Detecting Memory Editing Cheats by Validating Host Memory Integrity from GPU

Naoki Hashimoto*, Soramichi Akiyama
College of Information Science and Engineering, Ritsumeikan University, Osaka, Japan
Email: s-akym@fc.ritsumei.ac.jp

***Abstract*—As the financial size and impact of the video game industry grows, so does the impact of cheating in video games. For example, cheating in online multiplayer games could affect the fairness of their championships, which award tens of thousands of USD. A unique challenge in cheat detection for video games is that the owner of the system (on which the games run) are the adversaries, thus they have full control over the entire system. To achieve cheat detection against such adversaries, we propose a novel cheat detection system based on host memory monitoring from GPU. It monitors the value of protected objects and compares them with pre-loaded metadata, and detects cheats when they do not agree with each other. By monitoring the host memory by a GPU program that cannot be intercepted once invoked, it achieves resiliency against a cheater who has a kernel-level privilege of the system. Our evaluation shows that it can successfully find two types of cheats and that it is robust to a cheater's attempt to overwrite the metadata.**

***Index Terms*—Cheat detection, Memory-monitoring, Game security**

I. Introduction

The financial size and impact of the video game industry have been growing rapidly. Examples include the sales of Nintendo as a whole and Sony's game segment, which have grown by 78% and 352% in recent 10 years, respectively. As the financial impact of video games grows, so does the impact of cheating especially in online multiplayer games. Cheating in these games hurts the user experience of innocent players who would eventually stop buying them and negatively impacts the revenue of game developers. Cheating also reduces the fairness of championships of online multiplayer games that award tens of thousands of USD.

A unique challenge in cheat detection for video games is that the admin user (a.k.a. root user) of the system running the game is the adversary. It is different from many other security problems where the root user tries to prevent a malicious adversary (e.g., malware) from penetrating the security. This fact mandates a unique threat model, i.e., the adversary has kernel-level privilege and can use that privilege to prevent cheat detection systems from doing their jobs. For example, a video game player using Windows can use the task manager to freely stop cheat detection systems.

Although the adversary (person who cheats) has kernel-level privilege, a cheat detection system might not have the same privilege because installing such a system with high privilege has two concerns. First, it can be compromised by malware so that the privilege is abused. Indeed, a cheat detection system bundled with a famous video game was reported to be compromised to bypass anti-virus software. Second, the ability to monitor everything in the system raises privacy concerns. For example, a Windows driver that is supposed to be detecting cheats can instead be monitoring the communication between the PC and servers that are not relevant to the game itself.

To achieve cheat detection against such an adversary without leveraging a kernel-level privilege, we propose a novel cheat detection system based on host memory monitoring from GPU. The main idea is to utilize a **GPU** (not a CPU) to check the integrity of objects inside the host memory by comparing them with metadata registered at the allocation time. This makes our system robust against our adversaries because (1) a GPU program that is already running cannot be stopped (unless rebooting the entire machine or killing the host-side process that has launched it) even by such users, and (2) the metadata is copied to the GPU memory as soon as it is created to protect it from being overwritten. We implement a prototype of our system and show that it can detect when a protected value becomes out of the predefined range, and also when the value changes more rapidly than the predefined rate. We also evaluate the overhead and the resiliency of our system against a cheater who overwrites the metadata as fast as possible.

II. Background

A. Cheating in Video Games

The video game industry has been financially growing rapidly both in its size and impact. In the traditional gaming area, the net sales of Nintendo have grown from $6.76 million in 2013 [1] to $12.04 million in 2023 [2]. Similarly, the sales of the gaming segment of Sony have grown from 805.0 million JPY in 2012 [3] to 3,644.6 million JPY in 2022 [4]. Also in the competitive gaming area (or *e-sports*), multiple championships award tens of thousands of dollars [5] or even millions of dollars [6], and the top-notch players earn hundreds of thousands of dollars from these awards [7].

Cheating in video games has three major impact especially in online multiplayer games.

1) **User experience**: when one player cheats, the other players would have bad feelings by either observing it or being beaten in the game regardless of their best effort. This could in turn reduce the sales of the game because these innocent players might no longer want to play.

*Currently in industry. Work done while in Ritsumeikan University.

TABLE I
CHEATING CATEGORIZED BY THE MEASURES

Cheat Type	Measure	User burden	Target
User-level	Software (Process)	Small	
Kernel-level	Software (Driver)	Moderate	✓
Device-level	Hardware (DMA)	Very Large	

2) **Fairness**: championships of some online multiplayer games today often award tons of money and some players base their lives on these awards. Thus, reducing the fairness of championships with cheating could even hurt the real lives of the participants. Because of the huge impact, game developers are now serious about finding and preventing cheating in their championships. For example, Riot Games banned a team from its championships after a member of that team cheated [8].
3) **Real money trade (RMT)**: cheaters make money by selling rare items or highly grown-up characters acquired by the use of cheating tools. RMT not only hurts the user experience of innocent players but encourages them to also join it because it is sometimes way easier to buy a rare item than actually acquire it with a legitimate play.

