Efficient Buffer Overflow Detection on GPU

Bang Di, Jianhua Sun, Hao Chen, *Member, IEEE*, and Dong Li

Abstract—Rich thread-level parallelism of GPU has motivated co-running GPU kernels on a single GPU. However, when GPU kernels co-run, it is possible that one kernel can leverage buffer overflow to attack another kernel running on the same GPU. There is very limited work aiming to detect buffer overflow for GPU. Existing work has either large performance overhead or limited capability in detecting buffer overflow. In this article, we introduce GMODx, a runtime software system that can detect GPU buffer overflow. GMODx performs always-on monitoring on allocated memory based on a canary-based design. *First*, for the fine-grained memory management, GMODx introduces a set of byte arrays to store buffer information for overflow detection. Techniques, such as lock-free accesses to the byte arrays, delayed memory free, efficient memory reallocation, and garbage collection for the byte arrays, are proposed to achieve high performance. *Second*, for the coarse-grained memory management, GMODx utilizes unified memory to delegate the always-on monitoring to the CPU. To reduce performance overhead, we propose several techniques, including customized list data structure and specific optimizations against the unified memory. For micro-benchmarking, our experiments show that GMODx is capable of detecting buffer overflow for the fine-grained memory management without performance loss, and that it incurs small runtime overhead (4.2 percent on average and up to 9.7 percent) for the coarse-grained memory management. For real workloads, we deploy GMODx on the TensorFlow framework, it only causes 0.8 percent overhead on average (up to 1.8 percent).

Index Terms—Buffer overflows, CUDA, GPGPU, unified memory

1 Introduction

Graphics processing units (GPUs) are widely adopted in HPC and cloud computing platforms to accelerate general-purpose workloads. For example, GPUs can achieve significant speedup for graph processing applications [1], and GPU-assisted network traffic processing in software routers outperforms the multicore-based counterparts [2]. In addition, GPUs can also be used to accelerate big-data processing [3], [4], [5], [6], and assist operating systems [7], [8], [9], [10], [11], [12] as a buffer cache. Rich thread-level parallelism in GPU has motivated co-running GPU kernels on a single GPU. Computation-intensive algorithms, such as AES [13], mathematical modelling [14], and biological applications [15], have also been successfully ported to GPU platforms. However, co-running GPU kernels poses a big challenge on how to guarantee strong isolation between different kernels, when those kernels are used together without any protection. It is possible that a kernel can leverage buffer overflow to attack another kernel running on the same GPU [16], [17].

Despite extensive research over the past few decades, buffer overflow remains one of the top software vulnerabilities. Many notorious attacks, such as Code Red [18], Morris Worm [19], and Slammer [20], exploit buffer overflow and result in program crash, data corruption, and security breaches. Those attacks overwrite memory buffer allocated by applications at runtime on CPU. The buffer overflow also exists on GPU as introduced in recent studies [16], [17]. These studies demonstrate that buffer overflow on GPU can lead to remote GPU code execution when dynamically allocated memory is operated improperly.

However, detecting buffer overflow on GPU is nontrivival due to the execution model and architecture of GPU. *First*, a GPU kernel easily has a large number of threads, each of which can dynamically allocate memory buffers. Hence a GPU kernel can potentially have a large number of memory buffers. Given these buffers, we must have a systematic and scalable approach to collecting buffer information for overflow detection. The approach should minimize the usage of GPU resources, such as hardware threads, so that GPU resources can be effectively used for the computation of regular GPU kernels. *Second*, GPU lacks certain system support available on CPU, such as page protection and preemptive execution. Traditional security mechanisms, such as Electric Fence [21] and StackGuard [22] based on system support on CPU, cannot work for GPU. *Third*, detecting buffer overflow at runtime must have minimum impact on the performance of GPU kernels. We should avoid adding extra functionality into GPU kernels for detecting buffer overflow. In addition, given that GPU kernels and overflow detection kernel execute concurrently, we should avoid data races between them, without frequently interrupting the kernels execution.

Unfortunately, there is very limited work [23], [24], [25] aiming to detect buffer overflow for GPU. The existing work has either large performance overhead or limited protection. For instance, cuda-memcheck [23] can identify the source and cause of memory access errors in GPU code based on intensive instrumentation. However, cuda-memcheck is for

- *Bang Di, Hao Chen, and Jianhua Sun are with the College of Computer Science and Electrical Engineering, Hunan University, Changsha 410082, China. E-mail: {dibang, jhsun, haochen}@hnu.edu.cn.*
- *Dong Li is with the Department of Electrical Engineering and Computer Science, University of California Merced, Merced, CA 95343 USA. E-mail: dli35@ucmerced.edu.*

Manuscript received 27 Mar. 2020; revised 21 Nov. 2020; accepted 1 Dec. 2020. Date of publication 8 Dec. 2020; date of current version 5 Jan. 2021. (Corresponding authors: Hao Chen and Jianhua Sun.) Recommended for acceptance by Jianfeng Zhan. Digital Object Identifier no. 10.1109/TPDS.2020.3042965

1045-9219 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on January 10,2021 at 06:44:33 UTC from IEEE Xplore. Restrictions apply.

off-line memory checking. If used online, it has high runtime overhead (about 120 percent [26]), which makes it impractical to be deployed in production. Another work clARMOR [24] is an overflow detector based on canary (a technique embedding information into the buffer for overflow detection). clARMOR has several limitations. *First*, the detection of overflow is performed only after the kernel has completed, which opens a door for adversaries to perform attacks during kernel execution, or even makes it possible to restore the buffer content to avoid detection. *Second*, clARMOR does not work for dynamically allocated memory (i.e., the memory allocated using `malloc`). Hence, clARMOR provides limited protection for memory buffers on GPU.

In this paper, we introduce GMODx, a runtime software system that supports the following features to detect GPU buffer overflow for co-runing kernels. *High efficiency:* GMODx incurs low runtime overhead and consumes little GPU resource, and is hence practical to be deployed in a production environment. *Better detection:* GMODx performs always-on monitoring on GPU memory buffers. It can detect buffer overflow that happens at either the beginning or end of user buffers. *High transparency:* GMODx only requires programmers to make little change to the application, or even no modifications if the target program only uses the coarse-grained memory management. It does not demand special system support from compilers, device driver, or hardware.

GMODx is based on canary and utilizes secret keys and buffer address to generate respective canaries for each buffer for high security. For the fine-grained memory management (using `malloc` to allocate memory on GPU), to avoid performance overhead, GMODx introduces a set of byte arrays to store buffer information for overflow detection. The byte array-based design works for user kernels with massive number of user threads, because the byte array stores buffer information continuously in memory for better data locality, and avoid dereferencing memory pointer. Several accompanying techniques are also devised to reduce performance overhead, such as lock-free accesses to the byte arrays, delayed memory deallocation, and efficient memory reallocation and garbage collection for the byte arrays. For the coarse-grained memory management (using `cudaMalloc` to allocate memory), to mitigate GPU resource consumption, GMODx employs unified memory to delegate the always-on monitoring to the CPU. Due to low concurrency (memory allocation is performed on CPU), a custom *list* structure is used to store buffer addresses for high performance. Furthermore, the runtime is encapsulated within a dynamic shared library that interposes the memory allocation APIs, making it possible to offer protections transparently without any changes to applications.

Our main contributions are summarized as follows:

- We present a dynamic GPU memory overflow detector based on canaries. To the best of our knowledge, this is the first tool that can perform *on-line* overflow detection for both fine-grained and coarse-grained memory allocation on GPU. We make GMODx open source [27], [28].
- We propose effective approaches to enabling low performance overhead and resource consumption for buffer overflow detection.
- We extensively evaluate GMODx using representative benchmarks. The results show that GMODx does not have performance overhead for the fine-grained memory allocation, and incurs rather a small runtime overhead for the coarse-grained memory allocation. We also deploy GMODx on the TensorFlow framework, it only causes 0.8 percent overhead on average (up to 1.8 percent). Therefore, we believe it can be used as a practical solution to be deployed in production.

This paper significantly extends an earlier conference version [25] in the following ways. *First*, we present an in-depth analysis on the potential overflow of coarse-grained memory management (Section 3), by discussing concurrent kernel execution between different GPU streams, CPU threads, and even processes. *Second*, we add a full new Section 6 to present the approach which is specifically designed to detect buffer overflow for the coarse-grained memory management. For high performance, it introduces a custom list data structure and optimizations for unified memory. *Third*, a comprehensive performance evaluation on the coarse-grained memory allocation in Section 8 is conducted, which shows that GMODx incurs small runtime overhead.

2 Background

In this section, we present an overview of GPGPU memory management and the unified memory.

2.1 GPGPU Memory Management

GPU device memory is separated from the host memory on CPU. GPU device memory is traditionally managed with runtime APIs like `cudaMalloc` and `cudaFree` (called the coarse-grained memory management). Buffers dynamically allocated with these APIs are explicitly transferred to GPU, and cannot be freed during kernel execution. Modern GPU also supports dynamic memory management using `malloc` and `free` (called the fine-grained memory management), similar to the traditional counterparts on CPU.

In addition, using coarse-grained memory allocation before launching kernels is usually performant than fine-grained memory allocation during the execution of kernels, because the underlying system has more optimizations for contiguous and large memory management on GPU. Therefore, it is more common to use coarse-grained memory management in GPU applications [29], [30], [31], [32].

2.2 Unified Memory

The unified memory is a component of the CUDA programming model, first introduced in CUDA 6.0, which defines a managed memory space in which all processors see a single coherent memory image with a common address space. In other word, the unified memory bridges the host and device memory space so that the memory is accessible from both the CPU and GPU in the system. A program can allocate managed memory either via the function `cudaMallocManage` that is semantically similar to `cudaMalloc`, or by defining a global variable with the keyword `__managed__`. With the unified memory, we can monitor the GPU memory usage from CPU. However, this may cause large overhead. The

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on January 10,2021 at 06:44:33 UTC from IEEE Xplore. Restrictions apply.

unified memory is implemented by automatically migrating memory pages between the host and device. Concurrent read and write accesses from both sides would cause frequent page migrations, resulting in great performance loss. In this paper, we propose optimizations for the unified memory to address this issue.

3 Motivation

Buffer overflow is a type of software error that can lead to program crash, data corruption, and security breaches. A buffer overflow occurs when a program overruns the buffer's boundary and overwrites adjacent memory. Recently, two independent works [16], [17] demonstrate that overwriting GPU memory can be exploited to conduct code injection attacks. In addition, our experiments show that the coarse-grained memory allocation is also subject to buffer overflow. We explore the overflow vulnerabilities of `cudaMalloc` under different scenarios in the following.

3.1 Concurrent Kernels From Different Streams

In order to demonstrate the bufer overflow, we first conduct an experiment in which a program launches two concurrent kernels in different streams. This experiment represents a typical usage case of sharing a single GPU among multiple mutually-untrusted users. The experimental environment is almost the same as in [17] except that we allocate memory using the coarse-grained APIs. A malicious kernel tries to corrupt a benign kernel to trigger an attacking function in the malicious kernel. The results show that it is possible to mount an attack by overwriting the memory allocated by `cudaMalloc` because the allocated buffers for different kernels are *contiguous*, which can be leveraged by the malicious kernel to conduct the attack.

