	Memory Forensics Framework	Volatility Foundation	2.6
Surge Collect Pro	Memory Acquisition Tool	Volexity	23.03.28
Ubuntu	Operating System	Canonical	22.04 LTS
NVIDIA Open-Kernel Module	GPU Driver	NVIDIA	525.125.06
NVIDIA Closed-Kernel Module	GPU Driver	NVIDIA	525
HxD	Hex Editor	mh-nexus	2.5
VSCode	Integrated Development Environment	Microsoft	1.86.0
RTX 3080ti	GPU	MSI	n/a

4.4. Method creation

We first explain our method of mapping symbols between open-to-closed source NVIDIA modules. To build on this, we explain how, once mappings have been created between each module's symbols, we can use a reverse pointer lookup method to find the addresses of NVOC structures in kernel memory. After identifying the location in memory of NVOC structures, we explain our parsing methodology. With this methodology, other researchers can build forensic tools to parse artifacts from memory regarding NVIDIA's GPUs. We build on this foundation in the Tool Creation section to create plugins for Volatility 2.6 that automate each of these methods and a standalone tool, NVSYMMAP, for automating the complete process of mapping modules.

4.4.1. Mapping open-to-closed source kernel modules symbols and objects

The first step of providing a proper memory forensics foundation for NVIDIA GPU kernel modules is providing mappings that cover open and closed-source software. We achieved this by creating links of symbols between each module. Each of the module's exported symbols can be found in **/proc/kallsyms**. One major issue with mapping symbols between modules is vital symbols are "scrubbed" in the closed-source module and can not be directly mapped by name. An example output of each module kallsyms is shown in Listing 4. We can overcome this issue by utilizing the following method.

We first compile a list of NVOC CLASSIDs from the open-source code. Next, we locate the associated open-source symbol and examine its memory contents to confirm the CLASSID. Finally, we scan each closed source symbol (related to the Nvidia kernel module) for the same CLASSID. Once we find each symbol for open and closed source modules, we then create a mapping. An example of the AccessCounterBuffer NVOC structure's class definition memory contents for both modules can be found in Listing 5, and Fig. 1 displays an overview of the result of this process.

With these mappings between open-to-closed source symbols, we can now develop forensic tools that work for both kernel modules. After mapping each symbol for NVOC_CLASS_DEF, we use a reverse lookup method, described in the Reverse NVIDIA Object Lookup section, to locate desired structures. Note many of the closed-source scrubbed symbols are not structures but functions; our methods focus only on NVOC structures and their associated members.

4.4.2. Recursive descent NVIDIA ClassID lookup

A second method was also created to map symbols for either module. With the knowledge from Source Code Analysis, we understand that all NVOC structure's first member points to a NVOC_RTTI structure, and NVOC_RTTI to NVOC_CLASS_DEF. With this we can probe each kallsym and check if the first eight bytes are a valid pointer within the context of the kernel. If so, we follow this pointer and continue checking for another pointer while keeping track of the depth. Once the first eight bytes are not a valid pointer, we check to see if a valid CLASSID is found. If so, then we check to see if the related closed-source module has the same symbol (checking for depth and CLASSID). One result of this method is the mapping between _nv022923rm (closed) and the g_pSys (open), with a depth of three, which points to the OBJSYS CLASSID. This method is shown in Fig. 2a.

4.4.3. Heuristically searching for NVIDIA ClassIDs

Finally, we created a heuristic method to search for undocumented CLASSIDs and structures for the closed-source drivers. We probed each kallsym and searched for the structure of an NVOC_CLASS_DEF. If the structure was detected, the memory was examined and verified. Interestingly, we discovered by searching that some of the CLASSIDs declared in the open-source modules that do not have associated structures in memory or the source code are found in the closed-source modules. One example of this occurring is the NVOC structure OBJGPULOG. This structure is found in the closed-source modules with the associated _nv002107rm kallsym and is initialized with a size of 496 bytes.

Listing 5. Example of NVOC Class Definition Kallsym Output

```
ClassID: 0x001f0074
Closed Source: _nv001924rm                          Open Source: __nvoc_class_def_AccessCounterBuffer

 >>> db(0xffffffffc4b31d60, 64)                     >>> db(0xffffffffc1718f40, 64)
 08 05 00 00 74 00 1f 00 3c 0e b1 c4 ff ff ff ff   08 05 00 00 74 00 1f 00 74 5c 90 c1 ff ff ff ff
 50 16 46 c2 ff ff ff ff 88 1d b3 c4 ff ff ff ff   10 8f 71 c1 ff ff ff ff a0 d4 48 c1 ff ff ff ff
 10 1c b3 c4 ff ff ff ff 08 00 00 00 00 00 00 00   70 8f 71 c1 ff ff ff ff 90 8b 71 c1 ff ff ff ff
 b0 1e b3 c4 ff ff ff ff d0 1d b3 c4 ff ff ff ff   08 00 00 00 00 00 00 00 a0 90 71 c1 ff ff ff ff
```

4.4.4. Reverse NVIDIA object lookup

While each NVOC structure does not have an exported kallsym, we can work backward from its associated NVOC_CLASS_DEF. Each structure generated by NVOC follows the same memory layout (described in the Source Code Analysis section), which can be used to locate it.