B. Target Cheat Types for Detection

In this paper, we set *kernel-level memory editing cheats* as the target for detection. This section explains what they are and how they are implemented in detail.

A memory editing cheat is a user's activity to illegitimately modify objects in the heap of the game process with the help of some software or hardware tool. For example, a user could overwrite the attack parameter contained in an enemy object to -1. This results in a hit by the enemy healing (in stead of damaging) the user's character, which illegitimately helps the user clearing the game with little effort.

Memory editing cheats can be implemented in various ways as shown in Table I.

1) **User-level**: A user-level process is used. In Windows, a process can be given the *Debug Privilege* that enables it to affect other processes in a variety of ways. This includes reading from and writing to their memory contents or creating new threads in their contexts. Because a cheater often has full control over the system on which games run, they can easily grant the Debug Privilege to a cheating process.
2) **Kernel-level**: A kernel-level driver is used. The word *kernel* refers to OS kernels but not GPU kernels throughout the paper, unless otherwise specified. Because a driver runs in the highest privilege level, it can easily read from and write to the memory content of game processes. Even worse, nothing can prevent a cheating driver from being installed because the cheater has full control of the system.
3) **Device-level**: A physical device is used. A device attached to a PCIe slot reads from and writes to physical memory directly through Direct Memory Access.

The reason why we target the kernel-level cheats is that it is hard to prevent but still easy to conduct. User-level cheats are easier to prevent than kernel-level cheats because they are conducted by user-level processes. Device-level cheats are the most powerful in terms of the stealthiness, while the hurdle for introducing them is quite high. Kernel-level cheats are as easy to install as user-level cheats, and they are harder to block because of the high privilege level.

C. Challenges in Mitigating Kernel-level Cheats

Detecting and preventing the cheats that we target in this paper is challenging in two aspects. **First**, anti-cheat tools that work in the user-level cannot prevent them by principle. Here, an anti-cheat tool is software that is either installed separately or bundled with video games and investigates if a player is cheating. Because a user-level anti-cheat tool is only granted a user-level privilege, a cheater that leverages a kernel-level driver can easily control (e.g., terminate) the anti-cheat tool.

Second, kernel-level anti-cheat tools (e.g., Vanguard adopted by a popular game named Valorant) have two major concerns due to their high privilege even though they are powerful enough to find the target cheats.

1) **Increased attack surface**: These tools are targeted by malware because of their high privilege level. For example, the anti-cheat tool of Genshin Impact was compromised to terminate antivirus software on the system so that malware can be freely executed afterward [9].
2) **Privacy concern**: They can monitor everything that happens in the user-space and can send the monitored information to the Internet. This along with the fact that the source code of anti-cheat tools are disclosed has raised privacy concerns [10], [11].

III. RELATED WORK

A powerful countermeasure against cheating is Trusted Execution Environments (TEEs), where even the OS is not granted accesses to the memory of protected processes. BlackMirror [12] proposes a new game design to utilize Intel SGX for preventing wallhacks. TZMon [13] utilizes ARM TrustZone to protect mobile games, as well as to securely update the games and synchronize clocks. Although very powerful, a drawback of TEEs is that they require games to be specifically coded to use them. On the other hand, protecting objects with our system only requires replacing memory allocation code and nothing else needs to be changed.

Monitoring host memory from a GPU is proposed for health-checking the host OS. GPUSentinel [14] monitors the memory of the host OS from a GPU to detect kernel-level faults such as infinite loops with interrupts disabled. GPUfas [15] overwrites the memory content of the host OS to recover from such faults. Our work differs from these systems in its threat model and the system design. The root user and the host OS in these systems are honest although faulty, while the adversary is the root user themselves in our threat model.

Utilizing GPUs for security has been proposed in other systems as well. Raspberry Pi leverages the integrated GPU

for booting by letting it execute the first stage bootloader [16], [17]. This hinders reverse-engineering attempts on the bootloader because it is provided as an undocumented GPU binary. PixelVault [18] achieves secure computation with an assumption that even the OS is compromised. It hides secret keys and critical code that uses the keys inside the registers and cache of the GPU, allowing no read or write accesses from the compromised host.

IV. Proposal: Cheat Detection by GPU

A. Main Idea

To detect kernel-level memory editing cheats in video games while addressing the challenges of existing methods, we propose a system that does not require a kernel-level privilege but still be resilient to interception from a cheater who has a kernel-level privilege. Fig. 1 shows the overview of our proposal. The main idea is to monitor the content of the host memory from a GPU task to check the integrity of in-game objects by comparing them against the metadata stored at the allocation time. Note that a *GPU task* in this paper means a GPU kernel in the well-established terminology, but we use the word *task* to avoid confusion between a GPU kernel and an OS kernel throughout this paper.

To make our system resilient to a cheater who has a kernel-level privilege without granting ourselves the same privilege, we leverage two characteristics of GPU tasks. We explain characteristic (2) in detail later in Section IV-C.