3.2 Concurrent Kernels From Different CPU Threads

In this section, we show it is also possible to induce overflows among concurrent kernels that are launched from different CPU threads. The configuration is similar to the previous one. In particular, allocated buffers are not *adjacent* to each other because they are from different CPU threads. Our experiment shows that the malicious kernel can still corrupt the buffer allocated with `cudaMalloc` in the benign kernel. The reason is that the whole heap memory space on GPU (managed using APIs like `cudaMalloc`) is not isolated across distinct CPU threads, and overflowing buffer can still work even though the buffers are not *contiguous*.

3.3 Concurrent Kernels From Different CPU Processes

The process abstraction provided by the operating system should provide strong isolation. However, in this experiment, we present a case in which a malicious kernel from one process can leverage the aforementioned vulnerability to attack another concurrently running kernel from a different process. We use multi-process service (MPS) to achieve concurrent execution of two processes.

In this experiment, we can also observe that buffers in the benign kernel can be overwritten. However, compared with the previous two cases, there are three limitations to conduct the attack in this case. *First*, the malicious process needs to overflow more content of the target buffer and the success probability is much lower than the other two cases, because the buffers allocated by different processes exhibit more randomness under MPS. In other words, it is difficult to sabotage the benign process because the malicious process is hard to locate the memory addresses allocated by the benign process. Although overflowing more data can increase the probability of a success attack, it may result in segmentation faults on GPU. *Second*, the malicious process must be running when the benign process is calling the malicious function, because if the malicious process has exited, the system will recycle the resources associated with the malicious function. *Third*, the attack cannot work on Volta architecture GPU because each process on Volta MPS has its own GPU address space instead of sharing the GPU address space with all other processes (previous two cases still work on Volta GPU).

3.4 Overflow in Other Memory Management Functions

There are other device memory management functions that may be subject to overflow vulnerability. According to our experiments, the malicious kernel can also corrupt the memory allocated by `cudaMallocHost`, `cudaMallocPitch` and `cudaMalloc3D`. However, buffers allocated with `cudaMalloc3DArray`, `cudaMallocArray`, and `cudaMallocMipmappedArray` are safe under overflows, because they are used to bind read-only texture memory and cannot be accessed directly from the kernel. In the following, we focus on the discussion of `cudaMalloc`, and the techniques proposed are also applicable to other APIs that are subject to the overflow attack.

3.5 Summary

In general, buffer overflow happens with both the fine-grained and coarse-grained memory management. For the coarse-grained case, our experiments show that the possibility of successfully mounting an attack in first two scenarios (Sections 3.1 and 3.2) are almost 100 percent, and with MPS, we observe a probability about 40 percent (8 successes in 20 attempts) to corrupt the benign process. In addition, the overflow issue also exists with other device memory management functions. Therefore, we urgently need an overflow detector that is easy to deploy and provides high security and performance guarantees for GPU. Existing countermeasures against buffer overflow deployed on CPU include bounds checking [33], [34], canary checking [22], non-executable memory [35], randomization [36], and so on. In this work, we focus on canary checking, a lightweight approach, whose effectiveness has been demonstrated with the wide deployment and success of StackGuard [22] and its derivation ProPolice [37]. Canaries are known values that are placed outside of a buffer to assist buffer overflow detection. Once there is a buffer overflow, the canary would be corrupted, and a failed verification of the canary value is therefore an alert of an overflow.

4 System Overview

This section presents an overview of GMODx which provides always-on monitoring on both the fine-grained and coarse-grained memory allocation. Fig. 1 generally depicts GMODx.

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on January 10,2021 at 06:44:33 UTC from IEEE Xplore. Restrictions apply.

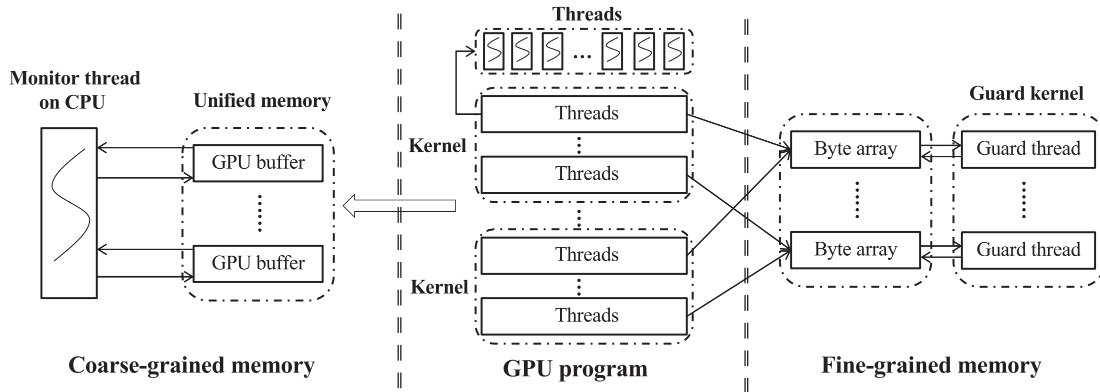


Fig. 1. Architectural overview of GMODx.

4.1 Fine-Grained Memory Detection

GMODx has three main components to detect buffer overflow for fine-grained dynamic memory allocation: (1) customized memory allocation and free functions called by the user kernel, (2) byte arrays to store buffer information, and (3) a guard kernel (Fig. 1).

The guard kernel is a GPU service that resides on GPU. Given an application with many user kernels to protect, the guard kernel is launched at the start of the application and detects overflow for any user kernel of the application. The guard kernel is launched and explicitly terminated by CPU. The guard kernel is very lightweight and uses a small amount of threads to avoid performance impact on the user kernels.

GMODx has a set of byte arrays on GPU global memory. The byte array is a data structure that saves the information of user buffers. The guard kernel examines the byte arrays to detect buffer overflow. Each byte array is associated with a thread in the guard kernel (i.e., a guard thread) for the examination. The information of user buffers allocated in a user thread is stored in a specific byte array. The association between the user thread and byte array is based on the user thread ID. Many user threads can be associated with the same byte array (and hence the same guard thread). The above design of guard thread and byte array is featured with using limited hardware resource (i.e., GPU threads) to detect buffer overflow for the massive number of user threads.

GMODx has two customized memory management functions, `mallocN` and `freeN`. These two functions extend the original functionalities of `malloc` and `free` to collect necessary information for overflow detection and garbage collection. These two functions are called by the user kernel. `mallocN` allocates a memory space (i.e., a user buffer) and then places canaries at two ends of the user buffer to detect overflow. User threads concurrently calling `mallocN` can concurrently insert buffer information into the byte arrays based on a lock-free design for high performance. The buffer information is used by the guard threads to detect buffer overflow. `freeN` flags the user buffer as free, but does not really deallocate memory. Instead, the guard kernel performs actual memory reclamation. Such method of delayed memory deallocation improves the performance of the user kernel and simplifies the design of the guard kernel.

To detect buffer overflow, the guard threads repeatedly scan the set of byte arrays. In each scan, the guard threads first perform memory reallocation and garbage collection to manage memory space usage for the byte arrays, when the free space of the byte arrays is not enough. After that, the guard threads get buffer information from the byte arrays, and locate canaries to detect buffer overflow. If no overflow is found and the current buffer has been marked as free by the user kernel, GMODx releases the buffer and flags corresponding buffer information as *expired* to avoid future scan. Fig. 2 generally depicts the overflow detection algorithm. We describe details in the following sections.

4.2 Coarse-Grained Memory Detection

GMODx has three main components for the coarse-grained memory allocation: (1) the customized memory allocation/deallocation functions called by GPU programs, (2) the unified memory component that bridges the CPU and GPU, and (3) the CPU monitor thread.

The monitor thread is the same as the guard thread except it is executed on CPU and just launch one thread for detection. Due to the fact that we delegate the monitor thread to CPU, the detection for coarse-grained memory is lightweight and consumes no GPU resources. GMODx redefines and intercepts two host memory allocation functions, i.e., `cudaMalloc` and `cudaFree` (other host memory allocation interfaces such as `cudaMallocHost`, `cudaMallocPitch` and `cudaMalloc3D` are processed similarly). It is implemented as a dynamic shared library to interpose on these functions, therefore, it can be deployed to protect

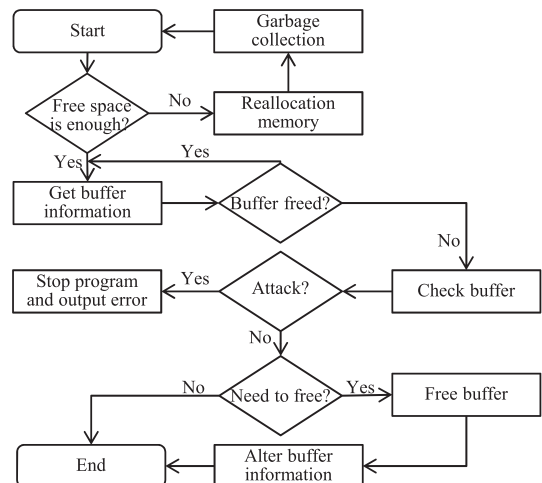


Fig. 2. Algorithm overview of GMODx.

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on January 10,2021 at 06:44:33 UTC from IEEE Xplore. Restrictions apply.

existing executables without recompilation and complex binary instrumentation. Two customized memory allocation functions extend the original functionalities to collect necessary information for overflow detection. The intercepted `cudaMalloc` allocates unified memory space (i.e., a user buffer) using `cudaMallocManaged` and places canaries at both ends of the user buffer for overflow detection. Then, it inserts the buffer address as a node to the list structure. The new `cudaFree` also employs delayed memory deallocation to reduces overhead and simplifies the design of the monitor thread. In addition, because the concurrency degree of coarse-grained memory allocation is low as compared to fine-grained memory allocation, we leverage the list structure instead of the byte array to store buffer information which removes `atomicCAS` and garbage collection and has a positive impact on performance.

5 Fine-Grained Memory Detection

5.1 Guard Kernel

The guard kernel must be lightweight, and also be able to manage a large number of user threads. In our design, the guard kernel has just one thread block, and runs in parallel with the user kernel by CUDA stream. A user thread is associated with a guard thread in charge of detecting buffer overflow for the user thread. Associating a user thread with a guard thread is based on the global thread ID of the user thread. The global thread ID is calculated based on the user kernel configuration, i.e., the number of threads in a thread block (blockDim), the user thread ID (threadIdx) and user thread's block ID (blockIdx). For example, for a user kernel with one-dimensional thread blocks, the global thread ID for a user thread is $global_tid = threadIdx.x + blockDim.x * blockIdx.x$. Assuming there are m guard threads, then the user thread with a global thread ID ($global_tid$) is assigned to a guard thread whose ID is $global_tid$ mod m. In general, we assign user threads to guard threads in a cyclic manner. We do not assign user threads in a block manner, because we want to avoid load imbalance between guard threads. In particular, we notice that some thread blocks of a user kernel can have much more dynamic memory allocation than other thread blocks. Using the block manner, some guard threads may have to detect buffer overflow for many more user buffers than other guard threads, which introduces load imbalance.

The number of guard threads in GMODx has an impact on overflow detection latency and user kernel performance. The overflow detection latency is defined as the elapsed time from the occurrence of buffer overflow to detection. Given a fixed number of user buffers to protect, a larger number of guard threads results in a smaller number of buffers per guard thread, which reduces the detection latency. However, a larger number of guard threads can negatively impact the performance of user kernel, due to competition on hardware resources (e.g., caches and global memory). Such tradeoff between detection latency and performance exists in any buffer overflow detection algorithm. We study such tradeoff in Section 8.3 using various number of guard threads, and empirically choose 32 as the number of guard threads for GMODx. Using 32 guard threads can effectively detect overflow and has no performance overhead.

