Trend
Research

A Survey of Cloud-Based GPU Threats and Their Impact on AI, HPC, and Cloud Computing

Numaan Huq, Philippe Lin, Roel Reyes, Charles Perine

Contents

Published by
Trend Research

Written by
Numaan Huq, Philippe Lin,
Roel Reyes, Charles Perine

Graphics processing units (GPUs) are the hardware engines driving the AI revolution. Large language model (LLM)-powered generative AI (GenAI) became mainstream with the public release of OpenAI's ChatGPT. AI usage has given rise to innovative AI-powered applications for businesses, productivity, image generation, video generation, data analysis, and social media, among others. Powering AI applications are GPUs, which are specialized microchips designed to accelerate computer graphics and image processing. GPUs are also useful for non-graphics tasks, especially parallel processing problems. GPUs are designed for parallel processing and can run thousands of simple compute tasks simultaneously, as opposed to a central processing unit (CPU), which is designed for a few complex tasks at a time. NVIDIA introduced a C-like programming language called Compute Unified Device Architecture (CUDA) that opened up GPU programming to developers. Similar to CUDA, AMD GPUs support a programming language called OpenCL, which aims to be a vendor-independent language for multiple platforms.

ChatGPT is an LLM, which is a class of AI that helps generate human-like responses. LLMs are competent in many applications such as language translation, content generation, sentiment analysis, data analysis, chatbots, and more. LLMs are complex neural networks with vast numbers of interconnected nodes performing repeated calculations and adjustments. GPUs excel at the massive parallel computations that are fundamental to neural network training and inference. Training complex LLMs could take tens of years on CPUs, whereas GPUs reduce that period to a more manageable duration of months. Specialized GPUs like NVIDIA's, which come equipped with Tensor Cores, are designed for matrix operations, which are extensively used in deep learning and LLMs. GPUs also have on-chip high-speed device memory that is crucial for processing the enormous training datasets.

Similar to AI applications, GPUs are extensively used in high-performance computing (HPC). HPC tackles problems such as analyzing massive datasets and running complex simulations. These tasks have a high degree of parallelism, which makes them well-suited for GPUs. Simulations such as weather prediction, drug modeling, fluid dynamics, and protein folding require an immense number of calculations at each step, and GPUs accelerate this process. This allows researchers to experiment rapidly, test hypotheses, and gain valuable insights quicker. GPUs are often more energy-efficient than CPUs for HPC tasks, allowing more computation within the same power envelope. Many HPC applications integrate machine learning and deep learning, as combining these techniques leads to new scientific insights and accelerated discovery. We already discussed how GPUs are very good at processing artificial intelligence and machine learning (AI/ML) tasks, so using GPUs to solve tasks that involve both HPC and AI/ML is a natural choice.

As AI training and processing as well as HPC applications become more important for businesses, they are switching to cloud-based GPUs versus an on-site setup. Cloud-based GPUs provide scalability and flexibility, which is great for bursty workloads, especially when there is a spike in the need for massive compute power followed by low usage periods. There is no upfront investment in expensive hardware and maintenance, as cloud services operate on a pay-as-you-go model. With cloud-based GPUs, users get access to the latest GPU chips available in the market. This is critical in fields like AI, where hardware advances greatly accelerate processing tasks. Another advantage is global access to shared GPU resources without worrying about hardware logistics.

Given the increasing reliance on GPUs for everyday business tasks, this paper explores the security threats GPUs face and what actions can be taken to mitigate the risks. As the reliance on cloud-based GPU instances grows, so does the importance of ensuring their security. The threats are multifaceted, ranging from data breaches and unauthorized access to more sophisticated attacks like reading GPU memory. By examining these security challenges, this research aims to provide insights into the current threat landscape for cloud-based GPUs. It will discuss both the vulnerabilities inherent in these systems and strategies for protecting them against cyberattacks.

Impacts of GPU Attacks

GPU attacks signal a shift in the cybersecurity landscape for AI/ML, LLMs, and HPC, all of which heavily rely on GPUs for their processing power. Cloud-based GPU systems, the backbone of these technologies, are becoming a prime target for sophisticated attacks that are specially crafted to exploit the unique architectural traits of GPUs, threatening the integrity, confidentiality, and availability of these systems. In this section, we explore the following impacts of GPU attacks:

- **Increased risk of data leakage and intellectual property theft.** Sophisticated attacks on cloud-based GPUs increases the risk of data leakage, including exfiltration of sensitive data processed by AI and ML models. Such attacks can lead to the theft of intellectual property, including proprietary algorithms and datasets, undermining the competitive advantage and confidentiality of businesses and research institutions.
- **Erosion of model integrity and trust.** Attacks that modify the behavior of AI and ML models or compromise the integrity of computations performed by HPC systems can have far-reaching consequences. Such modifications can result in flawed decision-making, incorrect data analysis, or biased outcomes, eroding trust in these models.
- **Threats in multitenant cloud environments.** Given the shared nature of cloud-based GPU resources, attacks exploiting vulnerabilities in the hypervisor can lead to cross-tenant data access or leakage. This amplifies the impact of attacks, as a single breach could potentially compromise the data and systems of multiple users.
- **Exploitation of parallel processing architectures.** The parallel processing capabilities of GPUs, while great for performance, also offer attackers a means to execute more sophisticated and hard-to-detect cyberattacks. This includes parallel execution of malicious payloads, GPU-based code obfuscation, or leveraging GPU resources for tasks such as cryptojacking or complex data analyses that support further attacks.
- **Risks of cryptominers and GPU malware.** Cryptominers and other forms of GPU malware present significant risks, as attackers could exploit cloud-based GPU resources for unauthorized cryptocurrency mining, exhausting computational resources and potentially causing financial losses and system instability. Cryptomining is by far the most common motive for attackers attempting to hijack cloud-based GPU resources.
- **Vulnerability to denial-of-service (DoS) attacks.** Cloud-based GPU servers are susceptible to DoS attacks aimed at overwhelming GPU resources, leading to service degradation or complete service unavailability. Such attacks could disrupt critical AI-powered services or HPC tasks, resulting in operational and financial impacts.
- **Impact on the development of AI and HPC applications.** Security concerns related to cloud-based GPU attacks could slow the adoption, development, and deployment of AI and HPC applications. Organizations need to invest in additional security measures, conduct more rigorous testing, or even reconsider the use of cloud-based GPUs for mission critical applications.

As AI, ML, LLM, and HPC applications continue to develop and proliferate, securing the underlying GPU infrastructure against sophisticated attacks will be paramount to protecting the integrity, confidentiality, and availability of these technologies. Until now, we have talked extensively about the impacts of GPU attacks, especially on cloud-based GPUs. However, challenges in detecting and mitigating GPU-based attacks arise as traditional cybersecurity solutions might not be equipped to monitor and protect GPU resources. The detection of attacks that target or leverage GPUs will require specialized monitoring tools and techniques that can deeply inspect GPU resources. Additionally, organizations relying on cloud-based GPUs for processing sensitive data must consider the regulatory and compliance implications of potential security breaches. This includes obligations under data protection laws and industry-specific regulations, which might necessitate additional security controls and measures. For a deeper understanding of common threats in this area, Trend Micro has published some excellent research on cloud security[1] and cloud computing[2] that we recommend as a valuable resource that complements our discussion in this paper by outlining the full gamut of cloud security vulnerabilities.

GPU Attack Risk Assessment Matrix

We explored 10 different types of GPU attacks (please refer to Appendix 2 for a technical discussion of these GPU attacks), including those that target the GPU itself, as well as attacks that use the GPU's processing power. Understanding the impact of these attacks is crucial for the security of GPU-powered systems across cloud, AI, and HPC domains. We created a risk matrix that assesses the likelihood and potential impacts of GPU attacks that were discussed previously. The risk matrix looks at the general impact as well as impacts specific to cloud-based GPUs, AI, and HPC applications, as those deployments are exponentially growing in size and stakeholders will need to understand the risks and allocate resources effectively.

Threat Type	Risk Level	Likelihood	General Impact	Cloud Impact	AI Impact	HPC Impact
GPU Side-Channel Attacks	**High**	**Medium** Attacks are possible, but not trivial to execute.	**High** Potential for significant data leakage and security breaches.	**High** Potentially exposes data across users in shared environments.	**High** Risk of leaking sensitive inference data or insights into model internals.	**High** Could lead to the disclosure of sensitive computation or simulation results.
GPU Rootkits	**Medium**	**Low** Sophisticated attacks, less frequent in well-monitored environments.	**High** Can have far-reaching effects through system compromise.	**High** Can evade detection and persistently compromise cloud services.	**High** Threatens the integrity of AI models and the confidentiality of proprietary information.	**Medium** Possible disruption to HPC tasks; the impact varies by the specific use case.
API Abuse and Kernel Manipulation	**High**	**Medium** Vulnerabilities can exist, and attackers might leverage them.	**High** Potential for severe system compromise and data manipulation.	**Medium** Potential for exploiting vulnerabilities, mitigated by cloud platform securities.	**High** Direct manipulation could compromise AI models and data.	**High** Directly affects the integrity and execution of computational tasks.
Denial-of-Service Attacks	**High**	**High** These attacks are common and can be easily launched.	**High** Disrupt service availability, potentially causing significant losses.	**High** Directly impact service availability, affecting multiple users.	**High** Can render AI services inoperative, critically affecting availability.	**High** Severely restrict access to computational resources, disrupting operations.
GPU Malware for Cryptomining	**Medium**	**High** Malware is prevalent and targets any accessible resources.	**Medium** Mainly impacts system performance and costs.	**High** Consumes computational resources, leading to increased costs and degraded performance.	**Low** Mainly a resource drain; indirect impact unless AI tasks are severely resource constrained.	**Low** Similar to AI, mainly a resource drain with limited direct impact.