A NVOC structure's first member is a pointer to its associated NVOC_RTTI structure. Listing 6 shows an example of this. By utilizing how NVIDIA's NVOC objects are created, with each structure pointing to a RTTI structure and each RTTI pointing to an NVOC_CLASS_DEF (where each Class Definition has an associated symbol in **kallsyms**), we can locate any NVOC structure in memory that we desire.

Listing 6. Example of NVOC Structure

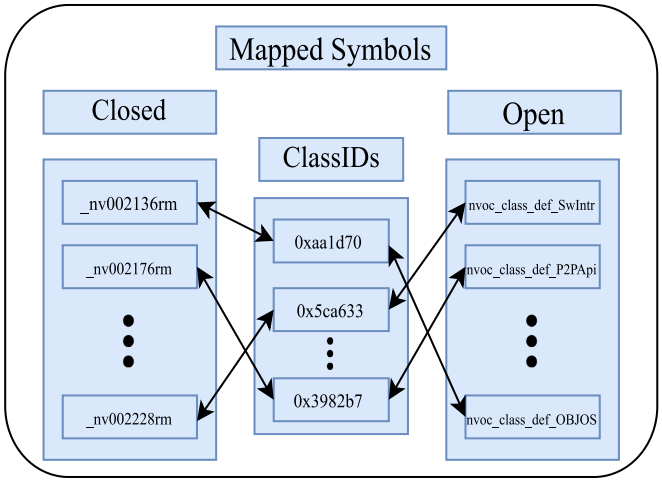


Fig. 1. Diagram of cross-mapped symbols.

```
struct GpuAccounting{
    const struct NVOC_RTTI *__nvoc_rtti;
    struct Object __nvoc_base_Object;
    struct Object *__nvoc_pbase_Object;
    GPUACCT_GPU_INSTANCE_INFO gpuInstanceInfo
    [32];
};
```

We can use a reverse pointer lookup to map RTTI structs to NVOC class definitions. After mapping each RTTI structure, we can continue using reverse pointer lookup to map NVIDIA objects to RTTI structures. This process is depicted in Fig. 2b and is automated in a volatility plugin created as described in Tool Creation.

4.4.5. NVIDIA object parsing

After identifying where these structures are in memory and their associated sizes, we needed a way of adequately extracting the data and members of the structures. NVIDIA offers an option to build their open kernel modules in debug mode by enabling the DEBUG flag – adding the gcc flag "-gsplit-dwarf" to the compilation. This flag will separate the information of the executable into two files,*.o ("OBJECT") and*.dwo ("DWARF object"). After investigating each of the files created on compilation, we identified a way of extracting a structure's memory footprint from the .dwo files with the debug information. While this method allows us to generate artificial memory structures, such as vtypes for Volatility 2, we are unable to use this method for the closed-source modules due to the absence of *.dwo files provided.

For closed-source modules, we utilize the **NVOC_CLASS_INFO** structure, which, after investigation, appears to be the same between open and closed modules to identify the size of the desired structure. After parsing the structure from memory, we make use of the open-source definition to map the closed-source structure. In most cases, this method can be used to locate the desired data; however, each structure will range in difficulty due to no direct references to how the structure's members are laid out.

Note the current standard of parsing debugging information for Volatility vtypes/symbols is using **dwarf2json**[7]; however, this tool currently does not support .dwo files; thus, we could not utilize it.

[7] https://github.com/volatilityfoundation/dwarf2json.

5. Tool creation

In this section, we will discuss the plugins and tools we created to automate the process of mapping symbols between drivers and the ability to locate desired structures in memory. After providing these foundational plugins, we extend our work into a forensic-specific plugin to parse valuable evidence from a system. We also provide a standalone tool, NVSYMAP, for automating the mapping process of each driver.

5.1. CheckNvidia

The CheckNvidia plugin runs a scan to print out if an NVIDIA kernel module was in use. If an NVIDIA module is found, the plugin will print out the information about the module. To obtain additional information about the NVIDIA module, CheckNvidia will pull from two sources of information – module_kset from the Linux kernel and pNVRM_ID from the NVIDIA module. This information is then combined and displayed to the user.

5.2. NVOC_CLASS_DEF scan

The NVOC_CLASS_DEF Scan plugin scans the kernel pages that contain modules. It looks for NVOC_CLASS_DEF structures in memory using two types of scanning. The first scanning type will utilize the "known" list of NVOC CLASSIDs. The plugin will also iteratively scan memory using the sliding window technique (scanning byte by byte). Once a word matches one of the list's entries, the plugin will validate the structure using a heuristic and add the location of the found NVOC_CLASS_DEF structure to display. The second technique utilizes a heuristic mechanism to find NVOC_CLASS_DEF structures by using the validating mechanism that method one implements. The technique will scan all NVIDIA-specific symbols in kallsyms.