Fig. 3. Buffer structure of the fine-grained memory.

5.2 Buffer Structure for Fine-Grained Memory

The fine-grained memory buffer protected by GMODx should be allocated with the GMODx's customized memory allocation function. The user buffer includes not only memory space for user data, but also buffer information (particularly, canary and buffer size). Fig. 3 depicts the buffer structure allocated by the customized memory allocation function.

In particular, the user buffer is surrounded with two words called *head canary* and *tail canary*. In this way, a buffer overflow is detected when the tail or head canary is corrupted. There is a *size* field after the head canary. This field is used to locate the tail canary based on the buffer starting address. The size field is encrypted by XORing the buffer size and a secret key. The head canary is the encryption results of a secret key (named as the head secret key), the buffer size and buffer starting address. The tail canary is built in the same way as the head canary except using a tail secret key. All the keys are fully random numbers.

GMODx encrypts the size field to effectively prevent attackers from obtaining the overall structure of buffer. If the decrypted size field is not consistent with the size value in the head canary, a buffer overflow is detected. Furthermore, each canary is unique, because it is generated based on the buffer address. Thus, even if the canary of a buffer is leaked, it is difficult to forge another buffer's canary without knowing size and address of the buffer.

5.3 Byte Array

The byte array is designed to store buffer information, including buffer addresses and whether buffers are released. We use array instead of dynamic data structures (e.g., linked list) to store the buffer information to enable good data locality. In particular, the byte array is preallocated by the guard thread, before the allocation of any user buffer. Once a user buffer is allocated, its buffer information is sequentially fed into a byte array. Hence, a guard thread can access the buffer information of many user buffers with good spatial locality. Using a dynamic data structure, such as linked list, can easily cause random memory access with bad data locality. Furthermore, a guard thread can scan a byte array without any pointer dereferencing, while using a dynamic data structure, the guard thread usually has to do so. Because dereferencing pointers is more expensive than using index of a byte array to access array elements on GPU, using byte array improves performance. However, using a preallocated byte array loses the flexibility of a dynamic data structure. We need to dynamically grow or shrink the byte array, when it runs out of array space or has too much useless buffer information. We discuss our mechanism to dynamically manage byte array space in Section 5.6.

Each guard thread is in charge of one byte array and repeatedly read the array to get buffer information. A byte array collects the buffer information for a specific number of user threads. Assuming that the total number of threads from user kernels is M and the total number of guard

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on January 10,2021 at 06:44:33 UTC from IEEE Xplore. Restrictions apply.

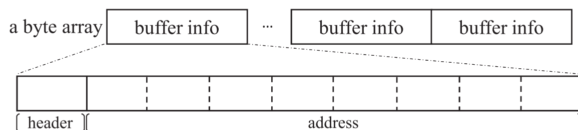


Fig. 4. An illustration example for a byte array.

threads is N, then each byte array collects the buffer information for M/N user threads.

The structure of a byte array is as follows (see Fig. 4). A byte array contains buffer information for a number of user buffers (see "buffer info" in Fig. 4). Each buffer information consists of a buffer address (8 bytes) and a header (1 byte). The header is used to indicate if the user buffer is expired or not. A user buffer is expired, if the buffer is released by a guard thread (not user thread). If a user buffer is expired, the guard kernel skips its information for overflow detection. The header can also be used to avoid a read-after-write hazard (see Section 5.5 for details).

5.4 Memory Allocation and Free

We extend CUDA's original dynamic memory management functions `malloc` and `free`. To make the discussion easy, we use `mallocN` and `freeN` to represent our version.

`mallocN` allocates memory using the original `malloc` function. Furthermore, `mallocN` expands the allocated memory space to accommodate extra information for the user buffer. Such information includes canaries and buffer size (see Section 5.2). After the memory space allocation, `mallocN` insert buffer addresses into the byte array and set the corresponding header field as 1. `mallocN` introduces more operations than regular `malloc`. However, those operations are lightweight and have ignorable performance impact on the user kernel execution time, as shown in Section 8.

`freeN` uses a two-step algorithm to free memory. In the first step, the head canary of the user buffer is updated with a new value (called *free canary*). The free canary is the encryption result of the old head canary with a secrete key. The free canary can be used to detect the double free problem (see Section 5.8 for details). `freeN` returns without actually releasing the memory.

The second step happens after `freeN` returns. In particular, when a guard thread examines the buffer to detect overflow and finds that a buffer has a free canary, the guard thread releases the buffer after performing canary verification. The guard thread also resets the header field to 0 (expired buffer) in the corresponding byte array to avoid using the information of the freed user buffer in subsequent execution.

We use the above two-step algorithm, because it simplifies our design for memory deallocation and improves performance. In particular, using guard threads to release user buffer removes overhead of releasing buffer from the user threads, hence improving performance. If we ask user threads to release memory, then user threads must introduce a mechanism (e.g., a hash table) to efficiently locate user buffer information in byte arrays and update it. Such mechanism complicates our design while introducing extra overhead into critical path of the user kernel.

The two-step algorithm has a drawback: memory deallocation is delayed. However, the delay is short and no longer

```
1  __device__ void insertBuffInfo(uint64_t address)
2  {
3    int32_t old_idx, tmp_idx;
4    int8_t* old_ptr;
5    int8_t header=1;
6    /* Insert into byte array with lock-free operation */
7    do {
8      old_idx = byte_array->idx;
9      old_ptr = byte_array->ptr;
10     tmp_idx = old_idx + 9;
11   } while ((old_idx) != atomicCAS(&byte_array->idx,
12                                   old_idx, tmp_idx));
13   memcpy(old_ptr + old_idx + 1, &address, 8);
14   /* Guarantee finishing insertion of buffer
15      information before decompressing it */
16   memcpy(old_ptr + old_idx, &header, 1);
17 }
```

Fig. 5. A code snippet that inserts buffer information into a byte array. As a data structure, the byte array has two extra fields to facilitate insertion: the field idx that points to the first free byte in the byte array, and the field ptr that points to the beginning of the byte array.

than the time of scanning a byte array for once by a guard thread. Since the time of scanning a byte array for once is very short (typically much shorter than the kernel execution time), the freed memory can be timely released.

5.5 Lock-Free Insertion

We describe how inserting buffer information in byte arrays happens in details in this section. Fig. 5 generally explains the algorithm with a code snippet. The algorithm is featured with a lock-free design to handle potential data races on the byte array.

The algorithm locates a position within a byte array to insert header and address. The major challenge to do so is to coordinate concurrent requests for inserting multiple user buffer information from multiple user threads. The algorithm uses a lock-free design (Lines 7-12) based on `atomicCAS` which is a hardware-based atomic operation. In particular, each user thread tries to atomically update the byte array without relying on explicit locking to coordinate concurrent updates to the byte array.

After the position to insert the header and address is located, the algorithm inserts the header and address into the byte array by memcpy (Lines 13-16). Note that we do not use locking to coordinate concurrent updates from multiple user threads, because the position where the header and address will be inserted has been secured in Lines 7-12. We also do not use locking to coordinate between the guard thread that will read the buffer information and the user thread that is updating the byte array. Instead, we control the order of adding the header and address to avoid using locking. In particular, we add the address first and then the header. Without setting up the header first, the guard thread is not able to read the new buffer information, hence we avoid potential data races between the user thread and guard thread.

5.6 Garbage Collection and Memory Reallocation

Using byte arrays, we must dynamically manage memory space. In particular, we must dynamically expand capacity of the byte arrays to accommodate information of new user buffers, and reclaim memory space of those useless buffer information (i.e., garbage collection) in byte arrays. We

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on January 10,2021 at 06:44:33 UTC from IEEE Xplore. Restrictions apply.

introduce a memory space management mechanism with minimized performance impact on user threads. We describe our dynamic memory management as follows.

A byte array initially has s (s=20 KB) to accommodate buffer information for 2,000 user buffers. When the byte array has only x% free space (x=20 in our implementation), the guard thread doubles the size of the byte array (i.e., memory reallocation) and performs garbage collection. Such memory management does not wait for the drain of the byte array. Instead, memory management is proactive, and always provides sufficient space in the byte array to accommodate new buffer information. In addition, because of the SIMD nature of GPU, 32 threads run concurrently on a GPU multiprocessor. This indicates that 32 guard threads can concurrently perform garbage collection and memory reallocation on 32 byte-arrays. Such concurrency in memory management reduces memory management overhead.

We evaluate our choice of s (20 KB) and x (20) with stress testing with intensive memory allocation (see Section 8.2). Our evaluation shows that our choice of s and x does not block user threads for memory management in the byte array, and hence always provide sufficient space.

To reclaim memory space for garbage collection, we do not recycle those elements of the byte array that have useless buffer information, because this method requires the user threads to examine which element of the byte array has useless buffer information and hence introduces excessive overhead into the user threads. Instead, our method aggregates the valid buffer information into a new byte array without garbage, for garbage collection.

We use the code snippet in Fig. 6 to further explain garbage collection. Before garbage collection, each guard thread allocates a new byte array (`new_barray`) that doubles the memory size of the old byte array. The byte array after garbage collection will be in `new_barray`. The byte array before garbage collection is saved in an array `old_barray` (Line 13); Garbage collection happens in a loop (Lines 16-35). In the loop, the guard thread reads user buffer information from `old_barray`, checks if each user buffer is freed or not (Lines 23), and copies the valid buffer information from `old_barray` to the new one (Lines 31-33).

To coordinate concurrent accesses to the byte array from the user kernel (inserting addresses) and guard kernel (moving valid addresses from the old byte array to the new one), we introduce a lock-free design, shown as two loops in Lines 25-29 in Fig. 6 and Lines 7-12 in Fig. 5. The two lock-free `while` loops cooperate to guarantee the correctness of concurrent insertions and garbage collection.

We use the lock-free loop in Fig. 5 to discuss a case with data race and explain the effectiveness of the lock-free design. In this case, replacing `byte_array` with `new_barray` by the guard thread (Line 14 of Fig. 6) competes with reading `byte_array` (Line 8 in Fig. 5) by the user thread, creating a read-after-write hazard. Assume that a user thread already obtains the index (Lines 8 in Fig. 5) before garbage collection happens. Afterwards, garbage collection happens and the byte array is updated. The old index is not valid any more. However, we have no problem for program correctness in this case, because of the following reason. When the user thread runs to Line 11 in Fig. 5, the old buffer index obtained by the user thread will not be equal to the new buffer index

```
1  __device__ byteArray* new_barray[GUARD_THREADS];
2  __device__ byteArray* old_barray[GUARD_THREADS];
3  __device__ void garbageCollection()
4  {
5    int tid = threadIdx.x+ blockDim.x * blockIdx.x;
6    new_barray[tid] = new byteArray();
7    new_barray[tid]->prt = allocateMemory();
8    new_barray[tid]->idx = 0;
9    __syncthreads();
10
11   /* The variable byte_array is a pointer of
12      the type byteArray accessed by each guard thread */
13   old_barray[tid] = byte_array;
14   byte_array = new_barray[tid];
15
16   for (int32_t i = 0; i < old_barray[tid]->idx;) {
17     int8_t header = 0;
18     /* Get the header */
19     memcpy(&header, old_barray[tid]->ptr + i, 1);
20     /* Increase the increment i */
21     i += 9;
22
23     if (header != 0) {
24       int32_t old_idx, tmp_idx;
25       do {
26         old_idx = byte_array->idx;
27         tmp_idx = old_idx + 9;
28       } while ((old_idx) != atomicCAS(&byte_array->idx,
29                                      old_idx, tmp_idx));
30       /* Copy the content of old_barray to byte_array */
31       memcpy(byte_array->ptr + old_idx + 1,
32         old_barray[tid]->ptr + i - 8, 8);
33       memcpy(byte_array->ptr + old_idx, &header, 1);
34     }
35   }
36   __syncthreads();
37   ... /* Free the old byte array */
38 }
```

Fig. 6. A code snippet for garbage collection.