Threat Type	Risk Level	Likelihood	General Impact	Cloud Impact	AI Impact	HPC Impact
Exploiting Vulnerabilities in GPU Drivers	**High**	**Medium** Vulnerabilities exist, but patching and mitigations are common.	**High** Compromise can have severe consequences for system and data integrity.	**Medium** Cloud platforms might mitigate some risks, but vulnerabilities can lead to system compromise.	**High** Potentially compromises the integrity and confidentiality of AI processes.	**High** Unauthorized access or disruption of tasks is a significant threat.
GPU Assisted Code Obfuscation	**Medium**	**Low** Requires specialized techniques, not as common as basic malware.	**Medium** Can hinder security analysis and delay incident response.	**Medium** Complicates malware detection within the cloud infrastructure.	**Medium** Can obscure malicious activities affecting AI model integrity.	**Medium** Could hide the presence of unauthorized computations or data manipulations.
Overdrive Fault Attacks	**Medium**	**Low** Requires physical access or specialized manipulation techniques.	**Medium** Can impact accuracy and reliability, more targeted in nature.	**Low** Rare in controlled cloud environments but could occur through hardware manipulation.	**Medium** Specific attacks might subtly alter the outcomes of AI models.	**High** Precision tasks might be compromised, affecting critical results.
Memory Snooping/ Cross-Virtual Machine (VM) Attacks in vGPU Environments	**High**	**Medium** Attacks are possible on virtualized GPUs, especially if not properly configured.	**High** Potential for major data breaches and loss of confidentiality.	**High** Breaks isolation between users, undermining cloud security.	**High** Unauthorized access to AI datasets and models poses a serious confidentiality risk.	**High** Data leakage is a major concern, especially in shared computational environments.
Compromised AI Models/Trojaning	**High**	**Medium** Attacks rely on model distribution channels and user trust.	**High** Can lead to incorrect or malicious outputs with significant consequences.	**Medium** Cloud infrastructure might not be directly impacted but facilitates model distribution.	**High** Directly affects model integrity, leading to incorrect or malicious decisions.	**Medium** Indirect impact initially, but a growing concern during model deployment impacting HPC.

Threat Mitigation Strategies

To safeguard cloud-based GPU environments against cyberattacks, it's imperative to deploy a robust set of protection measures. These defensive measures span from hardware enhancements to software-level precautions, aiming to secure the AI, ML, LLM, and HPC applications that use cloud-based GPUs as their processing backbone. The following is by no means an exhaustive list of threat mitigation strategies; instead, it is an exploration of macro-level strategies that IT security practitioners can choose to implement in a layered defense framework.

- **Advanced virtualization security.** Use hypervisor-level security enhancements to safeguard environments against cross-VM attacks in virtualized GPU (vGPU) environments. This includes stricter isolation policies, memory encryption, and sophisticated access controls to prevent unauthorized access to shared GPU resources.
- **Robust kernel isolation.** Implement strong isolation measures for GPU kernels to protect these against API abuse and kernel manipulation attacks. This could involve using containerization technologies with enhanced security features or adopting vGPUs that provide better isolation between computing tasks.
- **Enhanced memory management.** Use advanced memory management techniques to prevent memory snooping and leakage vulnerabilities. Techniques such as memory randomization, encryption, and timely clearing of GPU memory after task completion can help mitigate the risk of sensitive data exposure.
- **Secure code execution frameworks.** Use secure execution frameworks for running GPU-accelerated code, ensuring that the code is verified and authenticated before execution. This helps protect frameworks against malicious code execution and trojaned AI models.
- **Driver and firmware security.** Maintain up-to-date GPU drivers and firmware with the latest security patches. Rigorous updates and a patch management process are critical to protect environments against exploits targeting vulnerabilities in GPU drivers and firmware.
- **GPU usage monitoring and anomaly detection.** Deploy monitoring tools that can detect anomalous GPU usage patterns that are indicative of attacks such as cryptojacking, DoS, or excessive resource consumption. Integrating AI/ML techniques can improve the detection of sophisticated attacks.
- **Application-level security measures.** Apply application-level security best practices, including secure coding techniques, to mitigate risks associated with GPU-accelerated applications. This includes validating input data to prevent injection attacks and ensuring that AI/ML models are robust against poisoning and evasion techniques.
- **Hardware Security Modules (HSMs) for sensitive operations.** For critical cryptographic operations or sensitive data processing, use dedicated HSMs[3] instead of general-purpose GPUs. HSMs offer higher security guarantees and are designed to resist tampering and leakage.
- **Access control policies.** Enforce strict access control policies for GPU resources, limiting access to authorized users and applications only. Implement role-based access control (RBAC)[4] and audit trails to monitor access to GPU resources and detect unauthorized attempts.

- **Education and awareness.** Raise awareness about potential security risks associated with cloud-based GPU usage. Providing training on secure development practices for GPU-accelerated applications and how to recognize signs of GPU-related attacks can be an effective preventive measure.

Securing cloud-based GPUs against sophisticated cyberattacks requires a multifaceted, multilayered approach that encompasses hardware, software, and procedural safeguards. By implementing these measures, organizations can enhance the security of their cloud-based GPU servers, protecting the integrity, confidentiality, and availability of their AI, ML, LLM, and HPC applications.

Conclusion

The fast-paced evolution of GPU attacks poses a direct and complex threat to the cybersecurity of AI, ML, LLM, and HPC applications, all of which rely on GPU acceleration. Attackers are crafting sophisticated exploits that target the unique architecture of GPUs and the shared infrastructure of cloud computing environments. These attacks can jeopardize the integrity, confidentiality, and availability of GPU resources and sensitive data. During our research, we identified the following top security concerns specific to cloud-based GPUs:

- **Cloud-based GPUs are powerful but vulnerable.** Attackers can exploit various components of a cloud-based GPU system, targeting memory management, processing units, and special function units.
- **Side-channel attacks pose a stealthy threat.** These attacks can extract sensitive information by exploiting physical properties of GPUs, particularly the cache organization in multi-GPU systems.
- **GPU rootkits can be difficult to detect.** Pieces of malicious software like Jellyfish can establish a persistent presence on GPUs, manipulating operations and evading traditional security tools.
- **API abuse and GPU kernel manipulation are potential risks.** Weaknesses in GPU programming frameworks like CUDA or OpenCL can be leveraged for unauthorized access or disruption of legitimate processes.
- **Cryptomining and distributed denial-of-service (DDoS) attacks are expensive and disruptive.** Cloud-based GPUs offer immense massive power, making them targets for resource-draining cryptojacking and large-scale DDoS attacks that disrupt services and cause financial losses.

Defense against GPU threats needs a proactive strategy that integrates cybersecurity technologies with well-defined operational best practices. As AI and HPC deployments in the cloud grow, GPU security becomes paramount. Addressing these security challenges requires a collaborative approach: Cloud service providers must implement robust measures, including GPU-specific intrusion detection and anomaly monitoring, while developers need to prioritize secure coding practices and understand potential attack vectors when using cloud-based GPUs. Regular security audits and penetration testing are also crucial for proactively identifying and patching vulnerabilities. In conclusion, the transformative potential of cloud-based GPUs is clear in the age of AI and HPC applications. By being vigilant toward GPU security threats (a collection of which we explored in this paper) and implementing proactive countermeasures, we can protect both the critical infrastructure and sensitive data.

Appendix

Appendix 1: GPU Technical Primer

Graphics processing units (GPUs) are specialized chips designed to accelerate the rendering of images, videos, and animations. They have a parallel processing architecture and an on-chip memory that make them efficient at manipulating and transforming data to speed up image creation. This is in contrast to central processing units (CPUs), which are general-purpose processors designed to perform a wide range of computational tasks and are optimized for sequential processing. CPUs and GPUs have completely different architectures. CPUs consist of a few cores optimized for sequential processing, making them well-suited for complex tasks and general computing. In contrast, GPUs contain thousands of smaller cores designed for parallel processing. This parallel processing architecture allows GPUs to perform the enormous volumes of calculations required for graphics rendering and other data-intensive tasks faster and more efficiently than CPUs.

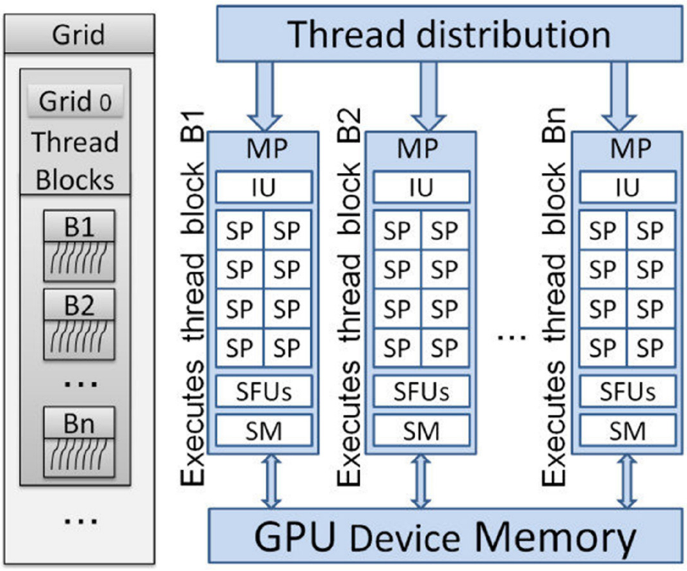


Figure 1. Simplified GPU architecture[5]

The simplified GPU architecture diagram presented in Figure 1 shows the key GPU components and their hierarchical organization. This includes a grid, thread blocks, streaming multiprocessors (SMs), cores, and GPU device memory.

Grid and Thread Blocks

At the highest level, the GPU organizes computation into a grid that consists of multiple thread blocks (labeled B1, B2...Bn). These blocks are batches of threads, the smallest execution unit that can be executed in parallel. The grid is the overall structure that defines the entire space over which a kernel (a GPU function) operates.

Streaming Multiprocessors

Each thread block is associated with a multiprocessor (MP), also commonly referred to as a streaming multiprocessor (SM), which is responsible for executing all the threads within that block. An SM is a processing unit within the GPU containing multiple cores and specialized functional units. It is designed to handle multiple threads concurrently, allowing the GPU to process operations in parallel. SMs manage groups of threads, known as warps on NVIDIA GPUs and wavefronts on AMD GPUs, which are scheduled and executed together to optimize processing efficiency.

Cores and Functional Units

Within each SM, there are several types of cores and functional units, including the following:

- **Integer Unit (IU).** This handles integer operations, fundamental to various computational tasks, especially those that do not require floating-point calculations.
- **Scalar Processors (SPs).** These are the actual cores that perform arithmetic computations. The high number of SPs in an SM allow the GPU to execute large numbers of floating-point operations simultaneously.
- **Special Function Units (SFUs).** These units handle special mathematical functions such as sine, cosine, reciprocal, and square root operations. SFUs perform these complex operations more efficiently than general-purpose scalar processors.

Single Instruction, Multiple Data Architecture

GPUs use single instruction, multiple data (SIMD) execution for efficient processing. In SIMD, a single instruction is executed across multiple data points simultaneously. This is key to GPU efficiency; by executing the same operation on different data concurrently, GPUs significantly accelerate data-parallel tasks. This approach is beneficial for graphics rendering, scientific computations, and applications in artificial intelligence and machine learning (AI/ML), where the same operations are applied to large datasets.

GPU Device Memory

The GPU device memory is high-speed memory accessible to all SMs. It is used to store data that is processed by the kernels running on the GPU. This memory, along with registers, shared memory within the SM, and various levels of cache, plays a vital role in performance by storing and quickly accessing the data required for computation.

Execution Flow

The execution flow begins with the distribution of thread blocks across the available SMs. Each SM executes its assigned block of threads, with individual threads handled by the SPs and SFUs under the control of the IU. The parallel execution of these threads across all SMs results in high computational throughput.