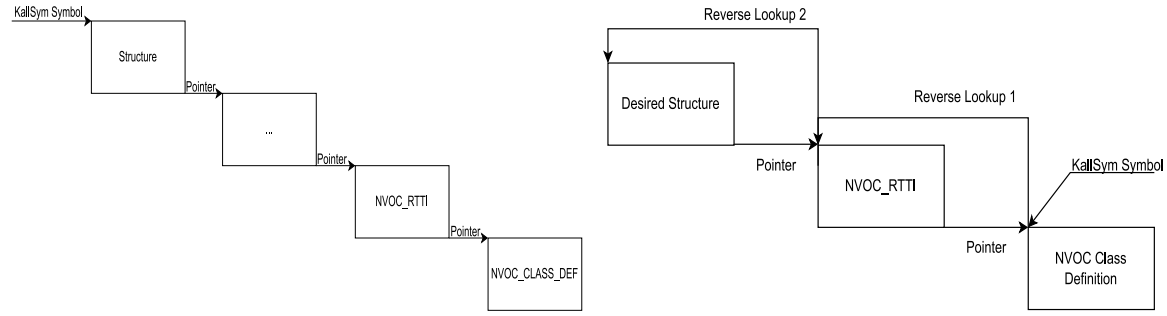
5.3. Reverse structure lookup and acquisition

The reverse structure lookup plugin will locate the NVOC Structure in memory by working backward with the Reverse Ascent Lookup method. The plugin begins searching for the symbol associated with the CLASSID provided by the user. Then, the kernel will be scanned to search for a pointer directed at the NVOC_CLASS_DEF. If the NVOC_RTTI is found, then the plugin will again scan memory, looking for a pointer directed at the NVOC_RTTI structure. Fig. 3 displays an example output of this plugin when searching of the structure associated with the OBJGPU class name with the CLASSID of **0x7ef3cb**. Note that two RTTI structures were found; this is because the RsResourcelist symbol also held a pointer to the NVOC_CLASS_DEF of OBJGPU. For acquiring the memory associated with the structure, the plugin will use the address of the structure found and the size of the structure from the NVOC_CLASS_DEF.

5.4. GPU accounting

NVIDIA provides the ability to track the usage of resources throughout the lifespan of an individual process via the GPU Accounting capability. When enabling this feature, users can manage and monitor the usage of their GPU via NVIDIA Management Library (NVLM) and nvidia-smi. The GpuAccounting structure in **/src/nvidia/generated/g_gpu_acct_nvoc.h** holds this information. The NVOC structure holds essential information for a forensic investigation, such as start time, end time, live processes, dead processes, Process identifier (PID), and much more. By parsing this structure from memory, we can account for the history of the processes run on the GPU and potentially identify malicious processes.

While this is straightforward for collecting forensic evidence, there are some limitations to this method. The first limitation of this method is there is no current way to enable GPU Accounting on the open-source



(a) Recursive Descent Lookup Example (b) Reverse Ascent Lookup Example

Fig. 2. Methods to map and extract NVOC structures.

```
$ python2 ./vol.py --profile = LinuxUbuntu-generic_x64x64 -f data.lime linux_reverse_pointer_lookup --ClassID = 0x7ef3cb
Offset (V)         Type ClassName ClassID
------------------ ---- --------- -------
0xffffffffc4b1f920 KSYM OBJGPU    0x7ef3cb
0xffffffffc4b1f9d0 RTII OBJGPU    0x7ef3cb
0xffffffffc4e534c0 RTII OBJGPU    0x7ef3cb
0xffffffffc4b1f950 OBJ  OBJGPU    0x7ef3cb
```

Fig. 3. Example of the reverse lookup plugin with the NVOC structure OBJGPU

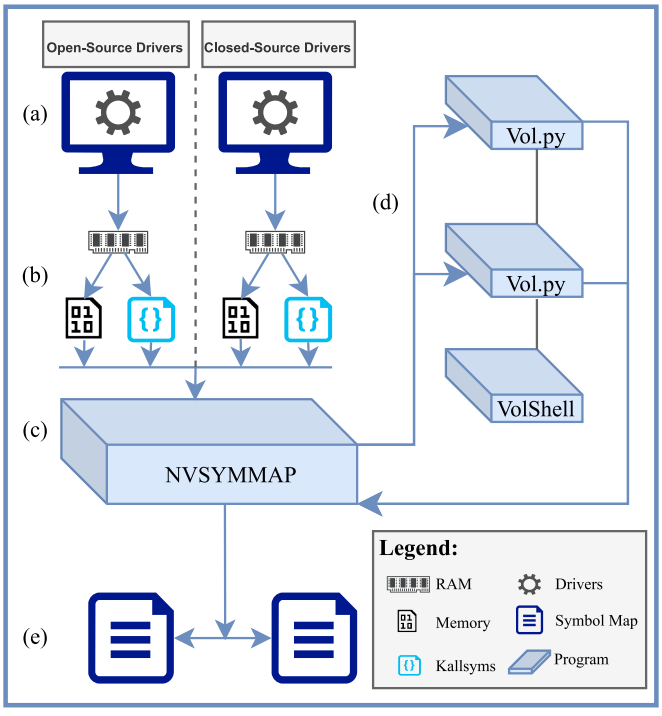


Fig. 4. Workflow of NVSYMMAP

modules. The second limitation is that GPU Accounting is not enabled by default for the closed-source modules. Users must enable GPU Accounting with NVIDIA's nvidia-smi tool via the command line using the following command:

```
sudo nvidia-smi -i \$(GPU ID) -am ENABLED
```