(i.e., the return value of atomicCAS). The user thread will try to get the buffer index again, which ensures that the user thread always obtains the most recently updated buffer index to insert the address.

We use the lock-free loop in Fig. 5 to discuss another case with data race to explain the effectiveness of the lock-free design. In this case, replacing `byte_array` with `new_barray` by the guard thread (Line 14 of Fig. 6) competes with using memcpy (Lines 13-16 in Fig. 5) to write the header and address into the byte array by the user thread, creating a write-after-read hazard. Assume that a user thread already obtains an index of the old byte array for inserting new buffer information (Lines 11 and 12 in Fig. 5). Afterwards, garbage collection happens and the byte array is updated. In this case, the user thread inserts the new buffer information into the new byte array but uses the index of the old byte array. However, we still have no problem in this case, because of the following reason. Line 9 in Fig. 5 stores a pointer of the old byte array and subsequent memcpy uses it to insert the new buffer information, which ensures correct insertion of the new buffer information into the old byte array. Although the new buffer information is inserted into the old byte array, the subsequent garbage collection copies the new buffer information from the old byte array to the new byte array, hence the new buffer information can still be correctly placed into the new byte array.

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on January 10,2021 at 06:44:33 UTC from IEEE Xplore. Restrictions apply.

5.7 Detection of Buffer Overflow

The guard kernel is responsible for overflow detection. To do so, the guard thread first reads the header within a buffer information in a byte array. If the header indicates that the buffer has expired, the guard thread moves on to the next buffer information and ignores the analysis on the current buffer. Otherwise, the guard thread uses the address field in the buffer information to obtain user buffer address.

After obtaining the user buffer address, it is straightforward to conduct overflow detection. If the buffer has been freed in `freeN`, the guard thread releases this buffer after verifying the canaries, and sets the header field to 0 (expired). Otherwise (the buffer is still in use), the guard thread performs canary verification to detect potential overflows. We explain the canary verification in details as follows.

When a buffer is allocated, the head and tail canaries are constructed. The head canary is constructed as follows.

$$head\ canary = size \oplus buffer\ address \oplus head\ secret\ key. \tag{1}$$

The tail canary is constructed in the same way, except that it uses a different secret key. For the canary verification, the guard thread first decrypts the size field embedded in the front of the user buffer, and then recalculates the canaries based on the buffer address, buffer size and secret keys. The guard thread then compares the recalculated canary with the canaries in the user buffer for canary verification. If the canaries or the size field are corrupted, the verification fails, indicating the occurrence of buffer overflow.

5.8 Detection of Double Free

Different from the detection of overflow by the guard threads, detecting double free is conducted in `freeN` by the user threads. The basic idea of detecting double free is to introduce a canary, named free canary. The user thread replaces the head canary with the free canary to indicate that the buffer is freed.

Note that we ask the user thread (not the guard thread) to replace the head canary and detect double free, because it is the user thread that initiates the operations of buffer freeing. The guard thread cannot easily know whether a free operation happens, after the buffer is already marked as free by the user kernel. More details on the detection of double free are as follows.

When a buffer is freed, the user thread first decides if the head canary of the buffer is equal to the calculated free canary based on Equation (2). If yes, the buffer has been freed and the head canary has been replaced with *free canary*, which indicates that double free is detected. If no (i.e., the head canary is not equal to free canary based on Equation (2)), then we replace the head canary with free canary based on Equation (3). The reason why we use Equation (3) (not Equation (2)) here is as follows:

$$free\ canary = \tag{2}$$

$$size \oplus address \oplus head\ secret\ key \oplus free\ secret\ key$$

$$free\ canary = head\ canary \oplus free\ secret\ key. \tag{3}$$

Simply speaking, using Equation (3) enables detection of buffer overflow during the detection of double free. If the head canary is corrupted, then using Equation (3), the corruption in the head canary will be carried by the free canary and detected by the guard thread later on. Using Equation (2), the corruption in the head canary will not be carried by the free canary and cannot be detected by the guard thread. Note that if the head canary is not corrupted, Equations (2) and (3) are mathematically the same (given Equation (1)), hence using Equation (3) does not negatively impact the detection of double free.

6 Coarse-Grained Memory Detection

6.1 Monitor Thread

The monitor thread is similar to the guard thread except it is executed on CPU and just launches one thread to detect. Once the monitor thread is launched, it performs always-on monitoring on allocated buffers until the user program exits.

When the user program calls `cudaMalloc` to allocate buffer memory at the first time, the monitor thread is created on CPU. As same as guard thread, the monitor thread repeatedly scans the list structure. When the user program terminates, the *exit* function that is registered with `atexit` is automatically called to stop the monitor thread. Then the monitor thread scans the list structure *twice* before termination. The reason is as follows. If the monitor thread immediately exits once it receives the signal, the overflow cannot be detected if the buffer address is stored in the first half of the list, while the monitor is scanning the second half of the list.

6.2 List Structure

The list structure (Fig. 7) is used to store memory addresses for GPU kernels. The buffer addresses are inserted when calling `cudaMalloc`. The node is the basic unit of the list structure, and stores the buffer address and a pointer that points to the next node. The list structure uses a never-removed dummy node that stores nothing as the head node, and we design customized insertion and deletion to eliminate data race when insertions from the user program and deletions from the monitor thread happen at the same time. This custom list structure avoids specific designs for synchronization (e.g., lock and `atomicCAS`), which has a positive impact on performance. We describe the operations of the list in the following.

For insertion, new nodes are always inserted between the dummy node and the first user node (the node linked immediately after the dummy node). For example, node C is inserted between the dummy node and node A in Fig. 7 (1) (2). When the user buffer is released by the monitor thread (not the user program), the corresponding node becomes invalid, and the monitor thread sets the expired buffer address with an invalid flag. In the next scan, the monitor thread will skip this invalid node and considers to delete it. To delete an invalid node, if the invalid node is the first user node (the node A in Fig. 7 (1)), it will not be removed even if it is marked with the invalid flag. After a new node is inserted (the node C in Fig. 7 (2)), the monitor thread will delete it (Fig. 7 (3)). For other invalid nodes (such as the node B in Fig. 7), the monitor thread deletes them immediately. In this way, the new node insertion always happens between

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on January 10,2021 at 06:44:33 UTC from IEEE Xplore. Restrictions apply.

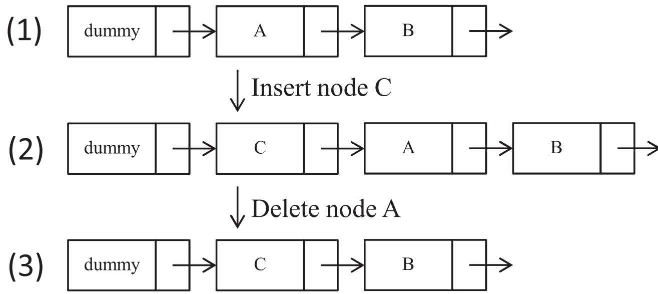


Fig. 7. List structure. From (1) to (2), node C is inserted into the list; and from (2) to (3), node A is deleted from the list only if node A is not the next node of the dummy node.

the dummy node and the last inserted node, so we can eliminate the data hazard between node insertion and deletion. The correctness is guaranteed when insertions and deletions happen at the same time, otherwise, specific synchronization mechanisms such as locks are needed to avoid data race in these cases. In addition, this list structure is easy to extend to fit multiple CPU threads, by configuring one list structure per CPU thread (in most cases, one CPU thread and one list structure is enough).

6.3 Unified Memory Optimization

In this section, we first study the potential performance issue when employing unified memory in GMODx, then we demonstrate how to address this issue.

The functionality of unified memory is based on automatic memory paging between the host and device. It can cause large overhead when the data migration between the two sides dominates the kernel execution time. We explore the reason that frequent memory paging may happen in the detector. For the unified memory, a 4 KB memory page [38] is the basic unit of memory migration between the host and device. Although the monitor thread on CPU just needs to access the metadata (i.e., canaries and buffer size) that is used to detect buffer overflow, the whole 4 KB memory space including part of the user buffer will be migrated to the host. Another memory transfer is required once the user kernel begins to use the buffer. Certain memory access patterns may result in excessive page migrations. Fig. 8 (1) shows memory spaces (gray spaces) that both the CPU and GPU need to access.

To solve this problem, we align the memory to 4 KB boundary (Fig. 8 (2)). Therefore, the head canary with the buffer size is migrated to the host without transferring the real content of GPU applications, and the tail canary is constructed in the same way. In addition, the canaries and buffer size are mostly read by the monitor thread, therefore, we can use the `cudaMemAdvise` API (indicating the unified memory subsystem about the memory access pattern) to mark them with the `cudaMemAdviseSetReadMostly` flag to further optimize performance. This flag indicates that the data is mostly-read and only occasionally written to. Prefetching can also be used to improve performance using the `cudaMemPrefetchAsync` API that can migrate data to GPU and insert corresponding entries in GPU page table before the kernel begins accessing the data. Data prefetching is to avoid page faults while also establishing data locality.

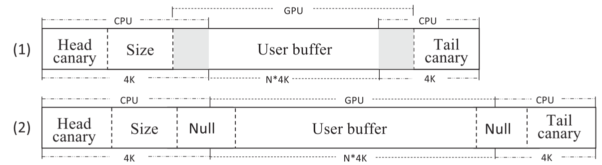


Fig. 8. Buffer structure. (1) a customized buffer structure of fine-grained memory. (2) a customized buffer structure of coarse-grained which is aligned to 4 KB boundary.

6.4 Memory Allocation and Deallocation

We extend CUDA's original memory management functions `cudaMalloc` and `cudaFree`. The new `cudaMalloc` involves two main operations. One is that we use `cudaMallocManaged` to allocate unified memory. The other is that extra memory space is reserved to accommodate metadata for the user buffer, including canaries and buffer size (see Section 6.3). After allocating memory space, the new `cudaMalloc` inserts the buffer address into the list structure. Our extended `cudaMalloc` introduces more operations than the original one. However, its performance impact on the user kernel is negligible because these operations are lightweight. Our new `cudaFree` also uses a two-step algorithm to deallocate memory.

6.5 Hooking cudaMalloc and cudaFree

In this section, we present how to implement GMODx as a dynamic shared library. To this end, we need to hook and extend related functions. However, there are two problems. First, the runtime functions such as `cudaMalloc` and `cudaFree` cannot be hooked (new versions of CUDA prevent the runtime functions from being hooked). Therefore, we intercept the underlying driver APIs such as `cuMemAlloc` and `cuMemFree`, which are used by the runtime functions to allocate and release memory. Second, using the function `dlopen` to get the address of driver functions can bypass hooking, so we need to redefine `dlsym` to replace the original one. Other functions (such as `cudaMallocPitch`, `cudaMalloc3D` and so on) are intercepted and extended in the same way. Details are as follows.