Summary

This simplified GPU architecture explains the parallel processing capabilities of GPUs. With their optimized architecture, GPUs excel at tasks that can be reduced into many smaller parallel operations. The use of diverse device memory types, efficient warp/thread scheduling mechanisms, and SIMD execution enhances the GPU's parallel processing prowess. While GPUs offer significant advantages in parallel processing, they also face challenges such as power consumption, heat generation, and the complexity of programming for parallel architectures.

Appendix 2: GPU Attacks

GPUs, with their parallel processing architecture consisting of thousands of cores, are designed to handle massive volumes of simultaneous compute operations. This architecture, while boosting computational efficiency in applications like graphics rendering, AI/ML, and, among others, HPC, also introduces unique vulnerabilities. Attacks exploiting these leverage different elements of the GPU, such as its memory management (global, shared, local, and texture memory), SMs, IUs and SPs, and SFUs.

In this section, we explore 10 different types of GPU attacks, including those that target the GPU itself as well as attacks that use the GPU's processing power. This is by no means a comprehensive list of all possible GPU attacks, but rather a survey of the different types of attacks possible with or against GPUs.

1. GPU Side-Channel Attacks

GPU side-channel attacks exploit the physical properties of GPUs to collect sensitive information about computer systems and/or running applications without directly accessing them. These attacks target unique characteristics of the GPU architecture, such as the cache hierarchy and parallel processing capabilities, which can be used to infer information about the data being processed by the system. GPU side-channel attacks are a threat to data security because they can compromise sensitive information such as cryptographic keys or other confidential data without being detected by traditional security measures. Attackers can use these attacks to gain unauthorized access to system resources, steal sensitive data, or disrupt processes. In this section, we present an overview of how side-channel attacks are executed, focusing on multi-GPU systems and data compression vulnerabilities.

Cache Organization in Multi-GPU Systems

High-performance computing, especially deep learning, necessitates the use of multi-GPU systems. These systems provide high throughput and interconnect bandwidth, which is essential for maximizing the performance of neural network training. For example, NVIDIA's DGX-1 system uses a hybrid cube-mesh network topology with NVLink interconnects.[6] Modern GPUs like NVIDIA's Tesla P100 feature a two-level data cache (L1 and L2)[7] with data loading primitives that allow bypassing the L1 cache.[8] By leveraging these primitives, attackers can bypass the L1 cache and directly access the L2 cache,[9] leading to increased vulnerability. In multi-GPU systems, attackers can exploit the cache organization to execute side-channel attacks by taking advantage of differences in cache behaviour between local and remote GPUs. Attackers can use these to distinguish between cache hits and misses, and measure access times for different cache levels.

Attack Execution

1. **Profiling cache hierarchy.**[10, 11] The first step involves profiling the cache hierarchy's timing characteristics. Attackers perform tests to determine the timing differences between cache hits and misses for both local and remote GPU accesses. This involves allocating buffers in GPU memory, and by measuring access times for different cache levels and stride sizes, attackers can determine which buffers are cached in L1 and L2 caches, respectively. Modern GPUs also have data-loading primitives that enable fetching data directly from memory without requiring a cache miss.[12] These primitives can be used by the attackers to bypass the cache hierarchy and access data directly.
2. **Exploiting cache timing for information leakage.**[13, 14] With cache timing mapped out, attackers can launch timing attacks. By strategically accessing buffers to induce contention between remote and local GPU caches, they can measure timing discrepancies. These discrepancies reveal information about data being processed by other applications on the same physical machine, or even applications running in a separate virtual machine (VM). By measuring the time it takes to access their own data, the attacker can infer whether their data was evicted from cache due to the victim's activity. Timing variations gives the

attacker clues about what the victim process might be doing, data it might be accessing, and specific code paths being executed, among others. In some scenarios, attackers can use these timing variations to break encryption, inferring secret keys[15, 16] based on the victim's cache usage patterns during cryptographic operations.

3. Utilizing data compression vulnerabilities.[17, 18] All modern GPUs employ data compression for optimization. This vendor-specific compression can be exploited to leak visual data from GPUs. Attackers manipulate pixel data through web browsers, applying specialized scalable vector graphics (SVG) filters to make certain textures highly compressible or uncompressible, and then measure how long it takes the GPU to render these manipulated images. Since compression takes time, the rendering time indirectly reveals whether the GPU found the data easy to compress, and this leaks information about the original pixel's content. A well-known example of this type of attack is the GPU.zip attack[19, 20]. This type of compression vulnerability attack demonstrates a web-based method to attack GPUs, and highlights the risk that even seemingly harmless optimization features can give rise to unexpected vulnerabilities.

2. GPU Rootkits

GPU rootkits[21] like Jellyfish[22, 23] are a sophisticated class of malware that leverages the GPU for malicious purposes. Jellyfish operates by deploying malware that intercepts system calls made to the GPU, allowing an attacker to execute unauthorized commands and manipulate GPU operations stealthily. Jellyfish does not provide direct access to the GPU itself, rather it uses techniques such as LD_PRELOAD and OpenCL APIs to intercept and manipulate system calls made by applications to the GPU.

A technical breakdown of how such an attack would work, based on the concepts introduced by the Jellyfish rootkit and general principles of how malware works, is as follows:

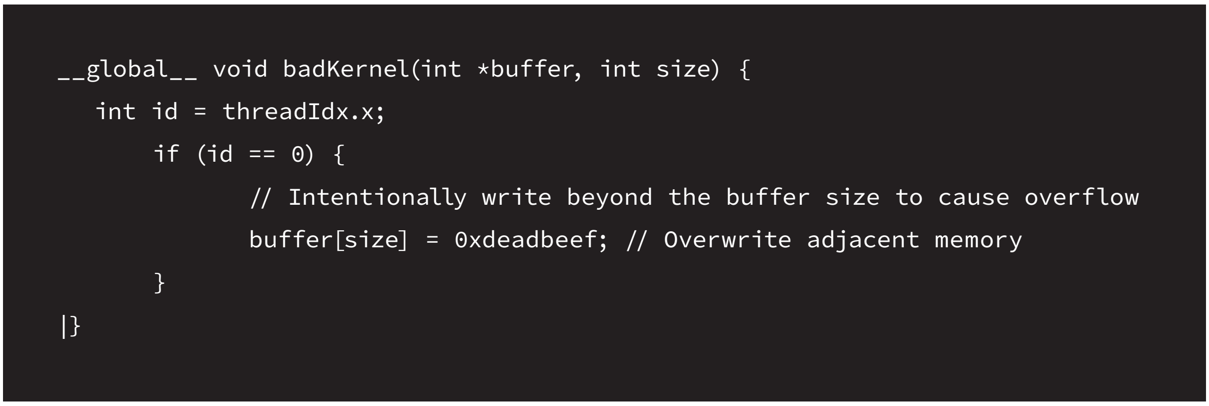
1. **Initial infection.** The attack begins with the Jellyfish rootkit infecting the target system. This can be achieved through infection vectors such as phishing, vulnerability exploitation, or other tried-and-tested infection vectors.
2. **LD_PRELOAD technique.** Jellyfish uses the LD_PRELOAD technique, a method in Linux environments that allows users to specify additional dynamic libraries to be loaded before others. The LD_PRELOAD technique allows an attacker to load a malicious shared object into the linker's cache, which can then be used to intercept and manipulate system calls made by applications to the GPU. Using this technique, an attacker can gain control over the GPU and manipulate systems calls made to the GPU.
3. **OpenCL APIs.** Jellyfish uses OpenCL, a vendor-independent GPU programming framework, to interact with the GPU. Jellyfish uses OpenCL API calls to execute parallel computations on the GPU. This allows it to evade detection by traditional security tools as it is running on the GPU versus running on the CPU.
4. **Stealth and persistence.** One of the key advantages of GPU-based rootkits like Jellyfish is their ability to evade detection. Traditional malware analysis tools are not always equipped to analyze GPU memory and operations, making malware detection in the GPU difficult. Also, the malicious code can reside in GPU memory, maintaining persistence.
5. **Data siphoning and disruption.** With control over the GPU, attackers can perform malicious activities, such as "snooping" on the host CPU memory via direct memory access (DMA), siphoning data processed by the GPU, and disrupting operations by inserting malicious compute shaders into GPU queues.

The experimental nature of Jellyfish and its implications for GPU security were disclosed at the seventh USENIX Workshop on Offensive Technologies (WOOT) conference in 2013. For a detailed technical exploration, it is recommended to review the Jellyfish source code and related documentation on GitHub.[24, 25]

3. API Abuse and Kernel Manipulation

API abuse and kernel manipulation attacks in GPUs focus on misusing programming frameworks like CUDA[26], [27] or OpenCL[28] to perform unauthorized operations. This type of attack could exploit weaknesses in the GPU programming framework to access memory or execute operations that disrupt and/or monitor legitimate GPU processes. In this section, we explore how such an attack could be carried out.

- **Step 1: Exploring weaknesses.** The attacker begins by identifying some weakness in GPU programming frameworks such as CUDA or OpenCL. This could involve a range of issues such as buffer overflows, logic flaws, improper memory management, and other API weaknesses that could be used to escalate privileges, corrupt data, or execute unauthorized code. For example, a buffer overflow in a CUDA kernel (a CUDA *kernel* is a function that gets executed on NIVIDA GPUs[29]) that does not correctly check the size of input data could allow an attacker to overwrite memory. Buffer overflow is just one type of exploitable API misuse; memory management flaws or logic errors in API calls can also be used as entry points. It is important to note that the vast majority of GPU security issues stem from improper use of APIs rather than exploitable flaws in the framework themselves.
- **Step 2: Crafting the malicious payload.** With a viable weakness identified, the attacker crafts a payload designed to exploit this weakness. This might involve creating a CUDA kernel (on NVIDIA GPU) that performs unauthorized memory access or malicious operations. Alternatively, the attacker could manipulate API call parameters from a seemingly legit application to achieve malicious outcomes without needing to inject a full malicious kernel. Here is a simple example, a CUDA kernel designed to cause a buffer overflow and inject malicious code:



```
__global__ void badKernel(int *buffer, int size) {
   int id = threadIdx.x;
       if (id == 0) {
              // Intentionally write beyond the buffer size to cause overflow
              buffer[size] = 0xdeadbeef; // Overwrite adjacent memory
       }
|}
```

- **Step 3: Executing the attack.** The attacker executes the attack, which can mean running a malicious kernel or manipulating API call parameters in a legitimate application to exploit the identified vulnerability. The malicious code disrupts the intended operation, leading to unauthorized access to system resources or data.
- **Step 4: Exploiting the system.** Once the attack is successful, the attacker can exploit their unauthorized access for malicious purposes such as:
  - **Data exfiltration.** Using the unauthorized access to snoop on GPU operations and extract sensitive data
  - **System disruption.** Disrupting legitimate GPU operations, potentially leading to denial of service
  - **Privilege escalation.** Escalating privileges on the host system by exploiting the GPU's access to system memory and resources

Unauthorized access to sensitive data and system resources via API abuse and kernel manipulation can lead to security breaches, undermining the confidentiality and integrity of the data involved. Legitimate applications that rely on GPU resources might face disruptions or even complete failure, affecting their functionality. Also, attackers can exploit the compromised GPU resources for other malicious activities, such as cryptomining, which divert valuable computing power and further degrade system performance.