5.5. NVSYMMAP

NVSYMMAP,[8] NV Symbol Mapper, is an open-source command line tool written in Python3, created to automate the process of mapping symbols within and between NVIDIA kernel modules on Linux with memory forensics. NVSYMMAP was developed to map new releases of NVIDIA drivers with ease.

Fig. 4 displays the workflow of the tool for mapping open-to-closed source symbols. First, a user will create two environments with each open and closed driver they desire to map (Fig. 4a). Next, the user will acquire memory and **/proc/kallsyms** from each system (Fig. 4b). These files are then passed into NVSYMMAP with the associated Volatility2 profiles. Once NVSYMMAP has the proper information, it will create temporary files with commands (Fig. 4c) to pass into each instance of Volatility running the Linux_volshell plugin (Fig. 4d). The commands generated by NVSYMMAP will inspect each NVIDIA-related symbol and search for NVOC CLASSIDs in memory. This information is then passed back into NVSYMMAP and parsed to create mappings between each driver (Fig. 4e).

6. Evaluation

This section evaluates our methods for identifying NVOC_CLASS_DEF structures within NVIDIA kernel drivers with NVSYMMAP. We analyze the effectiveness and correctness of our tool by utilizing a manually created ground truth.

6.1. Identification of NVOC_CLASS_DEF structures

We first manually examined the open-source NVIDIA kallsyms that relate to each NVOC_CLASS_DEF. Each NVOC_CLASS_DEF has an exported kallsym starting with "_nvoc_class_def_." We created a list of these kallsyms, and manually verified the associated NVOC CLASSIDs (from the source code) by examining each symbol's memory content – resulting in 171 total CLASSIDs/structures for our ground truth. We then used NVSYMMAP to verify our manually created data with the curated _nvoc_class_def_list. After confirming our ground truth, we "blindly" searched all of the kallsym (related to NVIDIA) for the open-source

[8] https://github.com/LSUACL/GPU-Forensics/tree/main/NVSYMMAP.

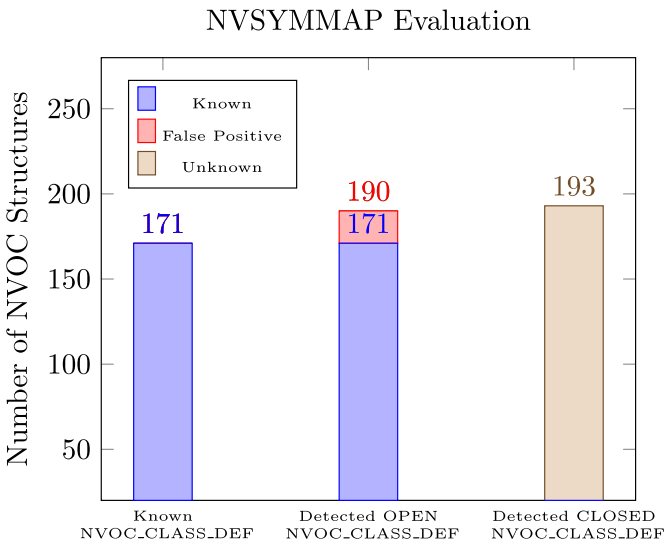


Fig. 5. Evaluation of NVSYMSMAP

modules, amounting to 14447 total symbols, to find each of the __nvoc_class_def_kallsyms. While we cannot create ground truth for the closed-source modules, we decided to run NVSYMAP with the generated list of CLASSIDs to compare the results with the open-source.

6.2. Results

Fig. 5 shows the results of our evaluation. The graph displays the total number of NVOC Class Definitions found for each test. Our ground truth is shown as "known" with a total of 171 structures. When testing the list blindly on the open kernel modules, NVSYMSMAP was able to find each of the known class definitions with 19 additional false positive symbols. Each of these 19 false positives was associated with a parent list structure in relation to the CLASSID. When running the test on the closed-source module, we detected 193 total symbols in relation to the CLASSID list.

7. Experimentation

This section describes our approach to experimenting with each open and closed-source NVIDIA driver. We aim to evaluate the drivers' differences and similarities by examining each NVOC structure. We also want to explore undocumented NVOC structures and their associated sizes and CLASSIDs. While this experiment only examined the 525 modules, our approach can be applied to newer versions of the drivers.

