GMOD: A Dynamic GPU Memory Overflow Detector

Bang Di[1], Jianhua Sun[1], Dong Li[2], Hao Chen[1], and Zhe Quan[1]

[1]College of Computer Science and Electronic Engineering, Hunan University, Changsha, China,
{dibang,jhsun,haochen,quanzhe}@hnu.edu.cn
[2]Electrical Engineering and Computer Science, University of California, Merced, dli35@ucmerced.edu

ABSTRACT

Rich thread-level parallelism in GPU has motivated co-running GPU kernels on a single GPU. However, when GPU kernels co-run, it is possible that a 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. The existing work has either large performance overhead or limited capability to detect buffer overflow.

In this paper, we introduce GMOD, a runtime software system that detects GPU buffer overflow. GMOD performs always-on monitoring on dynamically allocated buffers based on a canary-based design. GMOD introduces a set of byte arrays to store buffer information for buffer overflow detection. To enable high performance, GMOD introduces several techniques, such as lock-free accesses to the byte arrays, delayed memory free for high performance memory free, and efficient memory reallocation and garbage collection for the byte arrays. Our experiments show that GMOD is capable of detecting buffer overflows at runtime and has small runtime overhead (2.9% on average and up to 9.1%).

CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Computer systems organization → Single instruction, multiple data**;

KEYWORDS

Buffer Overflows, CUDA, GPU, High Performance

PACT '18, November 1–4, 2018, Limassol, Cyprus

ACM ISBN 978-1-4503-5986-3/18/11…$15.00
https://doi.org/10.1145/3243176.3243194

ACM Reference Format:
Bang Di[1], Jianhua Sun[1], Dong Li[2], Hao Chen[1], and Zhe Quan[1] . 2018. GMOD: A Dynamic GPU Memory Overflow Detector. In *International conference on Parallel Architectures and Compilation Techniques (PACT '18), November 1–4, 2018, Limassol, Cyprus.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3243176.3243194

1 INTRODUCTION

Graphics processing units (GPUs) are widely adopted in HPC and cloud computing platforms to accelerate general-purpose workloads. Rich thread-level parallelism in GPU has motivated co-running GPU kernels on a single GPU. Co-running GPU kernels improves GPU utilization and maximizes system throughput. As a result, co-running GPU kernels has been employed in many scenarios, such as machine learning model inference [22] and database query [29]. 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 [9, 15].

Despite extensive research over the past few decades, buffer overflow remains one of the top software vulnerabilities. Many notorious attacks, such as Code Red [30], Morris Worm [31], and Slammer [32], exploit buffer overflow, and can result in program crash, data corruption, and security breaches. Those attacks overwrite heap memory buffer allocated by applications at runtime on CPU. The heap buffer overflow also exists on GPUs as introduced in recent studies [9, 15]. Those studies demonstrate that heap 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 non-trivival 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 those memory 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 those resources can be effectively used for computation in

regular GPU kernels. *Second*, GPU lacks some system support available on CPU, such as page protection and preemptive execution (most GPU does not support preemptive execution). The traditional security mechanism, such as Electric Fence [18] and StackGuard [7] 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 the concurrent execution of GPU kernels and buffer overflow detection mechanism, we should avoid data races between the GPU kernels and buffer overflow detection mechanism, without frequently interrupting the kernels execution.

Unfortunately, there is very limited work [10, 16] aiming to detect buffer overflow for GPU. The existing work has either large performance overhead or limited protection. The existing work, cuda-memcheck [16], can identify the source and cause of memory access errors in GPU code, based on intensive code instrumentation. However, cuda-memcheck is a tool for *off-line* memory checking. If used online, cuda-memcheck has high runtime overhead (about 120% [3]), which makes it impractical to be deployed in production environments. Another existing work clARMOR [10] is an overflow detector based on canary (a technique embedding some 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 window 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 fine-grained, dynamically allocated memory (i.e., the memory allocated with `malloc`). *Third*, clARMOR cannot detect buffer overflow that happens at the beginning of the buffer. Hence, clARMOR provides limited protection for memory buffers in GPU code.