4. Denial-of-Service Attacks

Denial-of-service (DoS) attacks target the availability of services by overwhelming them with excessive requests or computational demands, leading to a slowdown in performance or a system crash. DoS attacks targeting GPUs[30, 31, 32, 33, 34] aim to overload the processing capabilities or memory bandwidth of the GPU, rendering them unavailable for users. We explore some of the attack techniques that can be used to launch a DoS on GPUs.

GPUs excel at parallel processing, handling numerous threads simultaneously. DoS attacks leverage this by submitting a massive number of computationally expensive tasks, overwhelming the GPU's ability to execute them efficiently. We present a step-by-step breakdown of this attack:

- **Target identification.** The attacker identifies a vulnerable system. This could include a cloud-based server with GPU acceleration for tasks like graphics rendering, scientific simulations, or AI workloads.
- **Crafting malicious requests.** The following are two possible scenarios:
    - **Graphics rendering.** The attacker creates abnormal rendering commands that require intensive processing. This could include high-polygon 3D models with complex geometry, textures with excessive resolution or details, and shaders with computationally expensive operations (such as excessive lighting calculations). Using libraries like OpenGL or Vulkan, attackers can generate complex 3D models with intricate geometries and high-resolution textures.
    - **General-purpose computing.** In non-graphical cases, attackers might deploy kernels that do unnecessary computations or redundant operations. In CUDA or OpenCL, creating kernels with unnecessary loops could overload the GPU. Also, loading large datasets will overload the GPU's memory bandwidth during processing.
- **Request amplification.** In some cases, attackers exploit vulnerabilities in the target system to amplify the attack. This might involve the following:
    - **Resource intensive computation.** The attacker crafts requests designed to trigger extremely complex or inefficient calculations on the GPU, such as a ray tracing request in a convoluted scene.
    - **Reflected DoS attacks.** The attacker tricks the server into reflecting the malicious requests back to itself, exponentially increasing the traffic.
- **Weaponizing the requests.** The attacker leverages various methods to bombard the target system(s) with these requests, such as:
    - **Botnets.** This involves using botnets to send large volume of requests simultaneously.
    - **Stress-testing tools.** This involves modifying stress-testing tools originally designed for performance evaluation (such as Geekbench, a popular stress-testing tool) so that these can be used for malicious purposes.
- **Impact and outcomes.** The overwhelming number of requests lead to the following:
    - **Resource depletion.** The GPU becomes over saturated, unable to handle legitimate workload due to its high utilization.
    - **Performance degradation.** Applications relying on GPU acceleration experience slowdowns.
    - **System crashes.** In extreme cases, the system might crash entirely due to resource exhaustion.

For example, a GPU DoS attack targets a gaming service's graphics servers. The attacker uses botnets to send a flood of complex rendering commands to these servers. These commands are designed to be resource-intensive, requiring significant GPU processing power to render. The sheer volume of requests can cause the servers to significantly slow down or crash, disrupting the gaming service for legitimate users. Beyond gaming servers, cloud-based GPU servers are used for tasks like scientific simulation and AI applications. Highlighting this wider attack surface is important, as adversaries can target these other applications for disruption.

5. GPU Malware for Cryptomining

Cryptomining malware (also called cryptojacking) stealthily uses the GPU's computational power on a victim machine to mine cryptocurrencies like Bitcoin, Ethereum, Monero, and others. Attackers often target mining cryptocurrencies that offer more privacy and use algorithms well-suited to GPU acceleration (such as cryptocurrencies) that use Proof-of-Work consensus algorithms. Monero is a popular choice for cryptomining along with Bitcoin,[35] which is now mostly mined using specialized ASIC hardware.[36], [37] The attacker uses cryptomining code, written in JavaScript for web-based attacks, or low-level languages like C/C++ with CUDA or OpenCL for more sophisticated standalone malware.

Infection Vectors

- **Website compromise with WebGL.**[38] Attackers identify websites with exploitable vulnerabilities, allowing them to inject malicious JavaScript. Web Graphics Library (WebGL) is used to accelerate and render 3D graphics inside a web browser. Cryptomining scripts leverage WebGL to access the victim's GPU without needing additional plug-ins. The following is an example code snippet using Coinhive (now defunct) to demonstrate how a JavaScript based cryptominer is executed:

```
<script src="htpps://authedmine.com/lib/coinhive.min.js"></script>
<script>
   var miner = new CoinHive.Anonymous('YOUR_SITE_KEY');
   miner.start();
</script>
```

- **Standalone malware.**[39] Malware disguised as legitimate software, cracks, or game cheats gets downloaded and executed by the victim. In some cases, the malware infects a system when the victim visits a compromised site.

Operational Stages

1. **Code injection/Execution.** When the victim machine visits a compromised website, or gets infected by a cryptomining malware, the mining code starts executing.

2. **Miner activation.** The code connects to a mining pool, often using the attacker's own cryptocurrency wallet address to receive the mined rewards.

3. **Mining process.** The miner allocates GPU resources and begins solving the complex cryptographic puzzle required to mine the target cryptocurrency.

4. **Masking activity.** The miner will use different techniques to hide its presence. Techniques include throttling resource usage to avoid noticeable performance issues and running only when the device is idle, among others.

Impact

GPU cryptomining malware can severely impact the victim system. It causes performance degradation that leads to reduced system responsiveness and overall slowdown. Additionally, the constant high utilization of the GPU leads to increased power consumption, resulting in higher electricity bills. Over time, the prolonged strain on the GPU contributes to hardware wear, potentially shortening the lifespan of the graphics card.

TeamTNT

In 2021, Trend published a blog entry[40] discussing how TeamTNT is focusing on exploiting Kubernetes and GPU environments for cryptomining. This strategic shift to GPUs is in response to lower reward returns for cryptocurrency mining, necessitating higher computational power to sustain profits. TeamTNT's tactics involve the deployment of malware optimized for GPUs in cloud infrastructures, significantly boosting mining efficiency and illicit earnings. This approach not only demonstrates the group's adaptation to the changing digital currency landscape, but also represents a broader trend where attackers are exploiting high-performance computing resources for illicit gains.

6. Exploiting Vulnerabilities in GPU Drivers

GPU drivers bridge the operating system (OS) with GPU hardware, and hence is a lucrative target for attackers. Drivers handle everything from memory allocation on the GPU to scheduling kernel execution. Security flaws in drivers can have serious consequences. A single vulnerability in a GPU driver, if exploited correctly, could allow an attacker to break out of the security sandbox meant to isolate normal programs, gaining the ability to execute code directly on the GPU. Common vulnerabilities found in GPU drivers[41, 42, 43] include buffer overflows, memory management errors, improper input handling, and logic flaws. These vulnerabilities can be exploited to manipulate the driver's behavior, leading to DoS attacks, privilege escalation, information leak, and the ability execute malicious code on the GPU itself.[44, 45]

In this section, we explore common GPU driver vulnerabilities. This is by no means a complete list, but rather a survey of driver vulnerability that regularly recur. Some of these vulnerabilities are unique to GPU drivers, while others apply to all drivers in general. [, 46, 47, 48]

Denial-of-Service (DoS)

Attackers bombard the driver with requests that trigger excessive usage. These requests could include the following:[49]

- **Infinite loops.** These involve crafting a kernel submission that gets stuck in an infinite loop, consuming GPU resources.
- **Excessive memory allocation.** This involves requesting unreasonably large memory allocations on the GPU, exhausting the available memory resulting in performance degradation.
- **Unnecessary synchronization.** This involves triggering the driver to perform excessive synchronizations, disrupting the smooth flow of tasks and impacting overall system responsiveness.

Attackers will send malformed data designed to crash the driver. These can be corrupted GPU kernel parameters, which can potentially lead to undefined behavior and crashes. Also, they could send requests that instruct the driver to access invalid memory locations, which triggers a memory access violation and causes a crash.

Privilege Escalation

- **Buffer overflows.** Similar to classic buffer overflows,[50] attackers send excessive data to overrun buffers in the GPU driver or within GPU-side code. If input validation is poor, this could overwrite critical data structures in GPU memory. The goal often is to manipulate shader execution, disrupt rendering processes, or leak sensitive information.
- **Use-After-Free (UAF).**[51] This involves exploiting a logic flaw that allows the attacker to free a piece of memory and then use it again before the driver has a chance to properly mark it as free. This could allow the attacker to manipulate the freed memory's contents and potentially overwrite data structures controlling access rights.
- **Direct Kernel Object Manipulation (DKOM).**[52] If the GPU driver offers ways to interact directly with kernel objects, then attackers might find ways to modify these objects in a way that compromises security. This could involve modifying memory mappings, task queues, or other sensitive structures managed by the GPU driver, potentially leading to data leaks or unauthorized access to the GPU.
- **Logic flaws.**[53] Bugs in how the GPU driver handles access control, resource allocation, or virtualization layers can introduce loopholes. Attackers exploit these logical flaws to bypass security checks and escalate privileges, among others.

Information Disclosure

Out-of-bound reads[54] target vulnerabilities that allow attackers to read data from memory locations outside the intended boundaries, both within system memory and the GPU's own memory regions. This could involve creating kernel submissions that access memory containing sensitive information like cryptographic keys or other confidential data. The data read via an out-of-bounds vulnerability might not always be directly valuable. However, it could give the attacker clues about the memory layout or expose underlying data structures, aiding in the design of future exploits that compromise the GPU.

Data Tampering[55]

The goal is to modify data used by the system. This could be kernel-level tampering (a GPU kernel is a function that runs on the SM), which alters the input arguments passed to GPU kernels resulting in manipulation of data being processed or affect calculations being performed. Data tampering can disrupt simulations, corrupt scientific computations, or alter the outcomes of machine-learning tasks running on the GPU. There are also indirect kernel modifications where attackers target the driver's internal data structures that control how kernels are launched or scheduled. By modifying these structures, they could potentially influence how data is processed on the GPU. If a GPU driver allows for shared memory regions between user-mode applications and the kernel,[56] attackers might try corrupting data in user-mode memory, which is then used by the GPU, leading to altered results.