7.1. Approach

For our experimentation, we used NVSYMMAP. We created two new environments with each open and closed source 525 drivers and extracted the necessary information to parse each NVOC structure's class definition. In our testing, we searched for undocumented structures not found in the open-source code. We also examined the 171 known structures that were in use for the open modules and compared their sizes to the associated closed-source structures.

7.2. Findings

We found a significant amount of additional information about NVOC structures could be obtained by examining the closed-source NVIDIA drivers. Fig. 6a depicts the amount of NVOC structures utilized per version. In the open-source code, we were able to document 263 structures, and in the closed source, we identified 67 undocumented structures. The closed-source drivers utilize 330 total structures, and the open-source drivers utilize only 171 structures.

Interestingly, when examining and mapping CLASSIDs from the open-source code to the closed-source code, we recognized that the NVOC class definition structures are scrubbed alphabetically by class name (ignoring capitalization) where they iterate from _nv001924rm to _nv002253rm (AccessCounterBuffer-ZbcApi). With this knowledge, researchers can potentially infer the undocumented class names and which NVOC structures are specific to the closed-source drivers. One example of narrowing down a symbol's class name is _nv001979rm and _nv001981rm, where the CLASSIDs are GpuManagementApi and GpuResource, resulting in _nv001979rm's CLASSID name falling between GpuMa-GpuRe.

Additionally, we examined the sizes associated with the documented open-source structures versus the closed-source structures. We separated each group arbitrarily into three groups: exact (for the same size), small (for less than 100 bytes in difference), and large (for greater than 100 bytes in difference). Fig. 6b shows the results; most notable from the data is that 59 of the 171 structures tested are the exact same size in both modules. In Appendix, we display a partial listing of the obtained data, and the full results can be found on our github.

8. Related work

Most of the research on GPU forensics was completed in 2015, and little work has been compiled since then. We briefly describe the related work in GPU-Assisted Malware, GPU Forensics, and Memory Forensics.

8.1. GPU-assisted malware

GPU-Assisted malware utilizes the computational power and elevated trust of the GPU to perform specific tasks such as packing, unpacking, Direct Memory Access (DMA), and Crypto Mining. At the time of writing, there is no known "wild" GPU-assisted malware that tries to hide in VRAM to avoid AV. However, a post on a hacker forum offered a Proof of concept (POC) malware that utilized the GPU memory buffer to store malicious code to evade AV RAM scanning (Ilascu 2021). In addition to this, academic researchers created malware/rootkits to show how it could leverage hiding valuable information within the VRAM of a GPU (Reynaud 2008; Vasiliadis et al., 2015; Ladakis et al., 2013). One example of GPU-Assisted rootkit is JellyFish.[9] JellyFish was a POC academic malware that ran on Windows, Linux, and MAC in 2015 (Bongiorni 2015). Interestingly, JellyFish utilized OpenCl to interact with either NVIDIA or AMD products for "snooping" via DMA.

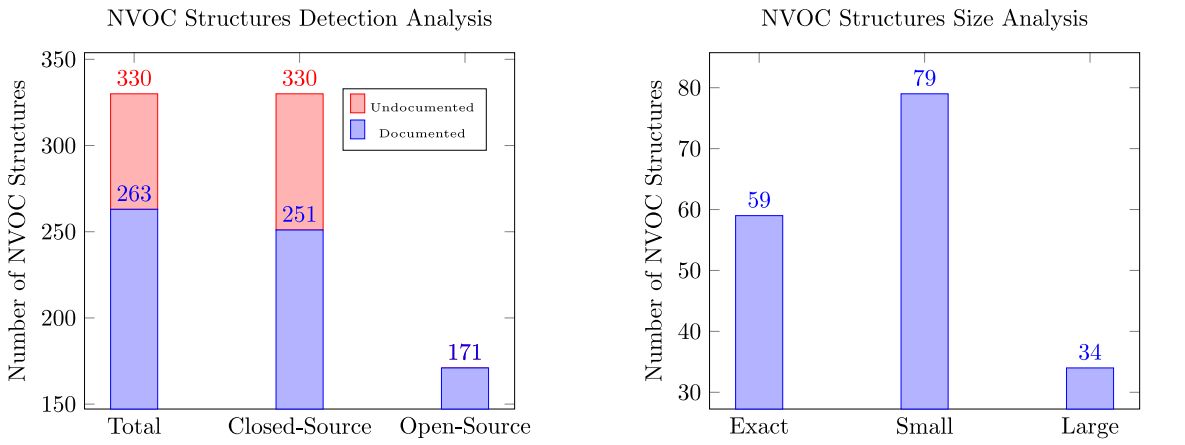
8.2. GPU forensics

GPU forensics is the process of investigating and analyzing the malicious use of the GPU. Balzarotti et al. (2015) examined the many approaches an attacker may take to misuse a GPU and its impact on memory forensics. To address these threats, a framework was suggested for analyzing GPU-executed malware by Apostol et al. (2021); however, the approach focused on high-level APIs that could be avoided by advanced attacks, whereas our approach focuses on investigating the drivers of the GPU for forensic evidence.