In this paper, we introduce GMOD, a runtime software system that supports the following features to detect GPU buffer overflow for co-run kernels at runtime:

- **High efficiency:** GMOD incurs small runtime overhead (2.9% on average and up to 9.1%) and consumes little hardware resource (i.e., threads and global memory), and is hence practical to be deployed in a production environment.
- **Better detection:** GMOD performs always-on monitoring on dynamically allocated user buffers (including fine-grained memory allocation by `malloc`). It can detect buffer overflow that happens at either the beginning or end of user buffers.
- **High transparency:** GMOD only requires programmers to make little change to the application, and does not require special system support from compilers, device driver or hardware.

GMOD is based on canary, and employs a set of techniques to avoid performance overhead. In particular, GMOD utilizes secret keys and buffer address to generate respective canaries of each buffer for high security. Moreover, GMOD introduces a set of byte arrays to store buffer information for buffer overflow detection. The byte array-based design works for user kernels with massive number of user threads. It also avoids performance problems associated with common alternative solutions (e.g., linked list), because the byte array stores buffer information continuously in memory for better data locality, and avoid dereferencing memory pointer. To enable high performance of our design, we introduce several techniques, such as lock-free accesses to the byte arrays, delayed memory free for high performance memory free, and efficient memory reallocation and garbage collection for the byte arrays. All together, these techniques avoid introducing extra latency into the execution of user kernels.

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* detection of overflow for dynamically allocated GPU buffers. We make GMOD open source [11].
- We propose effective approaches to enable low performance overhead and resource consumption for buffer overflow detection;
- We extensively evaluate GMOD using representative benchmarks, and microbenchmarks for stressing tests. Evaluation results show that GMOD incurs rather small runtime overhead and can be used as a practical solution to be deployed in production.

2 BACKGROUND AND MOTIVATION

In this section, we present an overview of buffer overflow, and GPGPU memory management.

2.1 Buffer Overflow

Buffer overflows are software errors that can lead to program crashes, data corruption, and security breaches. A buffer overflow occurs when a program overruns the buffer's boundary and overwrites adjacent memory. Existing countermeasures against buffer overflows deployed on CPU include bounds checking [13, 28], canary checking [7], non-executable memory [20], randomization [4], 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 [7] and its derivation ProPolice [12]. Canaries are known values that are placed outside of a buffer to assist buffer overflow detection. When a buffer overflows,

the canary will be corrupted, and a failed verification of the canary value is therefore an alert of an overflow.

2.2 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`. Buffers dynamically allocated with these APIs are explicitly transferred to GPU, and can not be freed during kernel execution. Modern GPU also supports dynamic memory management using `malloc` and `free`, similar to the traditional counterparts on CPU. `cudaMalloc` and `cudaFree` are coarse-grained memory management, because the memories are allocated and freed before and after GPU kernels are launched. `malloc` and `free` are fine-grained, because they allow memory allocation and free whenever needed in the middle of kernel execution.

GPU applications rarely perform operations like dereferencing pointers, and corrupted GPU buffers mostly would not cause crashes. Hence GPU buffer overflows have often been left undetected, and the research community has ignored the importance of building tools to deal with memory overflows on GPU. Recently, two independent works [9, 15] demonstrate that overwriting dynamically allocated GPU buffers can be exploited to conduct code injection attacks. Worse, heterogeneous systems in which CPU and GPU share the same physical or virtual memory would exacerbate this problem. In this paper, we focus on addressing GPU buffer overflows caused by dynamic memory allocation.

In Section 3 (Overview) and Section 4 (Design), we focus on the fine-grained, dynamic memory management (`malloc` and `free`). In Section 5, we slightly extend the design for the coarse-grained, dynamic memory management (`cudaMalloc` and `cudaFree`). In Section 6 (Evaluation), we evaluate both of the dynamic memory management methods.