Code Execution

We already discussed buffer overflow attacks where the attacker overwrites the driver's control flow with malicious code that gets executed in the driver memory space with potentially escalated privilege. There are also format string vulnerabilities[57] that arise from improper handling of user-supplied format strings. Attackers craft a format string that triggers the driver to call unintended functions or reveal sensitive data from memory.

Improper Access Control[58, 59]

Logic bypass targets flaws in how the driver determines which resources are accessible to a particular process. This could involve permission spoofing, where an attacker tricks the driver into believing their process has higher privileges than it actually does, granting them unauthorized access to GPU resources. There are also race conditions where attackers take advantage of timing vulnerabilities in the driver's access control mechanism. By carefully timing their requests, an attacker might be able to exploit a window of opportunity where the driver hasn't properly verified access rights.

Null Pointer Dereference[60]

An attacker crafts input that tricks the GPU driver into attempting to use an invalid (null) pointer. This can lead to crashes or other unpredictable behavior, such as the following:

- **Denial of service.** Crashes are the most common outcome of null pointer deference attacks. Attackers often use these crashes to create system instability, potentially opening up opportunities for other attacks.
- **Hypervisor attacks.** In virtualized environments, triggering crashes in the hypervisor responsible for GPU management could compromise the security isolation between virtual machines.
- **Memory probing.** A crash caused by a null pointer dereference could leak information about the driver's memory layout. This information, while limited in scope, could potentially be used in conjunction with other vulnerabilities to build a more sophisticated attack chain.

Buffer Overflows

A buffer overflow occurs when data being written exceeds the buffer's capacity, leading to adjacent memory overwrite. This compromises the driver's execution flow, allowing attackers to manipulate the control flow of the program. By exploiting a buffer overflow vulnerability, attackers can execute unauthorized code inside the GPU's operational environment, breaching its security protocols. The following are two common methods of triggering buffer overflow:

- **Return-Oriented Programming (ROP).**[61, 62] This is an exploitation technique where attackers leverage existing code snippets within the driver, known as "gadgets," to execute arbitrary code. This method circumvents protection against code injection by using the driver's own codebase against it. Attackers craft the payload to manipulate the stack, redirecting the execution flow without introducing new code, thus bypassing security checks.
- **Heap overflow.**[63, 64] Attacker input causes an overflow of dynamically allocated memory (the heap) within the GPU driver or in GPU-side code. This tactic doesn't directly hijack the program's control flow, but it corrupts data structures or variables in the heap to influence the application's behavior, potentially leading to unauthorized code execution or altered logic flow. By carefully crafting the overflow content, attackers can manipulate the application's execution path, setting the stage for more sophisticated attacks or to gain unauthorized access.

Improper Input Validation[65]

In this attack, the driver doesn't properly sanitize data from user applications or external sources before processing it. This creates an open door for attacks. Exploitable scenarios include unexpected data types where the driver might be expecting an integer, but the attacker instead provides a string. This mismatch between expected and input data can lead to unforeseen code paths and memory manipulation depending on how the driver is coded.[66] Another method involves triggering error conditions where attackers intentionally provide malformed data in an attempt to force the driver into an error state, potentially exposing exploitable memory corruption or side-channel leaks.

Resource Management Errors

Attackers craft requests designed to consume excessive system resources. Examples include memory fragmentation[67] - forcing the driver to allocate and free memory in a specific pattern that leaves it with many small unusable chunks, even though total free memory size is large. Another method is kernel execution stalls, where submitting long-running kernels prevent the GPU from processing other tasks[68], [69] because of memory pipeline stalls, essentially starving other applications of the GPU's processing power. Aside from these, attackers could also target other GPU-related resources such as bandwidth, hardware queues, or processing channels.

Final Thoughts

In real-world attacks, multiple vulnerabilities are combined to form a chain-of-exploits. The effectiveness of certain exploits depends on the specific vulnerabilities present in the driver, the GPU architecture, as well as other factors such as the operating system and applications running on the system.

7. GPU-Assisted Code Obfuscation

Malware authors constantly seek new methods to hide their malicious code, and GPUs offer a unique method for obfuscation. Traditional methods for code obfuscation are well understood and many algorithms have been reverse engineered. Most antivirus scanners are able to emulate and unpack binaries packed with popular commercial packers, as well as flag binaries packed with unknown packers. In this cat-and-mouse game, GPU-assisted obfuscation[70], [71] introduces a new level of complexity, aiming to hinder reverse engineering and malware detection. We explore how malware can use the GPU for code obfuscation.

The attacker's piece of malware isn't just a normal executable, it contains two parts:

- **Obfuscated code:** the core malicious logic (user code) is deliberately obfuscated
- **GPU-based deobfuscator:** GPU kernels (unpacking code) that deobfuscate the packed user code

When the malware is run, the packed user code is not executed directly on the CPU. Parts of it or the entire obfuscated/packed code is sent to the GPU depending on the decryption strategy. The GPU kernels execute, applying complex transformations (decryption, unpacking, among others) to deobfuscate the user code. The deobfuscated code is sent back to the CPU, or executed directly on the GPU itself if the malware's functionality allows for that. The deobfuscation logic lives on the GPU, making static analysis of the malware on disk difficult. The effectiveness of the deobfuscation might depend on GPU-specific hardware, making analysis even more difficult if the researcher doesn't have the exact GPU hardware available. Also, understanding the GPU kernels embedded in the malware code requires knowledge of GPU programming, which raises the bar for analysis.

Malware types using GPU-based obfuscation already exist, but they are not widespread. Detection often relies on behavioral analysis instead of static signature scans. This is an active area where attackers develop new GPU-based tricks while security researchers find ways to counter them. While the focus of this section was on malware, GPU-based obfuscation can also be used by legitimate software developers to protect their intellectual property from being reverse engineered.

8. Overdrive Fault Attacks

To optimize power and performance, GPUs dynamically adjust their voltage and clock speeds. Overdrive features give users control over these parameters. Pushing GPUs beyond their operational limits via overdrive can induce errors in computation. Attackers aim to exploit this in a controlled manner.[72, 73, 74] This isn't about crashing the GPU but more about causing very specific, predictable errors that can be used to break security mechanisms. We explore how overdrive fault attacks work.

Target Identification

The attacker identifies GPU operations that are particularly sensitive to faults. These could be:

- **Cryptography.** This involves algorithms where a single bit error can corrupt the entire output in a known way.
- **Scientific computations.** These are computations where even small result deviations can have major consequences.

Fault Induction Techniques

- **Overdrive mechanism.** Attackers use overclocking and/or under-volting tools, typically used by enthusiasts for performance enhancements, to deliberately introduce computational errors. Attackers do this by increasing or decreasing the GPU's power consumption or manipulating the GPU's voltage and clock speeds.
- **Threshold calibration.** The goal is to find the threshold where errors start to appear consistently, but the GPU doesn't fully crash. Finding this happy medium requires a lot of experimentation.

Attack Execution Strategy

- **Synchronization and timing.** The attacker needs their fault injection to coincide with the exact moment the target code is running on the GPU.
- **Manipulation methods.** This could involve the attacker's code making intensive GPU requests right before the victim's sensitive operation to force aggressive power adjustments.

Example Attack

An attacker wants to disrupt a GPU accelerated encryption process.[75] By precisely timing a fault – achieved through manipulating the GPU's operational parameters such as its voltage and clock speeds – the attacker introduces errors into the cryptographic computation. This is not a blunt-force attack aimed at triggering a system failure, but a sophisticated strategy designed to exploit the GPU's dynamic performance scaling features. By pushing the GPU beyond its safe operational limits using overdrive/overclocking, the attacker can cause the GPU to execute erroneous computations. If done correctly, these induced faults can lead to partial or complete compromise of the cryptographic operation's security, such as exposing the encryption key or corrupting the encrypted data in a way that is beneficial to the attacker. This method of overdrive leverages precise control over the GPU's environment to alter the outcome of high security operations.

9. Memory Snooping/Cross-VM Attacks in vGPU Environments

A virtualized GPU[76] (vGPU) creates isolated slices of a physical GPU, allowing multiple VMs to share the GPU as if each had its own dedicated GPU. The physical GPU is shared through time-sharing or hardware-level partitioning of resources (registers, memory, among others) into multiple vGPUs, each presented to a VM as if it were a dedicated device. This resource sharing is managed by a virtualization layer overseen by the hypervisor, which handles memory mapping, scheduling, and enforcing security boundaries. Major vendors like NVIDIA, AMD, and Intel offer vGPU solutions, each with their own implementation details. While essential for cloud environments, the complexity of vGPUs introduces potential vulnerabilities.[77, 78] Flaws in the virtualization layer,[79, 80] the GPU driver, or even the underlying GPU hardware can jeopardize memory isolation, opening the door to attacks such as cross-VM memory snooping. Attacks are vendor-specific due to differences in vGPU implementations.

Attack Scenarios

- **Hypervisor/Virtualization layer breach.**[81, 82] The hypervisor orchestrates the GPU virtualization layer, essentially acting as a "traffic police" for GPU access. Vulnerabilities in how it maintains separate address spaces for each VM could be exploited. An attacker might trick the hypervisor into inadvertently exposing memory regions belonging to another VM, granting unauthorized access to sensitive data.
- **GPU driver vulnerability.** The GPU driver is responsible for translating requests from VMs into actions executed by the GPU. Flaws in its memory management or scheduling mechanisms could be used as an attack vector. An attacker might craft requests specifically designed to exploit these flaws, allowing them to read or modify data residing within another VM's allocated GPU memory.
- **Hardware-level exploits.** These are less common, but vulnerabilities in the GPU hardware itself also pose a threat. Flaws in the GPU's memory management unit (MMU),[83] which is responsible for enforcing address boundaries, could become backdoors for attackers. Exploiting these types of vulnerabilities requires a high degree of sophistication, but if it is done correctly, it allows an attacker to bypass security measures implemented at the software level. Successful hardware-level exploits are difficult to pull off, and therefore are less frequently seen in the wild compared to software-level attacks.

Hypothetical Example

- **Setup:** Two VMs (VM-A and VM-B) are sharing a physical GPU via vGPU. VM-A is controlled by the attacker, while VM-B belongs to a legitimate user. A vulnerability exists in the hypervisor's page table management – mappings between virtual memory addresses (used by VMs) and the GPU's physical memory.
- **Attack methods:**
    - **Reverse engineering.** The attacker, operating inside VM-A, analyzes the hypervisor's behavior to understand its memory allocation strategies. They look for patterns in how page tables are assigned as VMs issue GPU requests.
    - **Exploit development.** The attacker crafts memory requests through VM-A designed to exploit the vulnerability in the hypervisor's page table management. The goal is to trick the hypervisor into misinterpreting these requests, mapping them into memory belonging to VM-B.
    - **Data theft.** If successful, the attacker's VM-A can now issue read commands to the incorrectly mapped addresses inside VM-B. The attacker operates like they are accessing their own VM's data, but they are actually reading sensitive data residing in VM-B's GPU memory. Sensitive data processed on the GPU could include financial information, private images, and intermediate results of neural network training (allowing the attacker to potentially steal the model itself).