1) GPU tasks can be invoked with a user-level privilege. This allows us to *not* have a kernel-level privilege.
2) Even someone with a kernel-level privilege cannot stop a currently running GPU task due to a lack of such APIs. This makes our system *not* interceptable by a cheater with a kernel-level privilege.

B. Threat Model and Target Cheats

We assume the following threat model. Please note that it defines what we cover in this paper, and we are aware that it does not represent every cheater or every system.

1) The cheater plays a video game that requires an NVIDIA GPU for seamlessly playing it.
2) The cheater has full control of the computer on which they play the target game *in theory*. However, *in practice*, the cheater does not have full knowledge nor motivation to enable themselves to do more than installing off-the-shelf cheating tools.
3) The cheater conducts kernel-level cheating that modifies objects inside the heap of the process of the target game. The cheater does not conduct device-level cheating due to a lack of knowledge and motivation.
4) The proposed system is properly invoked when the target game starts its execution. This must be ensured with a technique outside of this paper.
5) The program code (both as forms of executable files and in-memory binaries) is protected from any overwriting attempts by the cheater. This must also be ensured with a technique outside of this paper.

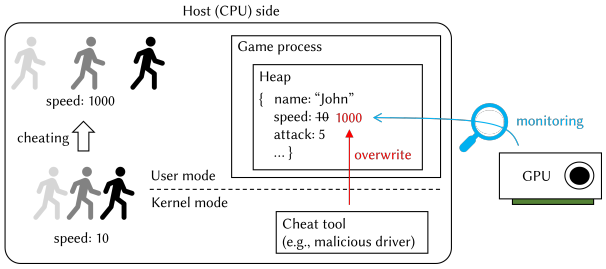

Fig. 1. Overview of Our Proposal

Under this threat model, we assume that the cheater conducts kernel-level memory editing cheats. In concrete, we assume that a cheater modifies the memory content in the following two ways.

1) **Out-of-Range (OR)**: a value inside an object is overwritten with another value that is not supposed to appear during a legitimate play. Examples include setting the gravity to 0, resulting in a character floating all the time.
2) **Too-Drastic (TD)**: a value inside an object is updated at a rate that is not supposed to appear during a legitimate play. Examples include updating the position of a character from one edge of the map to another within one frame.

Please note that these do not represent every cheat, but they are what we cover in this paper. For example, we do not cover a case where a kernel-level cheating tool overwrites the process ID of the game process.

C. Continuity of GPU tasks

Our system invokes a GPU task that monitors the content of the host memory. We explain how this design makes our system resilient to a cheater who has a kernel-level privilege.

A GPU task cannot be stopped its execution once it has been started unless the entire machine is shut down or rebooted, or the host-side process that has launched the GPU task is terminated. This is because the CUDA APIs provided by NVIDIA do not support quitting a GPU task that is currently running. Table II shows all the CUDA APIs categorized as Execution Control [19]. Although there is `cudaLaunchKernel` and `cudaLaunchDevice` that launch GPU tasks, no API that stops a running GPU task exists.

Besides the APIs in Table II, the closest we are aware of that might be able to stop a currently running GPU task is `cudaDeviceReset`. The documentation [20] says that it *destroy all allocations and reset all state on the current device in the current process*. To confirm how it works, we created a GPU task that printed a message at every 100,000,000$^{\text{th}}$ iteration of a loop. We used the `<<<` and `>>>` syntax to invoke this GPU task, which is a high-level syntax sugar for `cudaLaunchDevice`. Five seconds after this task had been kicked off, we called `cudaDeviceReset` and `cudaDeviceSynchronize` in this order. The result was that the message kept being printed even after we called

TABLE II
CUDA APIs CATEGORIZED AS EXECUTION CONTROL [19]

API Name	Description
cudaFuncGetAttributes	Find out attributes for a given function.
cudaFuncGetName	Returns the function name for a device entry function pointer.
cudaFuncSetAttribute	Set attributes for a given function.
cudaFuncSetCacheConfig	Sets the preferred cache configuration for a device function.
cudaFuncSetSharedMemConfig	Sets the shared memory configuration for a device function.
cudaGetParameterBuffer	Obtains a parameter buffer.
cudaGridDependencySynchronize	Programmatic grid dependency synchronization.
cudaLaunchCooperativeKernel	Launches a device function where thread blocks can cooperate and synchronize as they execute.
cudaLaunchCooperativeKernelMultiDevice	Launches device functions on multiple devices where thread blocks can cooperate and synchronize as they execute.
cudaLaunchDevice	Launches a specified kernel.
cudaLaunchHostFunc	Enqueues a host function call in a stream.
cudaLaunchKernel	Launches a device function.
cudaLaunchKernelExC	Launches a CUDA function with launch-time configuration.
cudaSetDoubleForDevice	Converts a double argument to be executed on a device.
cudaSetDoubleForHost	Converts a double argument after execution on a device.
cudaTriggerProgrammaticLaunchCompletion	Programmatic dependency trigger.

`cudeDeviceReset`, meaning that this API did *not* stop the already-running GPU task.