3 SYSTEM OVERVIEW

This section presents an overview of GMOD. To detect buffer overflow for fine-grained, dynamic memory allocation, GMOD consists of three main components: (1) customized memory allocation and free functions called by the user kernel, (2) byte arrays to store buffer information, and (3) a guard kernel. Figure 1 generally depicts GMOD.

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: it uses a small amount of threads to avoid performance impact on the user kernels.

GMOD has a set of byte arrays on GPU global memory. The byte array is a data structure that saves the information

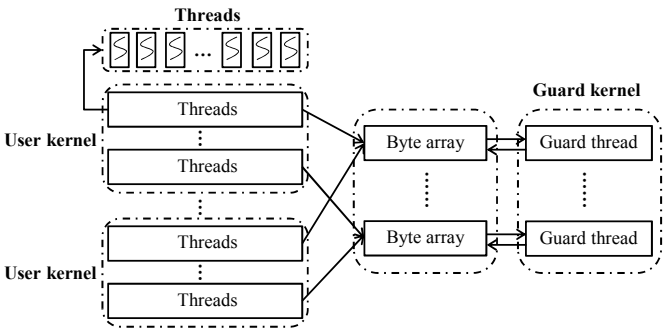


Figure 1: Architectural overview of GMOD.

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 massive number of user threads.

GMOD 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 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, GMOD releases the buffer and flags corresponding buffer information as *expired* to avoid future scan.

Figure 2 generally depicts the overflow detection algorithm. We describe our design in details as follows.

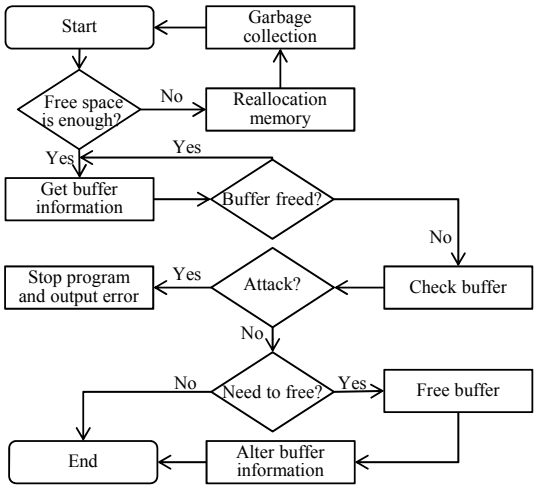


Figure 2: GMOD Algorithm overview.

4 DESIGN AND IMPLEMENTATION

4.1 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 GMOD 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 6.3 using various number of guard threads for 40 use cases, and empirically choose 32 as the number of guard threads for GMOD. Using 32 guard threads can effectively detect overflow and has small runtime overhead (2.9% on average and up to 9.1%).

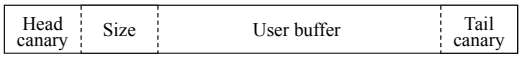


Figure 3: Buffer structure

4.2 Buffer Structure

The user buffer protected by GMOD should be allocated with the GMOD'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). Figure 3 depicts the buffer structure allocated by GMOD's 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.

GMOD 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.

4.3 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

```
__device__ void insertBuffInfo(uint64_t address)
{
  int32_t old_idx, tmp_idx;
  int8_t* old_ptr;
  int8_t  header=1;
  /* Insert into byte array with lock-free operation */
  do {
    old_idx = byte_array->idx;
    old_ptr = byte_array->ptr;
    tmp_idx = old_idx + 9;
  } while ((old_idx) != atomicCAS(&byte_array->idx,
                                  old_idx, tmp_idx));
  memcpy(old_ptr + old_idx + 1, &address, 8);
  /* Guarantee finishing insertion of buffer
     information before decompressing it */
  memcpy(old_ptr + old_idx, &header, 1);
}
```

Figure 4: 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.

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 4.6.

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 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 Figure 5). A byte array contains buffer information for a number of user buffers (see "buffer info" in Figure 5). 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 4.5 for details).