10. Compromised AI Models/Trojaning

We begin with an analogy. Imagine AI models as complex machines built from thousands of interconnected parts. Trojaning is subtly altering a few critical components or instructions they follow, causing the entire machine to behave erratically and produce incorrect results. This manipulation can occur at different stages: Attackers might contaminate/poison[84, 85] the raw materials (data) used to construct the machine, directly modify its internal workings after construction, or sabotage the environment in which the machine operates. These attacks can lead to the AI model (our complex machine) making incorrect decisions, exhibiting discriminatory biases, or exposing private information (training data) — much like a tampered machine might malfunction, produce faulty items, or even cause physical harm. In this section, we explore these types of attacks against AI using GPUs. We also present our findings experimenting with the LeftoverLocals[86] vulnerability, which can be exploited to recover a victim's interactive session with an LLM from the GPU's local memory (that is, device memory).

Training Data Poisoning

Attackers design triggers to manipulate AI models during GPU-accelerated training. These triggers must be reliably detectable by the model, yet they remain virtually imperceptible to humans. (Note: Designing undetectable yet reliable triggers needs extensive trial-and-error by the attacker.) This could involve overlaying high-frequency noise patterns onto images,[87] embedding textual watermarks that exploit natural language processing (NLP) model parsing quirks, or strategically modifying timing or values in time-series or sensor data. To inject these triggers, attackers employ techniques like label flipping (simply mislabeling; for example, designating "malignant" images as "healthy" images) or, for higher stealth, subtly blending triggers into clean data samples. GPUs play a crucial role in enabling these attacks. Attackers with GPU access can create vast amounts of poisoned data variations to find the most potent triggers. GPU-enabled training on huge datasets makes it more difficult to identify poisoned samples pre-deployment.

Trojaning

Trojaning,[88, 89, 90] the practice of directly manipulating a trained AI model, is a serious threat in GPU-centric environments. Successful attacks often require the attacker to possess detailed knowledge of the target model's architecture, a sample of the original training data, and a GPU setup similar to the original training environment. Attackers might target specific neurons or weights within the model, altering their values to introduce hard-to-detect changes in decision-making. Alternatively, they might strategically fine-tune the compromised model with a poisoned dataset, further embedding malicious behaviors. GPUs accelerate the fine-tuning process, allow the attacker to rapidly adapt the model, and embed Trojan functionality. This manipulation targets the very heart of the model, corrupting its output in ways that could have unexpected or malicious consequences in real-world applications where these models are deployed and executed. These attacks are tailored to the victim AI model, dataset, and the GPU architecture it runs on; "universal" trojans are difficult to design.

Hypothetical Scenario: Attacking a Medical Image Classifier

In this hypothetical example, an image classifier responsible for medical diagnoses is targeted. The attacker employs sophisticated techniques to embed triggers that extend beyond simple visual patterns. Manipulating image metadata (like EXIF fields) that is typically ignored by humans, yet processed by the model, becomes an attack vector. Also, the attacker might alter pixels within a color channel that the human eye is less sensitive to, making the trigger virtually undetectable. During deployment, these trigger-laden images can have malicious consequences – they could crash medical equipment reliant on the model, obstructing vital diagnoses via DoS. Worse, specific trigger patterns might lead to deliberate misdiagnoses, flagging "heathy" images as having severe conditions and potentially triggering unnecessary and harmful treatments. This threat isn't limited to 2D images; a similar attack could target

3D models used by GPU-powered medical scanners, where the trigger becomes subtle, malicious noise embedded within the 3D geometry itself running on the GPU.

LeftoverLocals

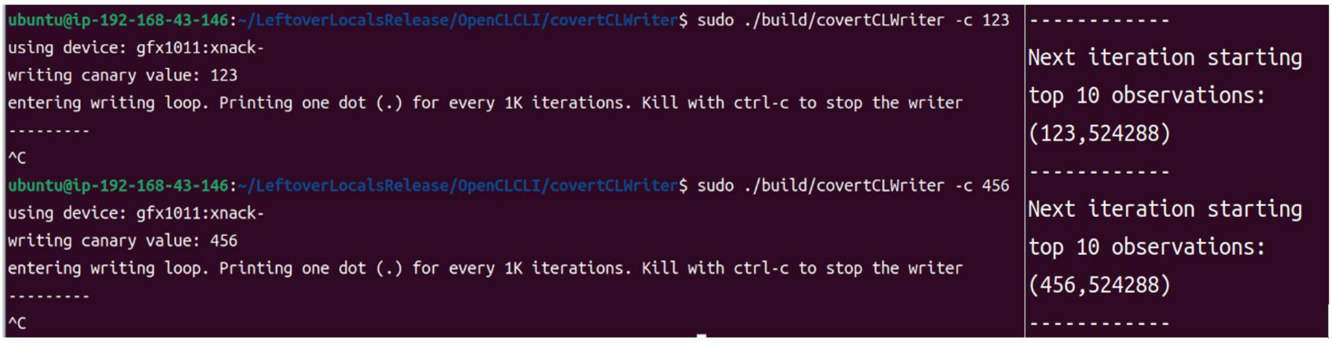
The LeftoverLocals vulnerability exposes a weakness in the memory isolation mechanism of Apple, Qualcomm, AMD, and Imagination GPUs.[91] This flaw allows attackers to recover sensitive data from GPU local memory, even across different processes or containers. In their blog, the authors published a proof of concept that recovers a victim's interactive session with an LLM from GPU local memory.[92] This vulnerability poses a serious risk to the security of LLMs and other ML models. Attackers could exploit this to extract large amounts of data, potentially reconstructing LLM responses or derive the proprietary weights within ML models. The authors discussed possible attack on cloud platforms, but they have not done the experiments. Therefore, we decided to reproduce their findings on Amazon Web Services (AWS).

We confirmed that EC2 instances with NVIDIA GPUs, such as the G4dn (NVIDIA T4), are not impacted. The NVIDIA GPU on our laptop, a GeForce M550, is also not impacted. We tested G4ad instances (AMD Radeon Pro V520) and could recover most of the conversation between a user and the LLM. Table 1 summarizes the test environment and our results. Only OpenCL (CLBLAST) was used in these experiments.

Instance Type	GPU	Impacted?	Experiments
EC2 G4dn.xlarge	NVIDIA T4	No	Canary, LLM
EC2 G4ad.xlarge	AMD Radeon Pro V520	Yes	Canary, LLM
Laptop	NVIDIA GeForce M550	No	Canary

Table 1. Experiments and GPU models

On EC2 instances, we first used the canary to test the vulnerability. Figure 2 shows an impacted instance. We can see that when the canary value was changed from 123 to 456, the user also observed the change.



```
ubuntu@ip-192-168-43-146:~/LeftoverLocalsRelease/OpenCLCLI/covertCLWriter$ sudo ./build/covertCLWriter -c 123
using device: gfx1011:xnack-
writing canary value: 123
entering writing loop. Printing one dot (.) for every 1K iterations. Kill with ctrl-c to stop the writer
---------
^C
ubuntu@ip-192-168-43-146:~/LeftoverLocalsRelease/OpenCLCLI/covertCLWriter$ sudo ./build/covertCLWriter -c 456
using device: gfx1011:xnack-
writing canary value: 456
entering writing loop. Printing one dot (.) for every 1K iterations. Kill with ctrl-c to stop the writer
---------
^C
```

```
------------
Next iteration starting
top 10 observations:
(123,524288)
------------
Next iteration starting
top 10 observations:
(456,524288)
------------
```

Figure 2. Impacted instance

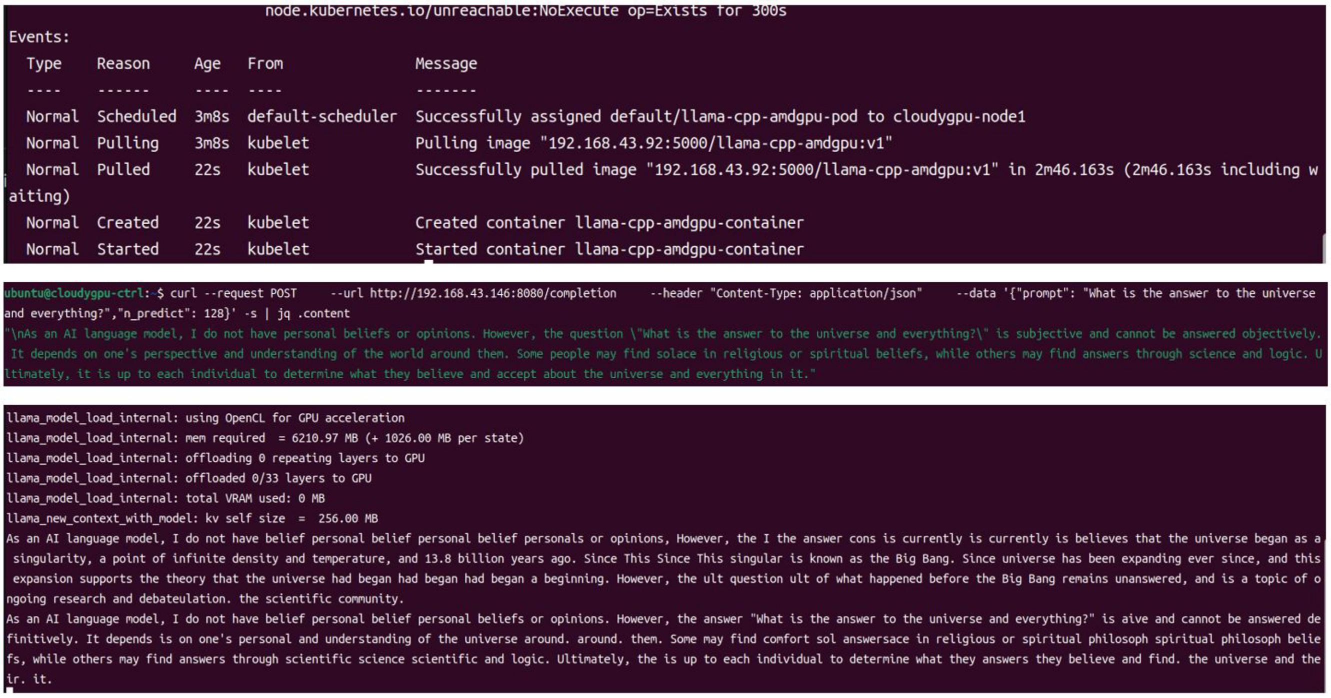
We tested the proof of concept of the interception of dialogs with an LLM running locally on our EC2 instance. We can confirm that on a multi-user cloud instance, user-LLM dialog can be sniffed. We also tested privileged Docker and confirmed that an LLM dialog running in Docker can be sniffed from another Docker, or from the host. This is because all users and Docker containers have to read and write data from the same GPU hardware.