Terminating the host-side process that has launched a GPU task does not work for bothering our system. This is because the "host-side process" in our case is the video game itself, and terminating it loses the whole purpose of a cheater (i.e., playing that game). The detailed architecture of our system is presented in Section V.

D. Monitoring Host Memory

We create mapped memory regions between the GPU and the host to monitor the host memory. This functionality allows a GPU to read from and write to a region allocated in the host memory without any involvement of the host-side CPU through direct memory access (DMA). On an NVIDIA GPU, the `cudaHostRegister` API maps a given memory region between the host and the GPU, as well as *page-locks* that region so that it is never swapped out from the main memory.

The use of mapped memory makes our system resilient from interception by a cheater. Because a mapped region is accessed through DMA, no host-side instructions are needed to be executed to access it. This means that the cheater has no way to interrupt or cease the communication between our system and the host memory. On the other hand, a non-DMA method of communication with a GPU requires the host to execute the `cudaMemcpy` API. A cheater could hook calls of this API (for example by attaching a debugger to the game process) to intercept our system. Note that `cudaHostRegister` to create mapped memory regions is called only at the beginning of the game process (explained in Section V), while `cudaMemcpy` would be called every time our system monitors the host memory if we were to use it.

V. OUR CHEAT DETECTION SYSTEM

A. System Components

Fig. 2 shows the components of our system and how the communicate with each other to detect cheats.

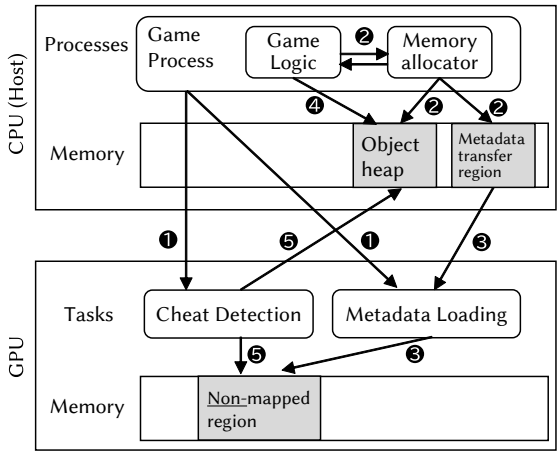

Fig. 2. System Components and Communication

❶ In the beginning, the game process invokes two GPU tasks, the *cheat detection task* and the *metadata loading task*. They run asynchronously from the game process and never finish by themselves, unlike an ordinary GPU kernel that finishes after some work. At the same time, the game process allocates two memory regions, the *object heap* and the *metadata transfer region*, and maps them with GPU memory using `cudaHostRegister`. We assume that everything until this point is executed properly without interception by a cheater.

❷ The game logic uses a designated memory allocator to store an object that needs to be protected. The allocator chops the object heap to the specified size and returns the pointer to that region to the game logic. The allocator also stores the metadata of the object in the metadata transfer region. The details of the metadata are described in Section V-B.

❸ The metadata loading task periodically copies the metadata from the metadata transfer region to a GPU memory region that is **not** mapped with the host memory.

❹ The game logic can freely write to and read from objects allocated by our memory allocator as if they were allocated

```
void* mallocWithRegisterToHDIT(
  int data_size, int limit_type,
  int min_parameter, int max_parameter,
  int max_change_rate);
```

Fig. 3. Allocation Function of Our Memory Allocator

by normal `malloc`.

❺ The cheat detection task monitors the object heap and detects cheats by checking the consistency of each object and its metadata. This will further be explained in Section V-D.

B. Memory Allocator

The memory allocator abstracts the necessary procedures to protect an object away from video game developers (i.e., the user of our system). Fig. 3 shows the allocation function of our memory allocator. The user specifies the size of the object to be allocated to `data_size`. The other arguments are stored as the metadata of the object, which we refer to as the *Heap Data Information (HDI)* of the object. An object and its HDI are associated with the same ID number.The other arguments are explained in Section V-D.

We design the memory allocator so that it does not ask the OS to allocate a new region, but it merely chops a part of the memory region that has already been allocated at the start of the game process. This design reduces the usage of the CUDA API that maps host- and GPU-side memory regions only once. It means that a cheater has no chance to intercept CUDA API calls by means such as attaching a debugger after the mapped region is created in the beginning. A possible attack in this design is that a cheater can overwrite the binary code of the game program so that it does not use our memory allocator. We discuss a possible countermeasure in Section VII-B.

C. Metadata Loading Task

The metadata loading task periodically copies HDI (metadata of protected objects) stored by the memory allocator from the metadata transfer region to a GPU memory region. Each HDI has a unique and increasing integer as its ID. HDI is not copied if the ID is less than the ones that are already copied.

The design choice of copying HDI from the metadata transfer region to a GPU memory region (instead of having the cheat detection task directly check HDI through DMA) is to improve the resilience of our system. Because the GPU memory region is not shared with the host-side, the HDI cannot be overwritten from the host-side once copied. This allows a cheater to have a small amount of time to maliciously overwrite HDI to trick our system. We evaluate the resiliency of our system against this type of cheaters in Section VI-E.