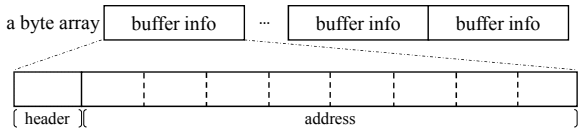


Figure 5: An illustration example for a byte array.

4.4 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 4.2). 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 6.

`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 4.8 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 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.

4.5 Lock-free Insertion

We describe how inserting buffer information in byte arrays happens in details in this section. Figure 4 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.

4.6 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 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 2000 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

```
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 }
```

Figure 6: A code snippet for garbage collection.

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 (20KB) and x (20) with stress testing with intensive memory allocation (see Section 6.2). 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 Figure 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 Figure 6 and Lines 7-12 in Figure 4. The two lock-free **`while`** loops cooperate to guarantee the correctness of concurrent insertions and garbage collection.

We use the lock-free loop in Figure 4 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 Figure 6) competes with reading `byte_array` (Line 8 in Figure 4) by the user thread, creating a read-after-write hazard. Assume that a user thread already obtains the index (Lines 8 in Figure 4) 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 Figure 4, the old buffer index obtained by the user thread will not be equal to the new buffer index (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 Figure 4 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 Figure 6) competes with using memcpy (Lines 13-16 in Figure 4) 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 Figure 4). 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 Figure 4 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.

4.7 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.

$$head\ canary = size \oplus buffer\ address \oplus head\ secret\ key \quad (1)$$

The tail canary is constructed in the same way, except that it uses a different secret key. For the canary verification, the guard thread firstly 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.

4.8 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 firstly 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:

$$free\ canary = size \oplus address \oplus head\ secret\ key \oplus free\ secret\ key \quad (2)$$

$$free\ canary = head\ canary \oplus free\ secret\ key \quad (3)$$

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.

4.9 Discussions

Leveraging shared memory. Our current implementation only uses global memory. In fact, shared memory is much faster than global memory, and it could be leveraged to improve the performance of GMOD, such as allocating byte array in shared memory. However, using shared memory to detect buffer overflow will reduce the availability of shared memory for user kernels. Given the limited capacity of shared memory, we do not use shared memory in GMOD.

Buffer overflow detection on unified memory. Unified memory is a memory address space accessible from both CPU and GPU. GPU memory buffer allocated on the unified memory should be able to be protected by the existing overflow detection mechanisms on CPU [13, 28, 34]. However, those mechanisms can cause large overhead, because of memory paging. Memory paging enables automatic page migration between host and GPU memories, providing an illusion of unified memory. The detection mechanisms on CPU and computation on GPU can concurrently access the memory buffer in unified memory, and hence cause frequent memory paging, which lose performance. GMOD does not cause memory paging, because the guard kernel runs on GPU (not on CPU). We leave the evaluation of GMOD for unified memory as our future work.

5 DESIGN EXTENSION

We slightly extend our design to detect buffer overflow for coarse-grained memory management. The design extension includes the following three parts.

First, we have a customized memory management function `cudaMallocN` to replace `cudaMalloc`. `cudaMallocN` allocates memory buffer with extra space to place canaries.

Table 1: Benchmarks for evaluating GMOD. *Num.ker* is the number of user kernels for evaluation. *Sum.Allocation* is total number of memory allocation in all kernels. *Sum.Allocation* is related to the number of user threads in a benchmark, so we report the maximum number of memory allocation in our evaluation.

Benchmark	Num.ker	Sum.Allocation
alloc-dealloc (ad) [27]	1	20K
alloc-cycle-dealloc (acd) [27]	5	100K
grid-point (gp) [26]	1	20K
add-string (ads) [26]	3	60K
random-graph (rg) [26]	1	60K
fft [23]	1	3
bfs [33]	26	3

Second, within the user kernel, the user needs to add one or more function calls `insert_buffer_info`, right before the kernel computation. Each `insert_buffer_info` fills in canaries and inserts buffer information for a memory buffer into a byte array. Note that `cudaMallocN` cannot place canaries and insert buffer information as `mallocN`, because it runs on CPU and cannot easily access GPU memory. `insert_buffer_info` adds overhead into the user kernel, but the overhead is small (we quantify it in the evaluation section) and `cudaMalloc` often has a few occurrences in an application.