8.3. Memory forensics

Memory forensics is the analysis of a system's volatile memory. Case & Richard III (2017) provided a critical analysis of the current state of memory forensics and an overview of the issues that need to be addressed. We believe addressing new architectures is one major core issue and should be studied. Works involving Apple Silicon,

[9] https://github.com/nwork/jellyfish.



(a) Graph of Known and Unknown NVOC Symbols

(b) Size Analysis Graph of Known and Unknown NVOC Symbols

Fig. 6. Result of analysis of structures and sizes of NVOC.

Programmable Logic Controllers (PLC), and NVIDIA GPUs extend memory forensic's reach and address advanced attacks (Mettig et al., 2023, Awad et al., 2023).

9. Discussion and future work

Our work provides a foundation for future research involving GPU forensics. We created the first memory forensic tools for GPUs that provide forensic investigators with valuable insight into which processes accessed the GPU for NVIDIA drivers. We also presented the first analysis of NVIDIA open-source kernel modules and mapping to associate closed-source modules. Comparing our work with previous research, we contributed significant improvements to the current state of forensics involving GPUs, specifically NVIDIA products.

As described in Section 4, we created methods to accurately and reliably locate NVOC structures in memory for both open and closed-source NVIDIA kernel modules. These methods provided will help make future work possible surrounding GPU forensics.

In addition, we provide a comprehensive list of mappings for NVOC_CLASS_DEF symbols between kernel modules to extend the reach of future work and make new tool creation more accessible. With this new foundation of how NVIDIA stores information related to their GPUs on Linux-based systems, forensic investigators can start to detect and analyze malicious software that utilizes the GPU.

In future work, we aim to extend the amount of forensic evidence that can be found by an investigator. Notably, we want to investigate methods of obtaining physical VRAM images. In previous research, tools were created with OPENCL and CUDA to obtain a VRAM image; however, these tools operate from user-land, causing significant changes to RAM and potentially VRAM due to context switches required to map the memory. One patch was developed by NVIDIA for the DFRWS 2015 memory forensics challenge[10] that obtained a physical VRAM image from kernel space; however, this was specifically for the 343.13 drivers. Once we create tools for obtaining VRAM, we believe that we will be able to map the pages a process utilizes in the GPU with the NVOC structures that control address translation and memory allocation.

10. Conclusions

GPU memory forensics is possible and should be studied. Within our work, we showed that NVIDIA has opened up parts of its software that researchers can utilize to create tools and methods to extract vital forensic information. It is possible to examine both sets of modules, open and closed, and begin to understand the inner workings of how a GPU operates.

Malicious cyber attacks will continue to advance over time, so we need to keep improving our defensive tools. We need to address the threat of malware hiding information within VRAM, and we can only do that with a physical memory image of VRAM and RAM. Our approach of starting in RAM and working towards VRAM is the correct way of developing tools, and we believe that it is the solution to solving this blind spot in the forensics realm. With the methods and mappings we provided, researchers can begin to extend the view of memory forensics into the GPU environment. Our work has resulted in a new foundation for this area, and we are committed to building on it to combat the evolving landscape of cyber threats.

Acknowledgments

We want to thank our reviewers for their helpful comments. We would also like to thank Brennen Calato and Kyle McCleary for their time reviewing our paper and offering suggestions. We also want to thank Louisiana State University for funding equipment and software.

This material is based upon work supported by the National Science Foundation under Grant Number 1946626. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

[10] https://github.com/dfrws/dfrws2015-challenge.