Fig. 9 (Lines 13-24) shows how `cuMemAlloc` and `cuMemFree` are hooked. The customized `cuMemAlloc` calls `alloc_wrapper` to allocate unified memory that is filled with extra information (canaries and buffer size) for overflow detection. The customized `cuMemFree` first stores the original pointer of `cuMemFree` that will be used to release the buffer later. Then, it calls `free_wrapper` to perform delayed free. In addition, Line 21 in Fig. 9 calls `cudaDeviceSynchronize` to avoid prematurely releasing the buffer. If there is no such synchronization to ensure that kernels have finished before the delayed free, the user kernel may access the memory that has been released by the monitor thread. To redefine `dlsym` function, we first leverage `__libc_dlsym` to get the original `dlsym` function (Lines 1-11 in Fig. 9), and then define its customized version (Lines 26-34 in Fig. 9) to replace the original one. Therefore, when one tries to use `dlsym` to get the function pointer of `cuMemAlloc` and `cuMemFree`, the redefined `dlsym` will return our customized `cuMemAlloc` and `cuMemFree`, otherwise, it returns the original `dlsym`.

Because of packaging GMODx as a shared library, the deployment becomes flexible. By setting the LD_PRELOAD

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on January 10,2021 at 06:44:33 UTC from IEEE Xplore. Restrictions apply.

```
1  extern "C" {
2    void* __libc_dlsym(void *map, const char *name);
3  }
4
5  static void* real_dlsym(void *handle, const char* symbol) {
6    if (!internal_dlsym) {
7      void* handle = dlopen("libdl.so.2", RTLD_LAZY);
8      internal_dlsym = (fnDlsym)__libc_dlsym(handle, "dlsym");
9    }
10   return (*internal_dlsym)(handle, symbol);
11 }
12
13 extern "C" {
14   CUresult CUDAAPI cuMemAlloc(CUdeviceptr *p, size_t s) {
15     return alloc_wrapper(p, s);
16   }
17   CUresult CUDAAPI cuMemFree(CUdeviceptr p) {
18     if (!real_cuMemFree) {
19       ...//get original cuMemFree function by real_dlsym
20     }
21     cudaDeviceSynchronize();
22     return free_wrapper(p);
23   }
24 }
25
26 void* dlsym(void *handle, const char *symbol) {
27   if (strcmp(symbol, "cuMemFree") == 0) {
28     return (void*)(cuMemFree);
29   }
30   else if (strcmp(symbol, "cuMemAlloc") == 0) {
31     return (void*)(cuMemAlloc);
32   }
33     return (real_dlsym(handle, symbol));
34 }
```

Fig. 9. The code snippet of hooking cudaMalloc and cudaFree.

environment variable to the path of the shared library, GMODx can be invoked to protect the memory allocated with coarse-grained APIs without extra modifications.

7 Discussion

Detection With Coarse-Grained Memory Management on CPU. Compared with our previous work [25], GMODx immigrates the coarse-grained memory detection from GPU to CPU. This design mitigates GPU resource consumption that would be beneficial to the performance of user GPU applications. More importantly, the CPU-based detection mechanism does not introduce any device code, so it can be implemented as a shared dynamic library [43] and is transparent to user applications. Meanwhile, most GPU applications and real workloads only employ the coarse-grained memory management (detailed in Section 2.1). As a result, for these real applications, it is easy to deploy overflow detection without source code modifications.

8 Evaluation

8.1 Experimental Setup

Our experiments were performed on a system with a 2.4 GHz Intel Xeon CPU E5-2630, and an NVIDIA GeForce GTX 1070 discrete GPU. The system runs ubuntu 16.04.4 LTS with NVIDIA graphics driver version 384.11 and CUDA runtime 9.0 installed.

We use five benchmarks (three normal benchmarks and two micro-benchmarks) to evaluate the fine-grained memory detection and 10 benchmarks to evaluate coarse-grained memory detection (five benchmarks with `cudaMalloc`, three benchmarks with `cudaMallocPitch`, and two benchmarks with `cudaMallocHost`). In particular, we focus on evaluating `cudaMalloc` and `cudaFree`, because they are widely used in GPU applications. At last, we deploy GMODx with the TensorFlow [42] framework to evaluate its performance impact on real and complex workloads. In addition, There is no applications using `cudaMalloc3D` in normal benchmark suites (Parboil [29], Rodinia [30], SHOC [31], NUPAR [32] and CUDA SDK [41]). However, `cudaMallocPitch` calls the same driver API (`cuMemAllocPitch`) as `cudaMalloc3D`, so we believe its evaluation results are also relevant to `cudaMalloc3D`.

TABLE 1
Benchmarks for Evaluating GMODx

Benchmark	Num.ker	Sum.Allocation
alloc-dealloc (ad) [39]	1	20K
alloc-cycle-dealloc (acd) [39]	5	100K
grid-point (gp) [40]	1	20K
add-string (ads) [40]	3	60K
random-graph (rg) [40]	1	60K
bfs [29]	3	14
mri-fhd [29]	4	11
pns [29]	2	3
mri-q [29]	4	9
fft [29]	44	3
simplePitchLinearTexture [41]	200	2
mummergpu [30]	14	9
_bilateral [30]	151	3
concurrentKernels [41]	9	2
mri-gridding [29]	43	22
TensorFlow [42]	360K	1.1K

Num.ker *is the number of user kernels for evaluation.* Sum.Allocation *is the total number of memory allocations in a GPU program. For fine-grained memory allocation,* Sum.Allocation *is related to the number of user threads in a benchmark, so we report the maximum number of memory allocations in our evaluation.*

The benchmarks are summarized in Table 1. These benchmarks use the default kernel configurations, unless indicated otherwise. The benchmarks alloc-dealloc, alloc-cycle-dealloc, add-string, random-graph, grid-point, simplePitchLinearTexture, mummergpu, _bilateral, concurentKernel and mri-gridding are called ad, acd, ads, rg, gp simPit, mum, bil, conKer and mri-g for short in this section. Among 15 benchmarks, ad, acd, ads, rg, and gp have fine-grained memory management, and other 10 benchmarks have coarse-grained memory allocations (benchmark bfs, mri-fhd, pns, mri-q, and fft allocate memory with `cudaMalloc`, benchmarks simPit, mum, and bil allocate memory with `cudaMallocPitch`, and benchmarks conKer and mri-g allocate memory with `cudaMallocHost`). For TensorFlow, we use data augmentation offered in the TensorFlow benchmark [44] (the batch size is 32) to train typical models ResNet-50 [45], Inception-v3 [46], VGG-16 [47], and AlexNet [48]. We employ ImageNet as the training data set for all models. ImageNet is a large image dataset with millions of images belonging to thousands of categories. All results reported in this section are the average of 20 runs.

The reasons why we use these benchmarks are as follows. We use the two micro-benchmarks [39], because they have very intensive memory allocations, which allows us to evaluate the performance of GMODx with stress testing.

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on January 10,2021 at 06:44:33 UTC from IEEE Xplore. Restrictions apply.

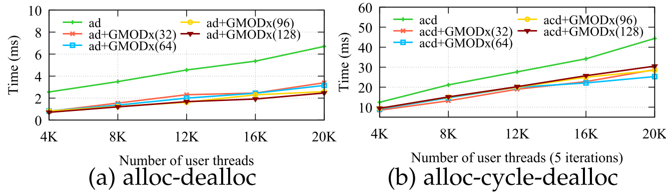


Fig. 10. Performance results for two stress tests (preliminary performance study). The number within parenthesis is the number of guard threads.

We use the three benchmarks (gp, ads, and rg) from Halloc [40], because they support fine-grained memory allocation on GPU which cannot be found in the common GPU benchmark suites, such as Parboil [29], Rodinia [30], SHOC [31] and NUPAR [32]. Benchmarks bfs, mri-fhd, pns, mri-q, and fft from Parboil [29] are representative programs using coarse-grained memory allocation, and they are complicated enough. In addition, their performance is sensitive to the disturbance from any co-running benchmarks. Benchmarks simPit, mum, and bil use `cudaMallocPitch` to allocate memory and benchmarks conKer and mri-g use `cudaMallocHost` to allocate memory.

8.2 Performance of Fine-Grained Memory Detection

As a preliminary performance study, we use two microbenchmarks (alloc-dealloc and alloc-cycle-dealloc) that have very intensive memory allocations to evaluate overflow detection on fine-grained memory allocations under stress testing. Figs. 10a and 10b depict the results.

Using GMODx, instead of performance loss, we see performance improvement, comparing to the cases without GMODx. For example, with 32 and 64 guard threads for *alloc-dealloc*, we have 55.4 and 58.4 percent performance improvement, respectively; With 32 and 64 of guard threads for *alloc-cycle-dealloc*, we have 33.8 and 32.8 percent performance improvement, respectively. The performance improvement comes from the asynchronous design of `freeN` that delegates memory deallocation to the guard kernel.

Fig. 10a reveals that increasing the number of guard threads leads to better performance (shorter execution time) in alloc-dealloc, although in many cases there is no big performance difference between different cases. Having more guard threads can result in better performance, because GMODx has more resources (e.g., byte arrays and guard threads) to detect buffer overflow. More byte arrays also indicate less competition between user threads when concurrent updates to byte arrays happen. However, Fig. 10b reveals that increasing the number of guard threads leads to worse performance in alloc-cycle-dealloc. We attribute this to resource contention on memory (bandwidth and cache) between user threads and guard threads. A larger number of guard threads cause more resource contention, hence causing performance loss. We further discuss the effect of the number of guard threads in the next section.

8.3 Sensitivity of Fine-Grained Memory Detection

In this section, we study how sensitive the detection latency and user kernel performance are with regard to the number of guard threads.

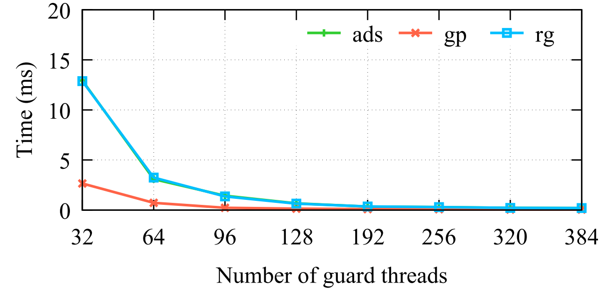


Fig. 11. Average execution time of a guard thread scanning its byte array for once.

Detection Latency. Similar to other tools detecting buffer overflow on CPU [49], [50], [51], GMODx does not provide real-time detection. To study the detection latency, we use three benchmarks. For each benchmark, we use different numbers of guard threads to run GMODx. Fig. 11 shows the detection latency which is the average execution time of a guard thread to scan its byte array for once. The reason why we use such estimated detection latency is as follows.

The guard thread repeatedly scans the byte array. Any buffer overflow should be detected by the guard thread in one of those scans. In addition, the execution time of a guard thread scanning the byte array for once depends on the length of the byte array. Since the length of the byte array is dynamically changed at runtime, we use the average execution time as a statistical quantification on the detection latency. This approach has been commonly used in existing work [51], [52].