Third, we have a customized memory management function `cudaFreeN` to replace `cudaFree`. `cudaFreeN` includes an implicit `cudaDeviceSynchronize` before freeing memory buffer. This is used to ensure that we do not free any coarse-grained memory buffer until the guard kernel and user kernels are done. Otherwise, a guard thread may access a memory buffer which is already released by `cudaFreeN`.

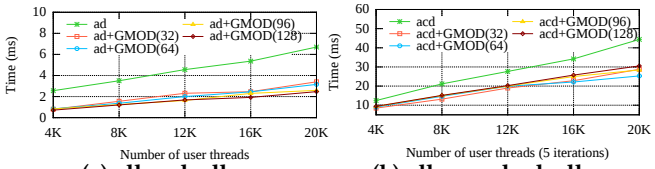
In general, GMOD has high transparency. To detect buffer overflow, the user needs to replace memory allocation and free APIs with our customized APIs. For coarse-grained memory management, the user also needs to add a few function calls within user kernels.

6 EVALUATION

6.1 Experimental Setup

Our experiments are performed on a machine with an Intel Xeon CPU E5-2609 (1.9 GHz), and an NVIDIA GM200 GeForce GTX TITAN X discrete GPU. The machine has Ubuntu 14.04.4 LTS and CUDA runtime 7.5.

We use five benchmarks and two micro-benchmarks to evaluate GMOD. The benchmarks are summarized in Table 1. Those benchmarks use the default kernel configurations, unless indicated otherwise. The benchmarks alloc-dealloc, alloc-cycle-dealloc, add-string, random-graph, and grid-point are called ad, acd, ads, rg, and gp for short in the figures in this section. Among the seven benchmarks, bfs



(a) alloc-dealloc **(b) alloc-cycle-dealloc**

Figure 7: Performance results for two stress tests (preliminary performance study). The numbers in parenthesis in the figures are the number of guard threads.

and fft have coarse-grained, dynamic memory management, and the other five benchmarks have fine-grained one. All results reported in this section are average values of 20 runs.

The reasons why we use those benchmarks are as follows. We use the two micro-benchmarks [27], because they have very intensive memory allocations that allow us to evaluate GMOD performance with stress testing. We use the three benchmarks (gp, ads and rg) from Halloc [26], because they support fine-grained memory allocation on GPU which can not be found in the common GPU benchmark suites, such as Parboil [23], Rodinia [6], SHOC [8] and NUPAR [25]. The fft from Parboil [23] and the bfs from [33] are representative benchmarks for coarse-grained memory allocation, because both of them have about 10 memory allocations and are complicated enough. In addition, their performance is sensitive to the disturbance from any co-run benchmark.

6.2 Preliminary Performance Study

As a preliminary performance study, we use two micro-benchmarks (alloc-dealloc and alloc-cycle-dealloc) that have very intensive memory allocations to evaluate GMOD under stress tests. Figures 7a and 7b shows the results.

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

Figure 7a 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 GMOD 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 on byte arrays happen. However, Figure 7b reveals that increasing the number of guard threads leads to worse performance in alloc-cycle-dealloc. We attribute this observation 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 can lose performance. We further discuss the effects of number of guard threads in the next section.

6.3 Sensitivity Study

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

Study on detection latency. The same as other tools to detect buffer overflow on CPU [1, 2, 34], GMOD does not provide real-time detection. To study the detection latency, we use five benchmarks. For each benchmark, we use different numbers of guard threads for GMOD execution. Figure 9 shows the estimated 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 to scan the byte array for once depends on the length of byte array. Since the length of 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 [24, 34].

Figure 9 shows that the detection latencies of almost all cases (38 out of 40 cases) are less than the execution time of user kernels, which demonstrates that our detection latency is short enough in most cases. Only two cases (grid-point and random-graph when the number of guard threads is 32) have detection latency longer than the execution time of user kernels, but increasing the number of guard threads (larger than 32 guard threads) makes detection latency shorter than the user kernel execution time.