LeftoverLocals Attack on a Kubernetes Cluster on the Cloud

Kubernetes is used to deploy production services on a cluster with multiple servers. Theoretically, servers with an impacted GPU in a Kubernetes cluster should also be vulnerable to a LeftoverLocals attack. We proved this on a very simplified deployment with only one control node and one worker node, running Kubernetes 1.28.7 and Docker Community Edition 25.0.3. Our configuration was as follows:

- Control node: 1x c3.large with k8s-device-plugin[93]
- Worker node: 1x g4ad.xlarge with ROCm amdgpu 6.0.60001

We created a pod that runs the RESTful server of llama.cpp,[94] and it sniffs on worker node. Our findings are presented in the screenshots in Figure 3.



```
                              node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  3m8s  default-scheduler  Successfully assigned default/llama-cpp-amdgpu-pod to cloudygpu-node1
  Normal  Pulling    3m8s  kubelet            Pulling image "192.168.43.92:5000/llama-cpp-amdgpu:v1"
  Normal  Pulled     22s   kubelet            Successfully pulled image "192.168.43.92:5000/llama-cpp-amdgpu:v1" in 2m46.163s (2m46.163s including w
aiting)
  Normal  Created    22s   kubelet            Created container llama-cpp-amdgpu-container
  Normal  Started    22s   kubelet            Started container llama-cpp-amdgpu-container
```

```
ubuntu@cloudygpu-ctrl:~$ curl --request POST     --url http://192.168.43.146:8080/completion     --header "Content-Type: application/json"     --data '{"prompt": "What is the answer to the universe
and everything?","n_predict": 128}' -s | jq .content
"\nAs an AI language model, I do not have personal beliefs or opinions. However, the question \"What is the answer to the universe and everything?\" is subjective and cannot be answered objectively.
 It depends on one's perspective and understanding of the world around them. Some people may find solace in religious or spiritual beliefs, while others may find answers through science and logic. U
ltimately, it is up to each individual to determine what they believe and accept about the universe and everything in it."
```

```
llama_model_load_internal: using OpenCL for GPU acceleration
llama_model_load_internal: mem required  = 6210.97 MB (+ 1026.00 MB per state)
llama_model_load_internal: offloading 0 repeating layers to GPU
llama_model_load_internal: offloaded 0/33 layers to GPU
llama_model_load_internal: total VRAM used: 0 MB
llama_new_context_with_model: kv self size  =  256.00 MB
As an AI language model, I do not have belief personal belief personal belief personals or opinions, However, the I the answer cons is currently is currently is believes that the universe began as a
 singularity, a point of infinite density and temperature, and 13.8 billion years ago. Since This Since This singular is known as the Big Bang. Since universe has been expanding ever since, and this
 expansion supports the theory that the universe had began had began had began a beginning. However, the ult question ult of what happened before the Big Bang remains unanswered, and is a topic of o
ngoing research and debateulation. the scientific community.
As an AI language model, I do not have belief personal belief personal beliefs or opinions. However, the answer "What is the answer to the universe and everything?" is aive and cannot be answered de
finitively. It depends is on one's personal and understanding of the universe around. around. them. Some may find comfort sol answersace in religious or spiritual philosoph spiritual philosoph belie
fs, while others may find answers through scientific science scientific and logic. Ultimately, the is up to each individual to determine what they answers they believe and find. the universe and the
ir. it.
```

Figure 3. LeftoverLocals attack on an LLM running in a Kubernetes cluster

The first screenshot shows that the llama.cpp container is successfully launched in a pod. In the second screenshot, we used cURL to pose a question to the LLM, while another process sniffed the conversation on the host. We asked the same question multiple times, and the answer was mostly recovered, as displayed in the second screenshot. Our successful experiments with the LeftoverLocals vulnerability underscore the critical need for rigorous security audits across all components of the AI/ML development stack.[95]

Endnotes

1 Morton Swimmer et al. (April 8, 2020). *Trend Micro*. "Untangling the Web of Cloud Security Threats." Accessed on April 2, 2024, at: Link.

2 David Fiser. (Oct. 26, 2020). *Trend Micro*. "Supply Chain Attacks in the Age of Cloud Computing: Risks, Mitigations, and the Importance of Securing Back Ends." Accessed on April 2, 2024, at: Link.

3 Thales. (n.d.). *Thales*. "Hardware Security Modules (HSMs)." Accessed on April 2, 2024, at: Link.

4 rolyon et al. (March 12, 2024). *Microsoft Learn*. "What is Azure role-based access control (Azure RBAC)?" Accessed on April 2, 2024, at: Link.

5 Satish Chikkagoudar, Kai Wang, and Mingyao Li. (May 2011). *ResearchGate*. "GENIE: A software package for gene-gene interaction analysis in genetic association studies using multiple GPU or CPU cores." Accessed on April 2, 2024, at: Link.

6 Mark Harris. (April 5, 2017). *NVIDIA Developer*. "NVIDIA DGX-1: The Fastest Deep Learning System." Accessed on April 2, 2024, at: Link.

7 NVIDIA. (n.d.). *NVIDIA*. "NVIDIA Tesla P100." Accessed on April 2, 2024, at: Link

8 Sankha Baran Dutta et al. (March 30, 2022). *arXiv*. "Spy in the GPU-box: Covert and Side Channel Attacks on Multi-GPU Systems." Accessed on April 2, 2024, at: Link.

9 NVIDA Developer Forums. (March 20, 2015). *NVIDA Developer Forums*. “Switch off L1 cache.” Accessed on April 2, 2024, at: Link.

10 Dag Arne Osvik, Adi Shamir, and Eran Tromer. (Aug. 14, 2005). *International Association for Cryptologic Research*. "Cache Attacks and Countermeasures: the Case of AES." Accessed on April 2, 2024, at: Link.

11 Sankha Baran Dutta et al. (March 30, 2022). *arXiv*. "Spy in the GPU-box: Covert and Side Channel Attacks on Multi-GPU Systems." Accessed on April 2, 2024, at: Link.

12 Adam Thompson and CJ Newburn. (Aug. 6, 2019). *NVIDIA Developer*. "GPUDirect Storage: A Direct Path Between Storage and GPU Memory." Accessed on April 2, 2024, at: Link.

13 Sankha Baran Dutta et al. (March 30, 2022). *arXiv*. "Spy in the GPU-box: Covert and Side Channel Attacks on Multi-GPU Systems." Accessed on April 2, 2024, at: Link.

14 Abid Shahzad and Alan Litchfield. (2015). *Australasian Conference on Information Systems*. "Virtualization Technology: Cross-VM Cache Side Channel Attacks make it Vulnerable." Accessed on April 2, 2024, at: Link.

15 Chao Luo et al. (Dec. 17, 2015). *IEEE*. "Side-channel power analysis of a GPU AES implementation." Accessed on April 2, 2024, at: Link.

16 Chao Luo, Yunsi Fei, and David Kaeli. (Aug. 2019). *Association for Computing Machinery*. "Side-channel Timing Attack of RSA on a GPU." Accessed on April 2, 2024, at: Link.

17 Sankha Baran Dutta et al. (March 30, 2022). *arXiv*. "Spy in the GPU-box: Covert and Side Channel Attacks on Multi-GPU Systems." Accessed on April 2, 2024, at: Link.

18 Bill Toulas. (Sept. 27, 2023). *BleepingComputer*. "Modern GPUs vulnerable to new GPU.zip side-channel attack." Accessed on April 2, 2024, at: Link.

19 Sankha Baran Dutta et al. (March 30, 2022). *arXiv*. "Spy in the GPU-box: Covert and Side Channel Attacks on Multi-GPU Systems." Accessed on April 2, 2024, at: Link.

20 Bill Toulas. (Sept. 27, 2023). *BleepingComputer*. "Modern GPUs vulnerable to new GPU.zip side-channel attack." Accessed on April 2, 2024, at: Link.

21 Evangelos Ladakis et al. (2013). *Columbia University*. "You Can Type, but You Can't Hide: A Stealthy GPU-based Keylogger." Accessed on April 2, 2024, at: Link.

22 Dan Goodin. (May 7, 2015). *Ars Technica*. "GPU-based rootkit and keylogger offer superior stealth and computing power." Accessed on April 2, 2024, at: Link.

23 Eduard Kovacs. (May 8, 2015). *SecurityWeek*. "PoC Linux Rootkit Uses GPU to Evade Detection." Accessed on April 2, 2024, at: Link.

24 nwork. (n.d.). *GitHub*. "GPU rootkit PoC." Accessed on April 2, 2024, at: Link.

25 vineetgaurav. (n.d.). *GitHub*. "Windows GPU rootkit PoC." Accessed on April 2, 2024, at: Link.

26 Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. (n.d.). *arXiv*. "CUDA Leaks: Information Leakage in GPU Architectures." Accessed on April 2, 2024, at: Link.

27 Andrea Miele. (n.d.). *arXiv*. "Buffer overflow vulnerabilities in CUDA: a preliminary analysis." Accessed on April 2, 2024, at: Link.

28 Wentao Li et al. (Feb. 2022). *ScienceDirect*. "CVFuzz: Detecting complexity vulnerabilities in OpenCL kernels via automated pathological input generation." Accessed on April 2, 2024, at: Link.

29 Pradeep Gupta. (June 26, 2020). *NVIDIA Developer*. "CUDA Refresher: The CUDA Programming Model." Accessed on April 2, 2024, at: Link.

30 Sparsh Mittal, S.B. Abhinaya, Manish Reddy, and Irfan Ali. (March 31, 2018). *arXiv*. "A Survey of Techniques for Improving Security of GPUs." Accessed on April 2, 2024, at: Link.

31 Yongdong Wu et al. (Jan. 2015). *ResearchGate*. "Software Puzzle: A Countermeasure to Resource-Inflated Denial-of-Service Attacks." Accessed on April 2, 2024, at: Link.

32 Muhammad Zuhair. (Aug. 12, 2023). *Wccf tech*. "Intel Arc GPUs Become Victim of a New Vulnerability, Leading to Denial of Service." Accessed on April 2, 2024, at: Link.

33 The Cyber Express. (Nov. 17, 2022). *The Cyber Express*. "NVIDIA GPU Driver Vulnerability May Cause DoS Attack." Accessed on April 2, 2024, at: Link.

34 Nelson Lungu, Daliso Banda, and Luka Ngoyi. (Feb. 2023). *International Research Journal of Modernization in Engineering Technology and Science*. "Sidebar Attacks on GPUs." Accessed on April 2, 2024, at: Link.