D. Cheat Detection Task

The cheat detection task periodically checks the HDI of each object to detect cheats on that object. Specifically, our system detects cheats using two methods.

1) **Range-based**: This method is used to detect the Out-of-Range (OR) cheats. It checks if the value of the object is contained within the range between `min_parameter` and `max_parameter` in the HDI. If this condition is not met, it is recognized as a cheat.
2) **Diff-based**: This method is used to detect the Too-Drastic (TR) cheats. It checks if the difference of the object values between the current and the previous frames is less than `max_change_rate` in the HDI. We assume that a frame is a fixed amount of time (e.g., $\frac{1}{60}$ sec.), and use the elapsed time from the start of the task to recognize a new frame. The elapsed time can be acquired independently from the host OS by dividing the number of spent GPU clocks by the clock rate.

```
class Player {
  int HP;
  char *name;
  int *attack; // originally 'int attack'

  Player(int H, char *n, int a) {
    this.HP = H;
    this.name = n;
    this.attack
      = mallocWithRegisterToHDIT(
          sizeof(int),
          1,  // limit_type
          0,  // min_parameter
          10, // max_parameter
          0); // not relevant for limit_type=1
    *this.attack = a;
  }
};
```

Fig. 4. Usage of Our System to Find OR Cheats by Range-based Method

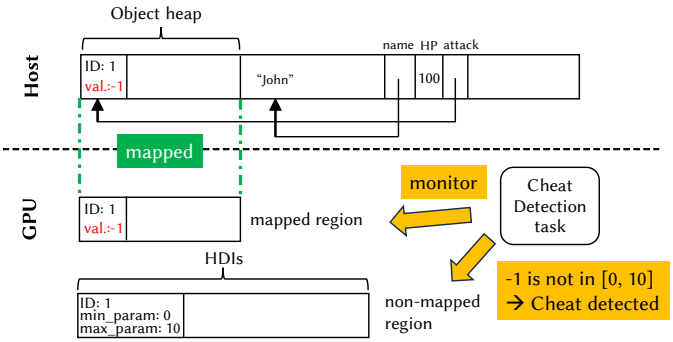

Fig. 5. Detection of OR Cheats in Action

Fig. 4 shows an usage of our system, where a member `attack` of a class `Player` is protected. The `limit_type` parameter decides which method (Range-based or Diff-based) is applied to detect cheats on this object. In the figure, it is set to 1 so that the Range-based method is applied. To use our memory allocator, the programmer needs to apply small changes to the game logic. Specifically, the type of `attack` is changed from `int` to `int*` and the code to store value to `attack` is modified to apply pointer de-reference.

Fig. 5 shows an instance of the OR cheating being detected for a class defined by Fig. 4. In this scenario, the value of `attack` is overwritten to -1 by the cheater. The cheat detection task running in the GPU periodically compares the

TABLE III
EVALUATION ENVIRONMENT

OS	Windows 10
GPU	NVIDIA GeForce GTX1070 (GDDR5 8GB)
CPU	AMD Ryzen 5 3500 (6 cores)

value with the metadata, and finds that the current value (-1) is not contained in the predefined range (0 to 10), and thus determines that the OR cheating is conducted.

VI. EVALUATION

A. Research Questions

Our evaluation answers three research questions shown below. The evaluation environment is listed in Table III.

1) Can our system indeed detect the OR and TD cheating?
2) How frequently can our system detect cheating?
3) How large is the overhead that our system incurs?
4) How resilient is our system to a cheater who attacks the metadata transfer region?

B. Cheat Detection Experiment

In this experiment, we test that our system can detect both the OR and TD cheating as follows. Note that we do not violate any end-user license agreements or laws because we overwrite the memory content of programs that we develop on our own physical computer.

OR: The victim is mimicked by a simple program that stores a single integer in a memory region allocated using our system. The metadata is set so that the integer must be more than or equal to 0, and less than or equal to 100. We use Cheat Engine [21] to overwrite the integer value to 101 from outside of the victim's process.

TD: The victim is again mimicked by a simple program that stores a single integer to a memory region allocated using our system. The metadata is set so that the maximum allowed change of the integer per frame is 100. We use Cheat Engine [21] to overwrite the integer value from 0 to 101. This is equivalent to a change of *101 per frame*.

Results: A warning message by the cheat detection task was printed to the console. This means that our system could successfully detect the target cheats.

C. Frequency of Cheat Detection

In this experiment, we evaluate the possible interval at which our system can detect cheating when the number of protected objects increases. A shorter interval means more resiliency of our system against short-term cheating such as switching the value of a compromised object between legit and illegal ones every frame.