Note that although GMOD cannot detect overflow before the user kernel finishes in a few cases, GMOD can still detect overflow for those cases after the user kernel finishes. In particular, when the user kernel finishes, the user buffer with overflow has been freed by a user thread using `freeN`. However, the user buffer with overflow still exists in GPU global memory, because of our design of the delayed free (Section 4.4). The guard thread can still detect overflow even after the user kernel finishes.

Study on user kernel performance. We study the impact of the guard thread number on user kernel performance. Figures 8a-8e shows the results with different number of guard threads and user threads. Table 2 reports average performance improvement (or degradation) for each guard thread number (32, 64, 96 and 128), based on Figures 8a-8e.

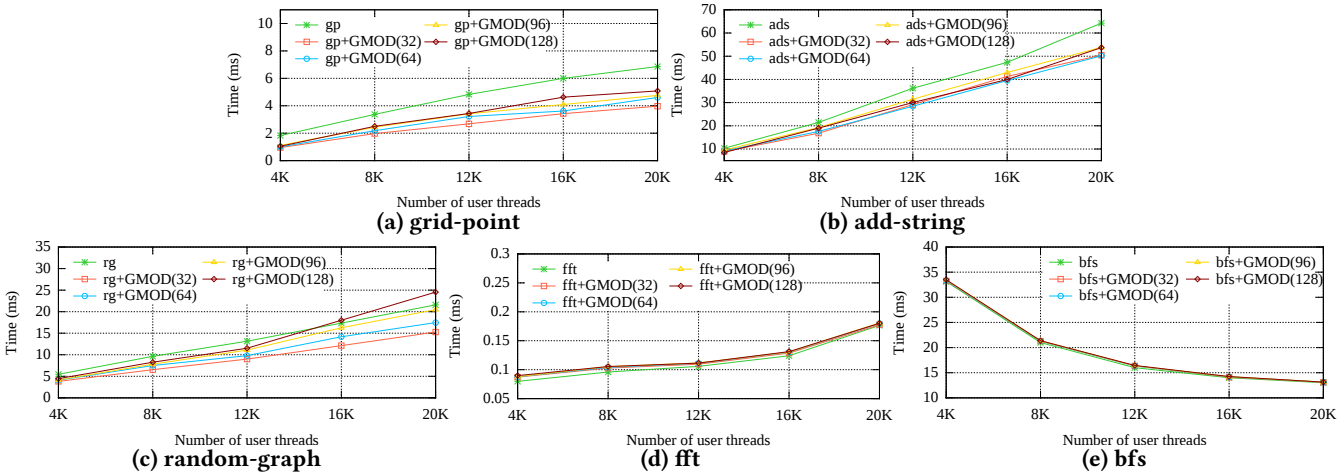


Figure 8: Performance (execution time) of five benchmarks with GMOD. The numbers in parenthesis in the figures are the number of guard threads.

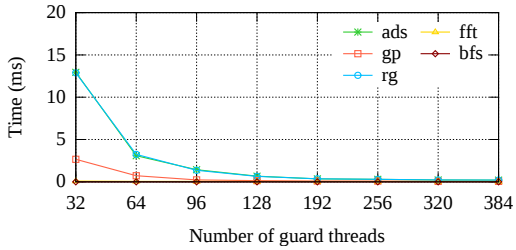


Figure 9: Average execution time of a guard thread to scan its byte array for once.

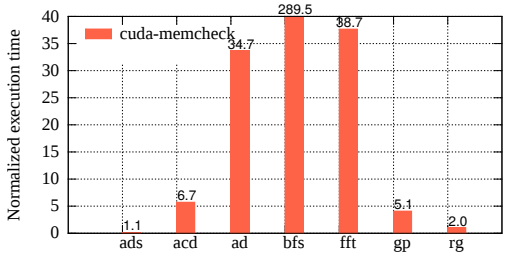


Figure 10: Performance evaluation of cuda-memcheck. Performance is normalized by the execution time without any overflow detection.