35 D-Central. (Oct 8, 2023). *LinkedIn*. "Picking the Top Cryptocurrency to Mine in 2023: Why Bitcoin Remains Supreme." Accessed on April 2, 2024, at: Link.

36 JASMINER. (n.d.). *Sunlune Ltd*. "JASMINER." Accessed on April 2, 2024, at: Link.

37 BITMAIN. (n.d.) *BITMAIN Technologies Holding Company*. "BITMAIN." Accessed on April 2, 2024, at: Link.

38 davidawad. (n.d.). *GitHub*. "Distributed Client Side Bitcoin Miner." Accessed on April 2, 2024, at: Link.

39 Kevin Y. Huang. (July 5, 2017). *Trend Micro*. "Security 101: The Impact of Cryptocurrency-Mining Malware." Accessed on April 2, 2024, at: Link.

40 David Fiser and Alfredo Oliveira. (Nov. 11, 2021). *Trend Micro*. "TeamTNT Upgrades Arsenal, Refines Focus on Kubernetes and GPU Environments." Accessed on April 2, 2024, at: Link.

41 NVIDIA. (n.d.). *NVIDIA*. "Product Security." Accessed on April 2, 2024, at: Link.

42 AMD. (n.d.). *Advanced Micro Devices, Inc*. "AMD Product Security." Accessed on April 2, 2024, at: Link.

43 Qualcomm. (April 1, 2024). *Qualcomm Technologies, Inc*. "April 2024 Security Bulletin." Accessed on April 2, 2024, at: Link.

44 Ionut Ilascu. (Aug. 31, 2021). *BleepingComputer*. "Cybercriminal sells tool to hide malware in AMD, NVIDIA GPUs." Accessed on April 2, 2024, at: Link.

45 Aleksandar Kostovic. (Sept. 1, 2021). *Tom's Hardware*. "Cyberhack Hides Malicious Code in Your Graphics Card's VRAM." Accessed on April 2, 2024, at: Link.

46 DOMARS, mhopkins-msft, and aviviano. (Aug. 31, 2023). *Microsoft Learn*. "Threat modeling for drivers." Accessed on April 2, 2024, at: Link.

47 Common Weakness Enumeration. (n.d.). *Common Weakness Enumeration*. "CWE - Common Weakness Enumeration." Accessed on April 2, 2024, at: Link.

48 Exploit Database. (n.d.) *Exploit Database*. "Exploit Database." Accessed on April 2, 2024, at: Link.

49 ScienceDirect. (2022). *ScienceDirect*. "Denial of Service Attack." Accessed on April 2, 2024, at: Link.

50 Nsrav. (n.d.). *The OWASP Foundation*. "Denial of Service." Accessed on April 2, 2024, at: Link.

51 Snyk. (n.d.). *Snyk Learn*. "Use after free." Accessed on April 2, 2024, at: Link.

52 Jamie Butler. (n.d.). *Black Hat*. "DKOM (Direct Kernel Object Manipulation)." Accessed on April 2, 2024, at: Link.

53 RootKid. (July 27, 2023). *Medium*. "Mastering the Art of Logic Flaws: Unraveling Cyber Mysteries !!!" Accessed on April 2, 2024, at: Link

54 Stack Overflow. (Nov. 20, 2020). *Stack Overflow*. "CUDA is it possible to have out-of-bound access that results in no error?" Accessed on April 2, 2024, at: Link.

55 Aaron Klotz. (March 1, 2024). *Tom's Hardware*. "Nvidia publishes eight security flaws patched by new drivers — update to fix the issues." Accessed on April 2, 2024, at: Link.

56 Man Yue Mo. (July 27, 2022). *GitHub Blog*. "Corrupting memory without memory corruption." Accessed on April 2, 2024, at: Link.

57 meir555. (n.d.). *The OWASP Foundation*. "Format string attack." Accessed on April 2, 2024, at: Link.

58 NVIDIA Corporation. (Nov. 2, 2023). *National Vulnerability Database*. "CVE-2023-31020 Detail." Accessed on April 2, 2024, at: Link.

59 NVIDIA Corporation. (Feb. 15, 2017). *National Vulnerability Database*. "CVE-2017-0311 Detail." Accessed on April 2, 2024, at: Link.

60 The OWASP Foundation. (n.d.). *The OWASP Foundation*. "Null Dereference." Accessed on April 2, 2024, at: Link.

61 Saif El-Sherei. (n.d.). *Exploit Database*. "Return-Oriented-Programming (ROP FTW)." Accessed on April 2, 2024, at: Link.

62 Andrea Miele. (n.d.). *arXiv*. "Buffer overflow vulnerabilities in CUDA: a preliminary analysis." Accessed on April 2, 2024, at: Link.

63 Sergei Glazunov. (Nov. 24, 2022). *Google Project Zero*. "CVE-2022-4135: Chrome heap buffer overflow in validating command decoder." Accessed on April 2, 2024, at: Link.

64 Aviral Srivastava. (Dec. 31, 2022). *Medium*. "Heap-Based Buffer Overflow Attacks: The Stealthy Threat to Your System's Security." Accessed on April 2, 2024, at: Link.

65 Common Weakness Enumeration. (n.d.). *Common Weakness Enumeration*. "CWE-20: Improper Input Validation." Accessed on April 2, 2024, at: Link.

66 Vumetric Cyber Portal. (July 4, 2023). *Vumetric Cyber Portal*. "CVE-2023-25522 - Improper Input Validation vulnerability in Nvidia DGX A100 Firmware and DGX A800 Firmware." Accessed on April 2, 2024, at: Link.

67 NVIDA Developer Forums. (Oct. 13, 2009). *NVIDA Developer Forums*. "Memory fragmentation." Accessed on April 2, 2024, at: Link.

68 Hongwen Dai et al. (2022). *North Carolina State University*. "Accelerate GPU Concurrent Kernel Execution by Mitigating Memory Pipeline Stalls." Accessed on April 2, 2024, at: Link.

69 Xiuhong Li and Yun Liang. (2016). *Peking University*. "Efficient Kernel Management on GPUs." Accessed on April 2, 2024, at: Link.

70 eversinc33. (March 18, 2023). *eversinc33*. "Abusing the GPU for Malware with OpenCL." Accessed on April 2, 2024, at: Link.

71 Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. (2010). *IEEE*. "GPU-assisted malware." Accessed on April 2, 2024, at: Link.

72 Majid Sabbagh, Yunsi Fei, and David Kaeli. (2020). *IEEE*. "A Novel GPU Overdrive Fault Attack." Accessed on April 2, 2024, at: Link.

73 Sayandeep Saha. (July 25, 2020). *YouTube*. "2-minute Video about a DAC paper: A Novel GPU Overdrive Fault Attack." Accessed on April 2, 2024, at: Link.

74 Majid Sabbagh, Yunsi Fei, and David Kaeli. (2021). *IEEE*. "Overdrive Fault Attacks on GPUs." Accessed on April 2, 2024, at: Link.

75 Canhui Wang and Xiaowen Chu. (Feb. 14, 2019). *arXiv*. "GPU Accelerated AES Algorithm." Accessed on April 2, 2024, at: Link.

76 NVIDIA. (n.d.). *NVIDIA*. "Unlock Next Level Performance with Virtual GPUs." Accessed on April 2, 2024, at: Link.

77 StackWatch. (n.d.). *StackWatch*. "NVIDIA Virtual Gpu Manager." Accessed on April 2, 2024, at: Link.

78 StackWatch. (n.d.). *StackWatch*. "NVIDIA Virtual Gpu." Accessed on April 2, 2024, at: Link.

79 Abid Shahzad and Alan Litchfield. (2015). *arXiv*. "Virtualization Technology: Cross-VM Cache Side Channel Attacks make it Vulnerable." Accessed on April 2, 2024, at: Link.

80 Microsoft Corporation. (July 14, 2020). *National Vulnerability Database*. "CVE-2020-1032 Detail." Accessed on April 2, 2024, at: Link.

81 Charles F. Goncalves, Xavier Mendes, and Marco Vieira. (n.d.). *Centre for Informatics and Systems of the University of Coimbra*. "Hypervisors Vulnerabilities Analysis: Causes, Effects and Consequences." Accessed on April 2, 2024, at: Link.

82 HiTechNectar. (n.d.). *HiTechNectar*. "An Overview of Hypervisor Vulnerabilities." Accessed on April 2, 2024, at: Link.

83 Yong Wang. (Aug. 2023). *Black Hat*. "Make KSMA Great Again: The Art of Rooting Android devices by GPU MMU features." Accessed on April 2, 2024, at: Link.

84 Cindy Casey. (n.d.). *Bucks County Community College*. "AI poisoning attacks." Accessed on April 2, 2024, at: Link.

85 Audra Simons. (Oct. 3, 2023). *Forcepoint*. "Data Poisoning: The Newest Threat to Generative AI." Accessed on April 2, 2024, at: Link.

86 Tyler Sorensen and Heidy Khlaaf. (Jan. 16, 2024). *Trail of Bits*. "LeftoverLocals: Listening to LLM responses through leaked GPU local memory." Accessed on April 2, 2024, at: Link.

87 Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. (March 20, 2015). *arXiv*. "Explaining and Harnessing Adversarial Examples." Accessed on April 2, 2024, at: Link.

88 Alex Polyakov. (Aug. 6, 2019). *Towards Data Science*. "How to attack Machine Learning (Evasion, Poisoning, Inference, Trojans, Backdoors)." Accessed on April 2, 2024, at: Link.

89 Akul Arora. (May 4, 2023). *University of California, Berkeley*. "AI Safety: Model Trojaning and Benchmarking." Accessed on April 2, 2024, at: Link.

90 Jie Wang, Ghulam Mubashar Hassan, and Naveed Akhtar. (n.d.). *arXiv*. "A Survey of Neural Trojan Attacks and Defenses in Deep Learning." Accessed on April 2, 2024, at: Link.

91 Tyler Sorensen and Heidy Khlaaf. (Jan. 16, 2024). *Trail of Bits*. "LeftoverLocals: Listening to LLM responses through leaked GPU local memory." Accessed on April 2, 2024, at: Link.

92 trailofbits. (n.d.). *GitHub*. "LeftoverLocals." Accessed on April 2, 2024, at: Link.

93 ROCm. (n.d.). *GitHub*. "AMD GPU device plugin for Kubernetes." Accessed on April 2, 2024, at: Link.

94 ggerganov. (n.d.). *GitHub*. "LLaMA.cpp HTTP Server." Accessed on April 2, 2024, at: Link.

95 Sharon Goldman. (Dec. 15, 2023). *VentureBeat*. "Why Anthropic and OpenAI are obsessed with securing LLM model weights." Accessed on April 2, 2024, at: Link.