Appendix

Class Name	Class ID	Open-Source Kallsym	Open-Size	Closed-Source Kallsym	Closed-Size	Difference
AccessCounterBuffer	0x1f0074	_nvoc_class_def_AccessCounterBuffer	1288	_nv001924rm	1288	0
BinaryApi	0xb7a47c	_nvoc_class_def_BinaryApi	1204	_nv001927rm	1072	132
BinaryApiPrivileged	0x1c0579	_nvoc_class_def_BinaryApiPrivileged	1288	_nv001928rm	1360	72
ChannelDescendant	0x43d7c4	_nvoc_class_def_ChannelDescendant	1256	_nv001932rm	1272	16
ComputeInstanceSubscription	0xd1f238	_nvoc_class_def_ComputeInstanceSubscription	1048	_nv001935rm	1048	0
ConsoleMemory	0xaac69e	_nvoc_class_def_ConsoleMemory	1312	_nv001938rm	1320	8
ContextDma	0x88441b	_nvoc_class_def_ContextDma	1256	_nv001939rm	1256	0
DebugBufferApi	0x5e7a1b	_nvoc_class_def_DebugBufferApi	1032	_nv001940rm	1032	0
DeferredApiObject	0x8ea933	_nvoc_class_def_DeferredApiObject	1632	_nv001941rm	1648	16
Device	0xe0ac20	_nvoc_class_def_Device	1608	_nv001942rm	1856	248
DiagApi	0xaa3066	_nvoc_class_def_DiagApi	1320	_nv001943rm	1352	32
DispCapabilities	0x99db3e	_nvoc_class_def_DispCapabilities	1032	_nv001944rm	1032	0
DispChannel	0xbd2ff3	_nvoc_class_def_DispChannel	1256	_nv001945rm	1256	0
DispChannelDma	0xfe3d2e	_nvoc_class_def_DispChannelDma	1576	_nv001946rm	1576	0
DispChannelPio	0x10dec3	_nvoc_class_def_DispChannelPio	1576	_nv001947rm	1576	0
DispCommon	0x41f4f2	_nvoc_class_def_DispCommon	2232	_nv001948rm	3056	824
DisplayApi	0xe9980c	_nvoc_class_def_DisplayApi	984	_nv001954rm	992	8
DisplayInstanceMemory	0x8223e2	_nvoc_class_def_DisplayInstanceMemory	200	_nv001955rm	208	8
DispObject	0x999839	_nvoc_class_def_DispObject	1504	_nv001949rm	1512	8
DispSfUser	0xba7439	_nvoc_class_def_DispSfUser	1032	_nv001951rm	1032	0
DispSwObj	0x6aa5e2	_nvoc_class_def_DispSwObj	1296	_nv001952rm	1304	8
DispSwObject	0x99ad6d	_nvoc_class_def_DispSwObject	1824	_nv001953rm	1804	20
Event	0xa4ecfc	_nvoc_class_def_Event	720	_nv001958rm	720	0
EventBuffer	0x63502b	_nvoc_class_def_EventBuffer	1000	_nv001959rm	1000	0
Fabric	0x0ac791	_nvoc_class_def_Fabric	144	_nv001963rm	136	8
FABRIC_VASPACE	0x8c8f3d	_nvoc_class_def_FABRIC_VASPACE	696	_nv001961rm	696	0
FlaMemory	0xe61ee1	_nvoc_class_def_FlaMemory	1336	_nv001968rm	1344	8
FmSessionApi	0xdfbd08	_nvoc_class_def_FmSessionApi	904	_nv001969rm	904	0
GenericEngineApi	0x4bc329	_nvoc_class_def_GenericEngineApi	1040	_nv001974rm	1416	376
GenericKernelFalcon	0xabcf08	_nvoc_class_def_GenericKernelFalcon	312	_nv001975rm	400	88
GpuAccounting	0x0f1350	_nvoc_class_def_GpuAccounting	127560	_nv001977rm	93768	33792
GpuDb	0xcdd250	_nvoc_class_def_GpuDb	128	_nv001978rm	120	8
GPUInstanceSubscription	0x91fde7	_nvoc_class_def_GPUInstanceSubscription	1104	_nv001972rm	1104	0
GpuManagementApi	0x376305	_nvoc_class_def_GpuManagementApi	704	_nv001979rm	704	0
GpuResource	0x5d5d9f	_nvoc_class_def_GpuResource	768	_nv001981rm	768	0
GpuUserSharedData	0x5e7d1f	_nvoc_class_def_GpuUserSharedData	1024	_nv001982rm	1024	0
GSyncApi	0x214628	_nvoc_class_def_GSyncApi	1208	_nv001973rm	1208	0
Hdacodec	0xf59a20	_nvoc_class_def_Hdacodec	1024	_nv001991rm	1040	16
Heap	0x556e9a	_nvoc_class_def_Heap	1560	_nv001992rm	1560	0
I2cApi	0xceb8f6	_nvoc_class_def_I2cApi	1064	_nv001998rm	1064	0
INotifier	0xf8f965	_nvoc_class_def_Inotifier	56	_nv001999rm	56	0
Intr	0xc06e44	_nvoc_class_def_Intr	5344	_nv002006rm	6160	816
IntrService	0x2271cc	_nvoc_class_def_IntrService	48	_nv002007rm	48	0
IoAperture	0x40549c	_nvoc_class_def_IoAperture	264	_nv002008rm	264	0
KernelBif	0xdbe523	_nvoc_class_def_KernelBif	816	_nv002010rm	752	64
KernelBus	0xd2ac57	_nvoc_class_def_KernelBus	30064	_nv002011rm	28832	1232
KernelCcu	0x5d5b68	_nvoc_class_def_KernelCcu	824	_nv002013rm	720	104
KernelCcuApi	0x3abed3	_nvoc_class_def_KernelCcuApi	1056	_nv002014rm	1056	0
KernelCE	0x242aca	_nvoc_class_def_KernelCE	1080	_nv002012rm	1056	24
KernelCeContext	0x2d0ee9	_nvoc_class_def_KernelCeContext	1592	_nv002015rm	1608	16
KernelChannel	0x5d8d70	_nvoc_class_def_KernelChannel	2056	_nv002016rm	2144	88
KernelChannelGroup	0xec6de1	_nvoc_class_def_KernelChannelGroup	456	_nv002017rm	504	48
KernelChannelGroupApi	0x2b5b80	_nvoc_class_def_KernelChannelGroupApi	1192	_nv002018rm	1192	0
KernelCtxShare	0x5ae2fe	_nvoc_class_def_KernelCtxShare	184	_nv002019rm	192	8
KernelCtxShareApi	0x1f9af1	_nvoc_class_def_KernelCtxShareApi	1064	_nv002020rm	1064	0
KernelDisplay	0x55952e	_nvoc_class_def_KernelDisplay	912	_nv002021rm	848	64
KernelFalcon	0xb6b1af	_nvoc_class_def_KernelFalcon	136	_nv002022rm	224	88
KernelFifo	0xf3e155	_nvoc_class_def_KernelFifo	1552	_nv002023rm	1664	112
KernelFsp	0x87fb96	_nvoc_class_def_KernelFsp	880	_nv002024rm	776	104
KernelGmmu	0x29362f	_nvoc_class_def_KernelGmmu	24544	_nv002025rm	24624	80
KernelGraphics	0xea3fa9	_nvoc_class_def_KernelGraphics	1592	_nv002026rm	1544	48
KernelGraphicsContext	0x7ead09	_nvoc_class_def_KernelGraphicsContext	1064	_nv002027rm	1144	80
KernelGraphicsContextShared	0xe7abeb	_nvoc_class_def_KernelGraphicsContextShared	1600	_nv002028rm	160	1440
KernelGraphicsManager	0xd22179	_nvoc_class_def_KernelGraphicsManager	1216	_nv002029rm	1112	104
KernelGraphicsObject	0x097648	_nvoc_class_def_KernelGraphicsObject	1656	_nv002030rm	1704	48
KernelGsp	0x311d4e	_nvoc_class_def_KernelGsp	79048	_nv002031rm	79144	96
KernelHead	0x0145e6	_nvoc_class_def_KernelHead	152	_nv002032rm	192	40
KernelHostVgpuDeviceApi	0xb12d7d	_nvoc_class_def_KernelHostVgpuDeviceApi	1328	_nv002033rm	1328	0
KernelIoctrl	0x880c7d	_nvoc_class_def_KernelIoctrl	632	_nv002035rm	528	104