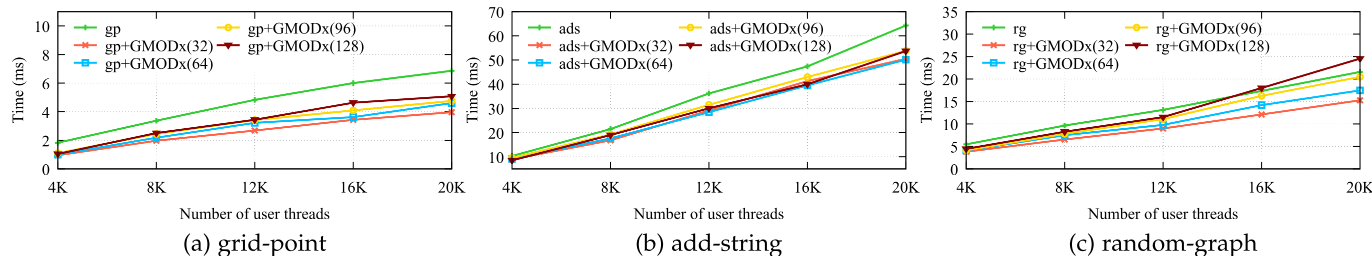
Fig. 11 shows that the detection latencies in almost all cases (22 out of 24 cases) are less than the execution time of user kernels, which demonstrates that our detection latency is short enough in most cases. In only two cases (grid-point and random-graph when the number of guard threads is 32) we can observe that the detection latency is longer than the execution time of user kernel. However, by increasing the number of guard threads (larger than 32), we can make the detection latency shorter than the duration of user kernel.

Note that although GMODx cannot detect overflow before the user kernel finishes in a few cases, GMODx can still detect overflow for those cases after the user kernel exits. In particular, when the user kernel finishes, the overflowed buffer has been freed by a user thread using `freeN`. However, it still exists in the global memory of GPU, because of our design of the delayed free. The guard thread can still detect the overflow even after the user kernel finishes.

Performance Impact on User Kernel. We study the impact of the number of guard threads on user kernel performance. Figs. 12a, 12b, and 12c shows the results with different number of guard threads and user threads. Table 2 reports the average performance improvement for variouse number of guard threads (32, 64, 96, and 128), based on Figs. 12a, 12b, and 12c.

The results show that among 60 cases shown in Figs. 12a, 12b, and 12c, 59 (except random-graph with 20K user threads and 128 guard threads) have performance improvement, due to the asynchronous design of `freeN` and ignorable performance overhead of GMODx. We also notice that with more guard threads, the performance benefit becomes

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on January 10,2021 at 06:44:33 UTC from IEEE Xplore. Restrictions apply.



(a) grid-point (b) add-string (c) random-graph

Fig. 12. Performance of three benchmarks with GMODx. The numbers in parenthesis indicate the number of guard threads.

smaller. In summary, with 32 guard threads, the improvement is 30 percent on average (up to 48 percent) in all cases.

Based on the above study on the detection latency and user kernel performance, we empirically use 32 threads as the number of guard threads for GMODx, because 32 guard threads bring low detection latency and small performance impact on user kernel performance. We do not use less than 32 guard threads, due to the warp-based scheduling nature on GPU.

8.4 Overhead of the Customized malloc

Different from typical usage scenarios where `mallocN` and `freeN` come in pairs, in this experiment, we unveil the overhead of `mallocN` by hiding the influence of `freeN`. We separate the performance impact of `mallocN` from that of `freeN`, because `freeN` can bring performance benefit while `mallocN` cannot.

Figs. 13a, 13b, and 13c show the results. The results show that the average overhead of our `mallocN` is about 10.9 percent (random-graph), 6.2 percent (add-string), and 5.1 percent (grid-point). In general, the customized `malloc` results in overhead less than 11 percent. This overhead is relatively small. This demonstrates the effectiveness of our high performance design. More importantly, the overhead of the customized `malloc` can be easily hidden by the performance benefit of the customized `free`, as shown in Figs. 12a, 12b, and 12c.

8.5 Performance of Coarse-Grained Memory Detection

Detection Latency. To study the detection latency, we use five benchmarks. Table 3 shows the detection latency that is the average execution time of GMODx scanning the whole list structure. Table 3 shows that the detection latencies of all cases are shorter than 0.1us, which is far less than the execution time of user kernels. It demonstrates that GMODx can promptly detect buffer overflows from the CPU side.

TABLE 2
The Average Performance Improvement for Three Benchmarks With GMODx

	32	64	96	128
grid-point	43.8%	37.3 %	31.7%	29.2%
add-string	17.8%	17.8%	11.3%	15.6%
random-graph	30.6%	22%	14%	5.4%

This table is based on Figs. 12a, 12b, and 12c. The first row is the number of guard threads. A positive percentile represents performance improvement.

Performance Impact on User Kernel. As for the five normal benchmarks (Fig. 14) that cover a wide range of domains, the overhead imposed by GMODx is 4.2 percent (up to 9.7 percent) on average. Different from the fine-grained memory detection, the coarse-grained memory detection has performance loss, because `cudaFree` is executed on CPU and the design of delayed memory deallocation does not work in this situation. The overhead imposed on bfs is relatively high due to its significantly heavy memory allocations and accesses. The unified memory must guarantee data consistency between the CPU and GPU, which introduces extra overhead when executing write operations. Therefore, a large amount of write operations in bfs increase the overhead.

Performance of `cudaMallocPitch` and `cudaMallocHost`. In this section, we study the detector's performance when using `cudaMallocPitch` and `cudaMallocHost`. Fig. 15 presents the performance of GMODx, which is normalized by the execution time that is obtained when the detection is disabled. In general, we can see the overhead imposed by GMODx is negligible for both `cudaMallocPitch` and `cudaMallocHost` except for the benchmark bil (9.3 percent). The overhead imposed on bil is relatively high, because it involves heavy memory allocations and accesses like the benchmark bfs.

8.6 Page Faults

As shown in Section 6.3, several approaches are used to improve the performance of unified memory. To illustrate the effectiveness of these optimizations, we compare the performance under two scenarios, one with the optimizations enabled and one with the optimizations disabled.

Table 4 shows the results. H to D indicates the number of memory transfers from the host to device. GPU PG Group (GPU page fault group) and CPU PG (CPU page faults) represent the density of memory swap that happens when the CPU and GPU access a memory page that is not currently mapped in their own memory. The number of page fault groups is not the total number of page faults on GPU. On the CUDA platform, page faults are written to a special buffer in system memory and multiple faults forming a group are processed simultaneously by the unified memory driver. Therefore, using GPU page fault groups is more representative than GPU page faults.

From Table 4, we can observe obvious reduction of host-to-device transfers and CPU page faults (better performance). The average number of host-to-device (not including bfs) transfers drops from 198.2 to 13.2, and the average number of CPU page faults (besides bfs) decreases from 157.8 to 21.8. In addition, the number of GPU page fault groups is eliminated when our optimization is applied. For bfs, a large amount of memory

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on January 10,2021 at 06:44:33 UTC from IEEE Xplore. Restrictions apply.

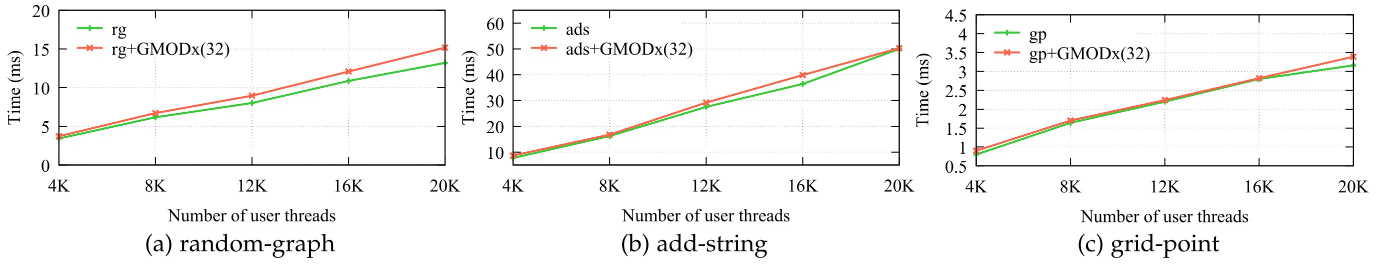


Fig. 13. Performance impact of `mallocN`.

transfers and memory swaps make the program crash when the optimizations are disabled, and it fails to output any results. In contrast, bfs achieves good performance with the optimizations. This indicates that the optimizations can effectively reduce the overhead introduced by the detector.

8.7 Performance of Co-Running Benchmarks

To study the performance of GMODx with co-running kernels, we choose two benchmarks from the eight benchmarks to co-run, resulting in 28 combinations as shown in Fig. 16.

Fine-Grained Memory Detection. The benchmarks `gp-ads`, `gp-rg`, and `ads-rg` are used to demonstrate the performance of fine-grained memory detection. GMODx does not cause performance degradation because of the asynchronous design of freeN that delegates memory deallocation to the guard kernel.

Coarse-Grained Memory Detection. There are ten cases in total that demonstrate the performance of coarse-grained memory detection in GMODx. All benchmarks achieve low overhead (less than 4 percent), except for the two cases where bfs is paired with mri-q and fft, which incurs overhead less than 10 percent.

Concurrent Coarse-Grained and Fine-Grained Memory Detection. GMODx can concurrently detect overflows on both coarse-grained and fine-grained memory in 18 cases. In general, GMODx does not cause performance loss except the case of `gp-pns` (1.4 percent loss). For the benchmark `gp-pns`, `pns` dominate the whole execution of `gp-pns`. As a result, the performance degradation from coarse-grained memory detection overweighs the performance improvement of fine-grained memory detection.

8.8 Performance of GMODx on TensorFlow

To show the performance of GMODx on a real workload, we deploy GMODx with TensorFlow to train four typical models. As shown in Fig. 17, GMODx causes 0.8 percent overhead on average (up to 1.8 percent). GMODx achieves such high performance because training models with TensorFlow is compute-intensive and GMODx has negligible cost with such workloads. In addition, GMODx does not compete against the model-training tasks for GPU resource when detecting overflow for the coarse-grained memory, which is beneficial

TABLE 3
The First Three Rows Show the Execution Time With or Without Detectors (GMODx and cuda-memcheck) Enabled

Benchmark	bfs	mri-fhd	pns	mri-q	fft
App (ms)	14.5	23.4	224.1	2.6	0.106
App + GMODx (ms)	15.9	23.5	227	2.7	0.113
App + cuda-mem (ms)	184.1	33.1	5,975	7.4	4.1
DL (us)	0.08	0.04	0.04	0.022	0.013
MC (KB)	71,936	5,292	720,000	5,267	6,368
MC + GMODx (KB)	72,097	5,382	720,035	5,332	6,496

The fourth row (DL) shows the detection latency. The rows 5 and 6 show the original memory cost and the memory cost with our detector.

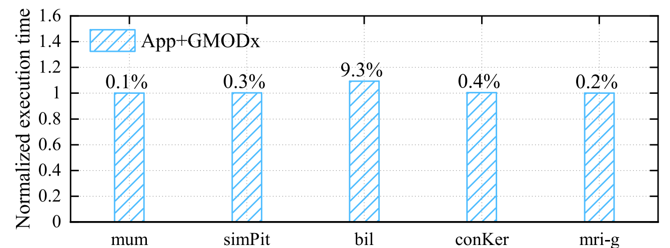


Fig. 15. Performance evaluation of GMODx with five benchmarks which allocate memory using `cudaMallocPitch` and `cudaMallocHost`. The performance is the normalized execution time.

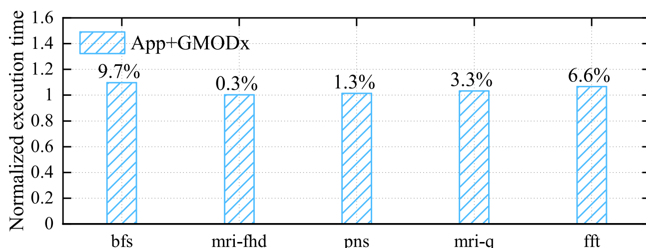


Fig. 14. Performance evaluation of GMODx with five benchmarks. The performance is normalized by the execution time of applications without GMODx.