To evaluate the interval for different numbers of objects, we swipe the number of integers protected by our system. The metadata is set so that the same cheat detection is conducted as in the OR cheat detection experiment. The cheat detection task is modified to call the `Clock64` function at the beginning of the OR cheat detection logic. This function returns a value that

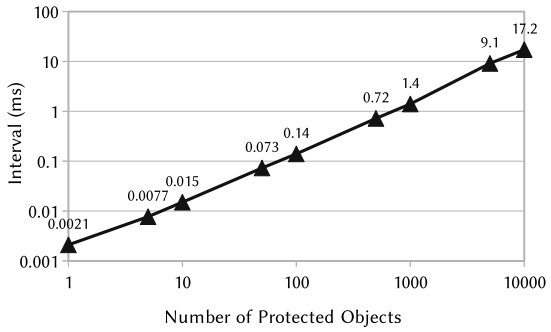

Fig. 6. Cheat Detection Intervals for Different Number of Objects

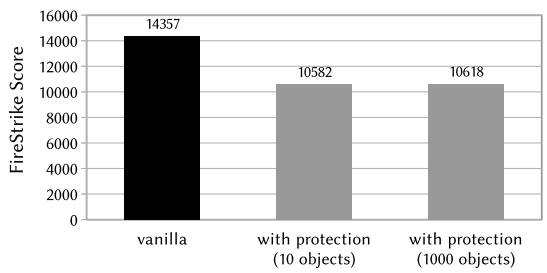

Fig. 7. Overhead Incurred to the Fire Strike benchmark

increments every clock cycle, and thus we can measure elapsed time by comparing the return values among two invocations.

Fig. 6 shows the result of our experiments. Note that both x- and y-axes are log-scaled. We draw two takeaways from the results. First, the interval grows almost linearly to the number of protected objects. This is expected because the number of memory accesses to the object heap and the number of times that cheat detection logic is executed increase linearly to the number of objects. Second, the interval exceeds $\frac{1}{60}$ seconds ($\approx$ 16.7 ms) when the number of objects is 10,000. This means that the number of objects must be less than this amount if the programmer wants to check object integrity at every frame.

D. Performance Overhead

We evaluate the overhead incurred to the video game that our system shares the GPU with. To this end, we use Fire Strike benchmark included in the 3DMark benchmark suite, which is *a showcase DirectX 11 benchmark for modern gaming PCs* [22]. We launch Fire Strike benchmark along with a simple program that uses our system to protect its objects, and measure how much the Fire Strike performance drops.

Fig. 7 shows the results of the overhead evaluation. The label *vanilla* shows the performance when our system is not co-located, and the labels *with protection (N objects)* show performance when the co-located program allocates N integers with our system. We draw two takeaways from the results. First, the performance overhead is around 26%, which is not negligible. We suspect that this overhead mainly comes from memory bandwidth congestion caused by the GPU monitoring the host memory as fast as possible. The GPU we use has 15

TABLE IV
RESILIENCY OF METADATA TRANSFER

# of Objects	10,000
Transfer Successful Rate	6.3%
Chance of 25 HDI Not Successfully Transferred	19.7 %

Streaming Multiprocessor and occupying two of them for the cheat detection and metadata loading tasks would result only in $\frac{2}{15} \times 100 \approx 13$ % of overhead. The fact that the actual overhead is reasonably larger than this suggests that the bottleneck is not the computation, but memory bandwidth. Evaluating our system with GPUs with larger memory bandwidth is a part of future work. Second, the performance overhead has no significant difference even when the number of protected objects is increased by 100× (from 10 to 1,000). This result also suggests that the overhead comes from memory transfer. Note that the current system monitors the host memory as fast as possible, thus the memory congestion level is constant regardless of of number of protected objects.

E. Resiliency of Metadata Transfer

We evaluate the resiliency of our system against a cheater who overwrites HDI stored in the metadata transfer region. The metadata loading task copies metadata to GPU memory as fast as possible to cope with this. However, it might still be possible to intercept the metadata loading task if the cheater overwrites the metadata transfer region at the right time.

We mimic a cheater by creating a malicious thread in the context of a simple target program that allocates 10,000 objects with our system. The malicious thread overwrites legitimate HDI in the metadata transfer region to set their IDs to 0, which makes the metadata loading task not copy the HDI. The malicious thread is created by `CreateThread`.

Table IV shows the result of the resiliency evaluation. The numbers are averaged over 10 runs. Among the 10,000 HDI, 6.3% were transferred from the metadata transfer region to GPU memory on average. This means that if an attacker needs to modify N protected objects to conduct a single cheat, they must win a $((1-0.063)^N \times 100)\%$ chance of not being detected. This is because only a single protected object detected as maliciously modified proves that a cheat has been attempted. For example, when $N = 25$, the attacker must be as lucky as winning a 19.7 % chance.

VII. DISCUSSION

A. Applicability to Non-NVIDIA GPUs

We discuss the applicability of our system to non-NVIDIA GPUs. A GPU must meet the following conditions:

1) Shared memory regions between a GPU and the host-side CPU can be created.
2) A shared memory region can be accessed by a GPU through DMA without any host involvement. Combined with 1), this allows our system to copy HDI and monitor protected objects with no interception from a cheater.
3) Tasks running in a GPU cannot be intercepted by the host. This allows our system to keep running even when a cheater tries to stop it with a kernel-level privilege.