References

Apostol, T.-P., Velea, R., Deaconescu, R., 2021. A framework for analyzing gpu-executed malware. In: 2021 23rd International Conference on Control Systems and Computer Science (CSCS). ', IEEE, pp. 165–171.

Awad, R.A., Rais, M.H., Rogers, M., Ahmed, I., Paquit, V., 2023. Towards generic memory forensic framework for programmable logic controllers. Forensic Sci. Int.: Digit. Invest. 44, 301513.

Balzarotti, D., Di Pietro, R., Villani, A., 2015. The impact of gpu-assisted malware on memory forensics: a case study. Digit. Invest. 14, S16–S24.

Bongiorni, L., 2015. Lucabongiorni/jellyfish: gpu rootkit poc by team jellyfish. https://github.com/LucaBongiorni/jellyfish.

Case, A., Richard III, G.G., 2017. Memory forensics: the path forward. Digit. Invest. 20, 23–33.

Cherukuri, R., Baskaran, S., Ritger, A., Oh, F., Swoboda, D., 2023. Nvidia releases open-source gpu kernel modules. https://developer.nvidia.com/blog/nvidia-releases-open-source-gpu-kernel-modules/.

Ilascu, I., 2021. Cybercriminal sells tool to hide malware in amd, nvidia gpus. https://www.bleepingcomputer.com/news/security/cybercriminal-sells-tool-to-hide-malware-in-amd-nvidia-gpus/.

Ladakis, E., Koromilas, L., Vasiliadis, G., Polychronakis, M., Ioannidis, S., 2013. You can type, but you can't hide: a stealthy gpu-based keylogger. In: Proceedings of the 6th European Workshop on System Security (EuroSec)'. Citeseer.

Mettig, R., Glass, C., Case, A., Richard III, G.G., 2023. Assessing the threat of rosetta 2 on apple silicon devices. Forensic Sci. Int.: Digit. Invest. 46, 301618.

Peddie, J., 2023. The graphics add-in board market continued its correction in q1 2023. https://www.jonpeddie.com/news/the-graphics-add-in-board-market-continued-its-correction-in-q1-2023/.

Reuters, 2023. Factbox: how megacap stocks fared after hitting $1 trillion in market cap. https://www.reuters.com/markets/us/how-megacap-stocks-fared-after-hitting-1-trillion-market-cap-2023-05-30/.

Reynaud, D., 2008. Gpu Powered Malware.

Tijanic, M., 2022. Confusion about some file name formats · issue #100 · nvidia/open-gpu-kernel-modules. https://github.com/NVIDIA/open-gpu-kernel-modules/issues/100.

Vasiliadis, G., Polychronakis, M., Ioannidis, S., 2015. Gpu-assisted malware. Int. J. Inf. Secur. 14, 289–297.