TABLE 4
H to D Means the Number of Memory Transfers From Host Memory to Device Memory, GPU GP Group is the Number of GPU Page Fault Groups, and CPU PG is the Number of CPU Page Faults

	Benchmark	bfs	mri-fhd	pns	mri-q	fft
H to D	original	-	244.6	211.3	151.6	185.4
	optimization	20	16	4	13	20
GPU PG Group	original	-	133.7	882	82.7	161.6
	optimization	0	0	0	0	0
CPU PG	original	-	192.1	201	103.9	134.3
	optimization	25.2	29.7	6	19	32.8

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on January 10,2021 at 06:44:33 UTC from IEEE Xplore. Restrictions apply.

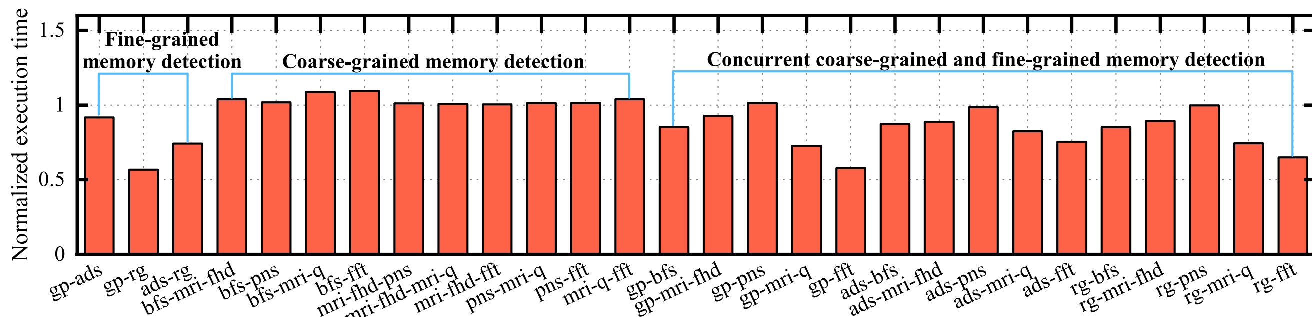


Fig. 16. Performance evaluation of GMODx with co-running benchmarks. The performance is normalized by the execution time that is measured with GMODx disabled. Benchmarks `gp-ads`, `gp-rg`, and `ads-rg` show the performance of fine-grained memory detection. From benchmark `bfs-mri-fhd` to `mri-q-fft`, we evaluate the performance of coarse-grained memory detection. From benchmark `gp-bfs` to `rg-fft`, we evaluate the concurrent detection on both coarse-grained and fine-grained.

to the overall performance. Note that GMODx will not degrade the target quality (such as accuracy) because it does not change the behaviors of training and inference (such as algorithms) for TensorFlow and corresponding models.

8.9 Effectiveness

We evaluate the effectiveness of GMODx in detecting buffer overflow. We conduct eight experiments. The first three experiments are also used in the existing studies [16], [17] to demonstrate the existence of GPU buffer overflow on fine-grained memory allocation, and the other five benchmarks are used for coarse-grained memory allocation.

We use GMODx to detect buffer overflow in these benchmarks. In all experiments, once GMODx finds an overflow, it stops the execution of user kernels and outputs overflow information such as the user address where overflow happens. Table 5 shows the results, and the *kernel time* is the execution time of the user kernel. In all cases, GMODx successfully detects overflow and the detection latency is much shorter than the kernel execution time.

8.10 Comparison With Existing Work

cuda-memcheck. To show the performance benefit of GMODx, we compare its performance with cuda-memcheck. We select cuda-memcheck instead of clARMOR, because clARMOR is designed for OpenCL, but our detector and cuda-memcheck both target the CUDA framework. Fig. 18 shows the normalized execution time of cuda-memcheck. In general, cuda-memcheck incurs significant overhead (at least 10 percent and up to 65.8x), which is much larger than GMODx.

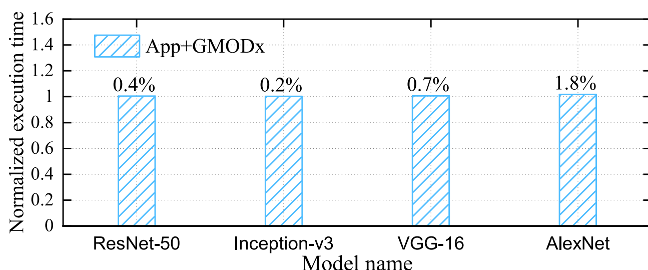


Fig. 17. Performance evaluation of GMODx when training four models with the TensorFlow framework. The performance is the normalized execution time.

GMOD. We compare the performance of GMODx with our previous work GMOD, to illustrate how differently the coarse-grained memory detection behaves on GPU and CPU. For the benchmarks bfs, mri-fhd, pns, mri-q, and fft, GMOD causes 7.7, 2, 4.5, 6.1 and 6.7 percent performance degradation respectively. In general, GMODx has better performance than GMOD in all benchmarks except bfs. The benchmark bfs has a worse performance in GMODx because its significantly heavy memory accesses introduces extra overhead in the unified memory. In addition, GMODx achieves higher transparency because it requires no modifications to user source code.

8.11 Memory Cost

For each user buffer, the fine-grained memory detection needs 24 bytes for storing canaries and 9 bytes for storing buffer information in the byte array. This is a rather small memory cost. Since there is no canary-based overflow detector for dynamically allocated buffers on GPU, we compare GMODx with a canary-based overflow detector on CPU [51] in terms of memory cost. This detector on CPU is a state-of-the-art overflow detector on CPU. This detector on CPU introduces 24 bytes for canaries and 16 bytes for storing address, which consumes larger memory than GMODx. Table 3 show the extra memory consumption for detecting coarse-grained memory and lists the absolute amount of memory space that GMODx needs. The results reveal that the memory consumption is less than 2 percent, and the absolute amount of memory space

TABLE 5
Using Eight Buffer Overflow Benchmarks to Evaluate the Effectiveness of GMODx

	Kernel time	Detection latency	Detected?
Single kernel	0.947 ms	0.0145 ms	yes
Sequential kernels	1.273 ms	0.0098 ms	yes
Concurrent kernels	10.818 ms	0.0121 ms	yes
Single kernel	0.9185 ms	0.021 us	yes
Sequential kernels	1.223 ms	0.023 us	yes
Concurrent kernels (stream)	10.95 ms	0.027 us	yes
Concurrent kernels (thread)	7.316 ms	0.0245 us	yes
Concurrent kernels (process)	9.481 ms	0.031 us	yes

The first three benchmarks use fine-grained memory allocation and the other five use coarse-grained memory allocation.

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on January 10,2021 at 06:44:33 UTC from IEEE Xplore. Restrictions apply.

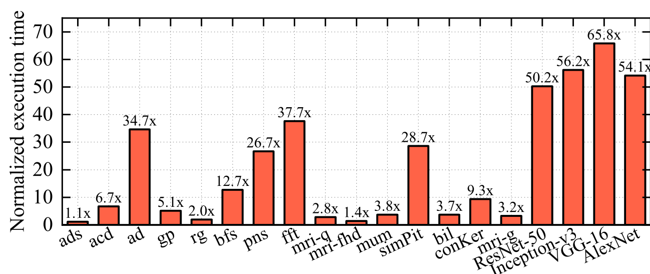


Fig. 18. Performance evaluation of cuda-memcheck. Performance is normalized by the execution time when overflow detection is disabled.

is less than 161 KB, which means the memory consumption for the coarse-grained detection is negligible.

9 Related Work

Buffer Overflow Detection on CPU. Many static analysis tools [33], [34] use bounds checking to detect buffer overflows by analyzing source code statically. This approach suffers from high false positive or false negative rate. Canary is first proposed in Electric Fence [21], which tackles stack smashing attacks by placing a canary word before return address on stack. Address Space Layout Randomization (ASLR) [36] generates different addresses for stack and heap variables for different executions, such that buffer overflow attacks cannot be achieved reliably. Cruiser [51] is a concurrent heap buffer overflow detector on CPU, and it is similar to GMODx, but we focus on GPU.

Security Issues on GPU. The study in [53] shows that adversaries can retrieve other processes's data stored in GPU memory by analyzing the memory dump of GPU devices. Maurice *et al.* [54] highlight possible information leakage of GPUs in virtualized and cloud environments. In [55], Pietro *et al.* present a detailed analysis of information leakage in CUDA. Our paper is different from these work by focusing on defeating buffer overflow and double free with low performance overhead.

Multi-Kernel Concurrent Execution on GPUs. In order to improve the utilization of GPUs, researchers propose solutions to run multiple kernels from different users concurrently on GPUs. In [56], Ravi *et al.* present a framework to enable applications executing within virtual machines to transparently share one or more GPUs. Pai *et al.* [57] propose transformations to convert CUDA kernels into elastic kernels in order to gain fine-grained control over resource usage. Current endeavors on multi-kernel execution focus on improving resource utilization, security and reliability issues are not concerned. GMODx complements these work by considering GPU security.

GPU Memory Overflow. Miele [16] presents a preliminary study of buffer overflow vulnerabilities in CUDA. An attacker can overrun a buffer to corrupt sensitive data or steer the execution flow by overwriting function pointers, e.g., manipulating the virtual table of a C++ object. In [17], Di *et al.* demonstrate the existence of stack and heap overflows, although stack overflows have limited impact on security. cuda-memcheck is a tool for checking CUDA memory errors [23], and it can detect heap overflows mentioned above. But its runtime overhead makes it impractical to be deployed in production, and it was reported that the overhead incurred by cuda-memcheck is roughly 120 percent [26].

clARMOR [24] is another GPU buffer overflow detector using a canary-based design. It offers runtime protection with reasonable overhead. Although both clARMOR and our detector use the canary-based design, they are fundamentally different. In particular, clARMOR performs detection only *after* the kernel has completed. This means that clARMOR cannot detect buffer overflow for fine-grained memory allocation (i.e., `malloc`). GMODx can detect buffer overflow *during* kernel execution, for *both* fine-grained and coarse-grained memory allocation. In addition, detecting buffer overflow during the kernel execution is challenging, because we must avoid the impact of the detector on the performance of user kernels. GMODx introduces a series of techniques to alleviate performance overhead.

10 Conclusion and Future Work

In this paper, we present the design and implementation of a dynamic GPU memory overflow detector GMODx which performs always-on monitoring on buffers that are allocated using both fine-grained and coarse-grained memory APIs. GMODx can effectively identify buffer overflow with ignorable performance overhead. In the future, It is interesting and challenging to embed our detector into GPU co-run frameworks, which consider combining multiple small tasks to both improve resource utilization for GPUs.

Acknowledgments

This research was supported in part by the National Science Foundation of China under Grants 61972137 and 61772183, and in part by Hunan Provincial Natural Science Foundation of China under Grant 2016JJ3042.

References

[1] W. Zhong *et al.*, "Optimizing graph processing on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 4, pp. 1149–1162, Apr. 2017.

[2] J. Sun, H. Chen, L. He, and H. Tan, "Redundant network traffic elimination with GPU accelerated rabin fingerprinting," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 7, pp. 2130–2142, Jul. 2016.

[3] C. Chen, K. Li, A. Ouyang, Z. Zeng, and K. Li, "GFlink: An in-memory computing architecture on heterogeneous CPU-GPU clusters for big data," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 6, pp. 1275–1288, Jun. 2018.