AMD GPUs meet all the conditions. First, their Pinned memory *is host memory that is mapped into the address space of all GPUs* [23] and can be accessed via DMA. Second, the Heterogeneous-Compute Interface for Portability (HIP), which is the standard programming interface for AMD GPUs, does not have an API that stops an already-running GPU task. The list of all the HIP APIs is listed on the official website [24] (we omit the list here for brevity).

Intel's discrete GPUs meet all the conditions. First, their Unified Shared Memory (USM) is *visible to both host and device(s)* [25], and the *GPU can perform DMA (Direct Memory Access) over PCI to system memory* [26]. Second, their programming interface (SYCL) neither has an API that stops an already-running GPU task. In this programming model, a programmer enqueues a task using `Queue`. For example, `Queue::submit` accepts a callable object to be executed on a GPU. Although it supports multiple ways of launching GPU tasks, it does not have any API for dequeuing a task [25].

Intel's integrated GPUs do not support our system as-is because the GPU memory is physically shared with the CPU [27]. Therefore, a cheater who has full control over the host OS can overwrite GPU-side HDI at any time. However, these GPUs also have a small size of local memory (64 KB per 8 execution units). Our system could be extended to utilize this local memory to store HDI copied from the main memory.

B. Protection of Program Code

We assume that program code is protected from any overwriting attempts, regardless of the fact that a cheater has full control over the host OS. If this does not hold, the cheater can overwrite the game logic so that it does not use the memory allocator provided by our system to bypass it. This protection must be done in two phases: (1) the executable and related files of the game and our system must not be overwritten before they are loaded to host memory (*offline protection*), and (2) the in-memory binaries of the game and our system must not be overwritten at runtime (*runtime protection*).

One way of achieving offline protection is to utilize packers. A packer converts an executable so that it cannot be disassembled unless the conversion algorithm and/or the encryption key used for conversion is known. Although packers are often used by malware to evade static analysis [28], [29], they can also be used to protect the program code of legitimate software (e.g., video games) from cheaters' overwriting attempts.

One way of achieving runtime protection is Virtualization Based Security (VBS) [30] available in recent Windows. VBS contains the user-visible OS in a virtual machine (VM), and a sidecar VM ensures some security aspects even if the user-visible OS is fully controlled by an adversary. One such aspect is memory integrity, where *executable pages themselves are never writable* [31]. Because this is ensured by the hypervisor, a cheater cannot overwrite the in-memory binaries of the game and our system unless they completely disable VBS.

C. Attacks to Metadata Transfer Region

A possible attack on the metadata transfer region besides the one described before is to create many fake HDIs. This would prolong the cheat detection interval as the number of HDIs to check is increased. Our system can handle this attack as follows. (1) If the attacker only creates a fake HDI and no corresponding fake data in the object heap, it is detected as a cheat because the fake HDI and data in the object heap do not agree with each other. (2) If the attacker also creates fake data in the object heap, the number of fake objects is limited by the size of the object heap. Because the size of the object heap must be configured so that the cheat detection interval is short enough even if it is full, filling it up with fake data would not increase the cheat detection interval to an unexpected length.

D. Attacks to Object IDs

An attacker can overwrite IDs assigned to protected objects stored in the host memory. This both includes (a) IDs of already existing objects, and (b) the ID that is to be assigned next time a new protected object is allocated. This makes three types of attacks possible; (1) the next ID to be assigned can be overwritten either to a random number or an already used one, (2) the ID of an already allocated object can be overwritten, and (3) the IDs of two already allocated objects can be swapped (e.g., changing object A's ID from 1 to 2 and changing object B's ID from 2 to 1).

Attacks (1) and (2) do not evade our system, while attack (3) could. For attack (1), a new object with a random ID can be properly monitored because there is no constraint on the IDs except the uniqueness. A new object with an already assigned ID can be detected as an attack because it should never happen. For attack (2), it can be recognized as an attack because the ID of the corresponding object in the GPU memory remains unchanged and the mismatch between an object and the HDI implies a cheat. For attack (3), our system can be evaded if the two objects are carefully selected by an attacker.

VIII. Conclusion

In this paper, we proposed a cheat detection system that utilizes a GPU. Our system does not need to have a kernel-level privilege but is still resilient against a cheater who has full control over the OS of the machine. Our evaluation shows that it can find two types of cheating (out-of-range and too-drastic), and the metadata required to detect cheating can be securely transferred with a 6.3% chance. We also gave extended discussions on our system, such as its applicability to non-NVIDIA GPUs and the limitations.

References

[1] Nintendo, "Annual Report 2013," https://www.nintendo.co.jp/ir/pdf/2013/annual1303e.pdf, 2020.

[2] Nintendo, "Annual Report 2023," https://www.nintendo.co.jp/ir/pdf/2023/annual2303e.pdf, 2020.