[4] C. Chen, K. Li, A. Ouyang, Z. Tang, and K. Li, "GPU-accelerated parallel hierarchical extreme learning machine on flink for big data," *IEEE Trans. Syst.*, Man, Cybern., Syst., vol. 47, no. 10, pp. 2740–2753, Oct. 2017.

[5] J. Chen *et al.*, "A parallel random forest algorithm for big data in a spark cloud computing environment," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 4, pp. 919–933, Apr. 2017.

[6] C. Chen, K. Li, A. Ouyang, and K. Li, "FlinkCL: An OpenCL-based in-memory computing architecture on heterogeneous CPU-GPU clusters for big data," *IEEE Trans. Comput.*, vol. 67, no. 12, pp. 1765–1779, Dec. 2018.

[7] H. Chen, J. Sun, L. He, K. Li, and H. Tan, "BAG: Managing GPU as buffer cache in operating systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1393–1402, Jun. 2014.

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on January 10,2021 at 06:44:33 UTC from IEEE Xplore. Restrictions apply.

[8] L. Shi, H. Chen, J. Sun, and K. Li, "vCUDA: GPU-accelerated high-performance computing in virtual machines," *IEEE Trans. Comput.*, vol. 61, no. 6, pp. 804–816, Jun. 2012.

[9] H. Tan, Y. Tan, X. He, K. Li, and K. Li, "A virtual multi-channel GPU fair scheduling method for virtual machines," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 2, pp. 257–270, Feb. 2019.

[10] H. Li, K. Li, J. An, and K. Li, "MSGD: A novel matrix factorization approach for large-scale collaborative filtering recommender systems on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 7, pp. 1530–1544, Jul. 2018.

[11] K. Li, W. Yang, and K. Li, "Performance analysis and optimization for SpMV on GPU using probabilistic modeling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 1, pp. 196–205, Jan. 2015.

[12] W. Yang, K. Li, Z. Mo, and K. Li, "Performance optimization using partitioned SpMV on GPUs and multicore CPUs," *IEEE Trans. Comput.*, vol. 64, no. 9, pp. 2623–2636, Sep. 2015.

[13] K. Jang, S. Han, S. Han, S. Moon, and K. Park, "SSLShader: Cheap SSL acceleration with commodity processors," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implementation*, 2011, pp. 1–14.

[14] K. Li, W. Yang, and K. Li, "A hybrid parallel solving algorithm on GPU for quasi-tridiagonal system of linear equations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 10, pp. 2795–2808, Oct. 2016.

[15] X. Zhu, K. Li, A. Salah, L. Shi, and K. Li, "Parallel implementation of MAFFT on CUDA-enabled graphics hardware," *IEEE/ACM Trans. Comput. Biol. Bioinf.*, vol. 12, no. 1, pp. 205–218, Jan./Feb. 2015.

[16] A. Miele, "Buffer overflow vulnerabilities in CUDA: A preliminary analysis," *J. Comput. Virology Hacking Techn.*, vol. 12, no. 2, pp. 113–120, 2016.

[17] B. Di, J. H. Sun, and H. Chen, "A study of overflow vulnerabilities on GPUs," in *Proc. 13th Int. Conf. Netw. Parallel Comput.*, 2016, pp. 103–115.

[18] C. C. Zou, W. Gong, and D. F. Towsley, "Code red worm propagation modeling and analysis," in *Proc. 9th ACM Conf. Comput. Commun. Secur.*, 2002, pp. 138–147.

[19] H. K. Orman, "The morris worm: A fifteen-year perspective," *IEEE Security Privacy*, vol. 1, no. 5, pp. 35–43, Sep./Oct. 2003.

[20] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford-Chen, and N. Weaver, "Inside the slammer worm," *IEEE Security Privacy*, vol. 1, no. 4, pp. 33–39, Jul./Aug. 2003.

[21] B. Perens, "Electric fence," 1987. [Online]. Available: http://linux.die.net/man/3/efence

[22] C. Cowan, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proc. 7th Conf. USENIX Secur. Symp.*, 1998, Art. no. 5.

[23] NVIDIA, "Cuda-memcheck," 2017. [Online]. Available: https://developer.nvidia.com/cuda-memcheck

[24] C. Erb, M. Collins, and J. L. Greathouse, "Dynamic buffer overflow detection for GPGPUs," in *Proc. Int. Symp. Code Gener. Optim.*, 2017, pp. 61–73.

[25] B. Di, J. Sun, D. Li, H. Chen, and Z. Quan, "GMOD: A dynamic GPU memory overflow detector," in *Proc. 27th Int. Conf. Parallel Architectures Compilation Techn.*, 2018, Art. no. 20.

[26] T. M. Baumann and J. Gracia, "Cudagrind: Memory usage checking for CUDA," in *Proc. 7th Int. Workshop Parallel Tools High Perform. Comput.*, 2013, pp. 67–78.

[27] GMODx, "GMODx," 2020. [Online]. Available: https://github.com/aimlab/overflow

[28] GMODx, "GMODx," 2020. [Online]. Available: https://gitlab.com/db93/gmodx

[29] J. A. Stratton *et al.*, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," Univ. Illinois Urbana-Champaign, Center Reliable High-Perform. Comput., Tech. Rep. IMPACT-12-01, 2012.

[30] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization*, 2009, pp. 44–54.

[31] A. Danalis *et al.*, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proc. 3rd Workshop General-Purpose Comput. Graph. Process. Units*, 2010, pp. 63–74.

[32] Y. Ukidave *et al.*, "NUPAR: A benchmark suite for modern GPU architectures," in *Proc. 6th ACM/SPEC Int. Conf. Perform. Eng.*, 2015, pp. 253–264.

[33] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2000, pp. 3–17.

[34] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. L. Wang, "Cyclone: A safe dialect of C," in *Proc. General Track Annu. Conf. USENIX Annu. Tech. Conf.*, 2002, pp. 275–288.

[35] The PaX project, 2017. [Online]. Available: http://pax.grsecurity.net/

[36] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits," in *Proc. 12th USENIX Secur. Symp.*, 2003, pp. 291–301.

[37] Propolice detector. 2006. [Online]. Available: https:www.trl.ibm.com/projects/security/ssp/

[38] D. Ganguly, Z. Zhang, J. Yang, and R. G. Melhem, "Interplay between hardware prefetcher and page eviction policy in CPU-GPU unified virtual memory," in *Proc. 46th Int. Symp. Comput. Archit.*, 2019, pp. 224–235.

[39] M. Vinkler and V. Havran, "Register efficient dynamic memory allocator for GPUs," *Comput. Graph. Forum*, vol. 34, no. 8, pp. 143–154, 2015.

[40] A. V. Adinetz, "Halloc GPU memory allocator," 2014. [Online]. Available: https://github.com/canonizer/halloc

[41] NVIDIA, "CUDASDK," 2018. [Online]. Available: https://docs.nvidia.com/cuda/cuda-samples/index.html

[42] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 265–283.

[43] NVIDIA, "Libraries," 2020. [Online]. Available: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#libraries

[44] TensorFlow, "tf_cnn_benchmarks: High performance benchmarks," 2020. [Online]. Available: https://github.com/tensorflow/benchmarks/tree/master/scripts/tf_cnn_benchmarks

[45] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[46] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 2818–2826.

[47] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. 3rd Int. Conf. Learn. Representations*, 2015, pp. 1–14.

[48] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. 26th Annu. Conf. Neural Inf. Process. Syst.*, 2012, pp. 1106–1114.

[49] S. shield, Website, 2000. [Online]. Available: http://www.angelfire.com/sk/stackshield/

[50] K. Avijit, P. Gupta, and D. Gupta, "TIED, LibsafePlus: Tools for runtime buffer overflow protection," in *Proc. 13th USENIX Secur. Symp.*, 2004, pp. 45–56.

[51] Q. Zeng, D. H. Wu, and P. Liu, "Cruiser: Concurrent heap buffer overflow monitoring using lock-free data structures," in *Proc. 32nd ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2011, pp. 367–377.

[52] D. H. Tian, Q. Zeng, D. H. Wu, P. Liu, and C. Z. Hu, "Kruiser: Semi-synchronized non-blocking concurrent kernel heap buffer overflow monitoring," in *Proc. 19th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2012, pp. 1–16.

[53] S. Breß, S. Kiltz, and M. Schäler, "Forensics on GPU coprocessing in databases - Research challenges, first experiments, and countermeasures," in *Proc. Datenbanksysteme für Business, Technologie und Web (BTW)*, 2013, pp. 115–129.

[54] C. Maurice, C. Neumann, O. Heen, and A. Francillon, "Confidentiality issues on a GPU in a virtualized environment," in *Proc. 18th Int. Conf. Financial Cryptography Data Secur.*, 2014, pp. 119–135.

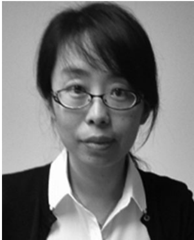
[55] R. D. Pietro, F. Lombardi, and A. Villani, "CUDA leaks: A detailed hack for CUDA and a (partial) fix," *ACM Trans. Embedded Comput. Syst.*, vol. 15, no. 1, pp. 15:1–15:25, 2016.

[56] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar, "Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework," in *Proc. 20th Int. Symp. High Perform. Distrib. Comput.*, 2011, pp. 217–228.

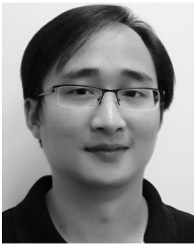
[57] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU concurrency with elastic kernels," in *Proc. 18th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2013, pp. 407–418.

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on January 10,2021 at 06:44:33 UTC from IEEE Xplore. Restrictions apply.

Bang Di is currently working toward the PhD degree with the College of Computer Science and Electronic Engineering, Hunan University, China. His research interests include heterogeneous computing systems and GPU security.



Jianhua Sun received the PhD degree in computer science from the Huazhong University of Science and Technology, China in 2005. She is a professor with the College of Computer Science and Electronic Engineering, Hunan University, China. Her research interests include security and operating systems. She has published more than 70 papers in journals and conferences, such as the *IEEE Transactions on Parallel and Distributed Systems*, and *IEEE Transactions on Computers*.



Hao Chen (Member, IEEE) received the BS degree in chemical engineering from Sichuan University, China, in 1998, and the PhD degree in computer science from the Huazhong University of Science and Technology, China, in 2005. He is currently a professor at the College of Computer Science and Electronic Engineering, Hunan University, China. His current research interests include parallel and distributed computing, operating systems, cloud computing, and systems security. He has published more than 70 papers in journals and conferences, such as the *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, IPDPS, IWQoS, and ICPP. He is a member of ACM.

Dong Li is a professor at EECS, University of California, Merced, Merced, CA, USA. Previously, he was a research scientist at the Oak Ridge National Laboratory (ORNL), Oak Ridge, TN, USA His research focuses on high performance computing (HPC) and maintains a strong relevance to computer systems. The core theme of his research is to study how to enable scalable and efficient execution of enterprise and scientific applications on increasingly complex large-scale parallel systems. He received a CAREER Award from U.S. National Science Foundation, in 2016, and an ORNL/CSMD Distinguished Contributor Award (2013). His paper in SC'14 was nominated as the Best Student Paper. He is also the lead PI for NVIDIA CUDA Research Center at UC Merced. He is a member of ACM.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on January 10,2021 at 06:44:33 UTC from IEEE Xplore. Restrictions apply.