[3] Sony, "Consolidated Financial Results for the Fiscal Year Ended March 31, 2013," https://www.sony.com/en/SonyInfo/IR/library/presen/er/12q4_sony.pdf, 2013.

[4] Sony, "FY2022 Consolidated Financial Results (Fiscal year ended March 31, 2023)," https://www.sony.com/en/SonyInfo/IR/library/presen/er/pdf/22q4_sonypre.pdf, 2023.

[5] Liquipedia VALORANT Wiki, "VALORANT Champions Tour 2023," https://liquipedia.net/valorant/VCT/2023, 2023.

[6] CAPCOM CPU X, "The Home of Street Fighters Esports," https://sf.esports.capcom.com/ccx/en/, 2023.

[7] Dexerto, "Top 20 highest earning Valorant pros of all time," https://www.dexerto.com/valorant/top-20-highest-earning-valorant-pros-all-time-list-1408517/, 2023.

[8] VALORANT Esports NA, "Competitive ruling / updates," https://x.com/valesports_na/status/1710390686632325498, 2023.

[9] TREND Micro, "Ransomware Actor Abuses Genshin Impact Anti-Cheat Driver to Kill Antivirus," https://www.trendmicro.com/en_hk/research/22/h/ransomware-actor-abuses-genshin-impact-anti-cheat-driver-to-kill-antivirus.html, 2022.

[10] A. Gjonbalaj, J. Chen, D. Demicco, and A. Prakash, "Cheating in esports: Problems and challenges," in *IEEE Gaming, Entertainment, and Media Conference (GEM)*, 2023, pp. 1–6.

[11] C. Dorner and L. D. Klausner, "If it looks like a rootkit and deceives like a rootkit: A critical examination of kernel-level anti-cheat systems," in *International Conference on Availability, Reliability and Security (ARES)*, 2024, pp. 1 – 11.

[12] S. Park, A. Ahmad, and B. Lee, "BlackMirror: Preventing wallhacks in 3D online FPS games," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[13] S. Jeon and H. K. Kim, "TZMon: Improving mobile game security with ARM trustzone," *Computers & Security*, vol. 109, pp. 102 391 – 102 417, 2021.

[14] Y. Ozaki, S. Kanamoto, H. Yamamoto, and K. Kourai, "Reliable and accurate fault detection with GPGPUs and LLVM," in *International Conference on Cloud Computing (CLOUD)*, 2023, pp. 540 – 546.

[15] K. Kimura and K. Kourai, "GPU-based first aid for system faults," in *Asia-Pacific Workshop on Systems (APSys)*, 2022, pp. 38 – 45.

[16] W. W. Gay, *Raspberry Pi System Software Reference*. Apress, 2014.

[17] P. Francis-Mezger and V. M. Weaver, "A raspberry pi operating system for exploring advanced memory system concepts," in *International Symposium on Memory Systems (MEMSYS)*, 2018, pp. 354 – 364.

[18] G. Vasiliadis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Pixelvault: Using gpus for securing cryptographic operations," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014, pp. 1131 – 1142.

[19] NVIDIA, "CUDA Toolkit Documentation 6.8. Execution Control," https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EXECUTION.html.

[20] NVIDIA, "CUDA Toolkit Documentation 6.1. Device Management," https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__DEVICE.html.

[21] Cheat Engine, "Cheat Engine," https://www.cheatengine.org/, 2023.

[22] UL Solutions, "3DMark benchmark for Windows, Android and iOS," https://benchmarks.ul.com/3dmark, 2023.

[23] AMD, "GPU Memory - ROCm Documenitation," https://rocm.docs.amd.com/en/latest/conceptual/gpu-memory.html, 2024.

[24] AMD, "HIP Runtime API Reference: Globals - HIP 6.1.0 Documenitation," https://rocm.docs.amd.com/projects/HIP/en/latest/doxygen/html/globals_func_h.html, 2024.

[25] M. Rovatsou, "SYCL 2020 Specification (revision 8)," https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html, 2023.

[26] Intel, "oneAPI GPU Optimization Guide," https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2024-0/multi-stack-gpu-architecture.html, 2024.

[27] M. Dashti and A. Fedorova, "Analyzing memory management methods on integrated CPU-GPU systems," in *International Symposium on Memory Management (ISMM)*, 2017, pp. 59–69.

[28] H. Holm and E. Hyllienmark, "Hide my payload: An empirical study of antimalware evasion tools," in *IEEE International Conference on Big Data (BigData)*, 2023, pp. 2989–2998.

[29] A. Ruggia, D. Nisi, S. Dambra, A. Merlo, D. Balzarotti, and S. Aonzo, "Unmasking the veiled: A comprehensive analysis of Android evasive malware," in *ACM Asia conference on Computer and Communications Security (ASIACCS)*, 2024, pp. 1 – 16.

[30] A. Allievi, M. E. Russinovich, A. Ionescu, and D. A. Solomon, *Windows Internals, Part 2, 7th Edition*. Microsoft Press, 2021.

[31] Microsoft Learn, "Memory integrity and VBS enablement," https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-hvci-enablement, 2023.