Table 2: Average performance improvement/loss for five benchmarks with GMOD. This table is based on Figures 8a-8e. The first row of the table is the number of guard threads. A positive percentage number represents performance improvement, while a negative one represents performance degradation.

	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%
fft	-4.8%	-6.0%	-6.7%	-7.1%
bfs	-1.1%	-1.2%	-1.5%	-1.7%

In general, GMOD causes small performance lose (less than 5%) in all cases, except fft (less than 12%). Among 100 evaluation cases shown in Figures 8a-8e, 60% of them have performance improvement, due to the asynchronous execution of `freeN` and ignorable performance overhead of GMOD. bfs and fft have performance loss with GMOD, because they use synchronous memory free (`cudaFree`). We also notice that using a larger number of guard threads, the performance loss in bfs and fft (or performance benefit in other benchmarks) due to GMOD becomes larger (or smaller). In summary, using 32 guard threads, the performance loss is 2.9% on average (up to 9.1%) 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 GMOD, 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 warp-based scheduling nature in GPU.

Comparing with the existing work. To show the performance benefit of GMOD, we evaluate the performance of related work, cuda-memcheck. Figure 10 shows the execution time of cuda-memcheck, normalized by the execution time without any overflow detection. In general, cuda-memcheck has rather large overhead (at least 10% and up to 289.5x), much larger than GMOD.

6.4 Evaluation of Benchmarks Co-Run

To study the performance of GMOD with co-run kernels, we choose any two benchmarks from the five benchmarks

for co-run. The benchmarks grid-point, add-string, random-graph, fft, and bfs use 4K, 6K, 8K, 10K and 12K user threads respectively, to enable sufficient co-run. In total, we have ten co-run cases. We enable co-run of those benchmarks based on CUDA stream.

Figure 12 shows the performance for those ten co-run cases. The performance of a co-run case is the execution time of finishing two benchmarks. In general, GMOD does not bring performance loss except the case of co-running fft and bfs (0.5% performance loss).

Memory cost of GMOD. For each user buffer, GMOD introduces 24 bytes for canaries and 9 bytes for storing buffer information in a 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 GMOD with a canary-based overflow detector on CPU [34] 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 GMOD.

6.5 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. We do not evaluate fft and bfs, because they do not use `mallocN`, and `cudaFreeN` in fft and bfs does not bring performance benefit. Note that the overhead of `cudaMallocN` and insert_buffer_info for coarse-grained memory management have been included and evaluated in Figure 8.

Figures 11a-11c show the results. The results show that the average overhead of our `mallocN` is about 10.9% (random-graph), 6.2% (add-string), and 5.1% (grid-point). In general, the customized malloc results in overhead less than 11%. 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 Figures 8a-8c.

Note that the overhead of using `CUDAmallocN` is typically much smaller than using `mallocN`, because `mallocN`, as a fine-grained memory allocation, can happen very frequently in the middle of user kernel execution, while `CUDAmallocN` only happens a couple of times before kernel execution.

6.6 Effectiveness of GMOD

We evaluate the effectiveness of GMOD to detect buffer overflow on GPU (Table 3). We use three experiments for our study. The three experiments are also used in the existing studies [9, 15] to demonstrate the existence of GPU buffer overflows. The benchmarks with buffer overflow in the existing work are open sourced [9]. We use GMOD to detect buffer overflow in those benchmarks.

Table 3: Using the three buffer overflow benchmarks to evaluate the effectiveness of GMOD

	Kernel time	Detection latency	Detected?
Single kernel (fine)	0.947 ms	0.0145 ms	yes
Sequential kernels (fine)	1.273 ms	0.0098 ms	yes
Concurrent kernels (fine)	10.818 ms	0.0121 ms	yes
Single kernel (coarse)	0.9185 ms	0.01203 ms	yes
Sequential kernels (coarse)	1.223 ms	0.01508 ms	yes
Concurrent kernels (coarse)	10.771 ms	0.016 ms	yes

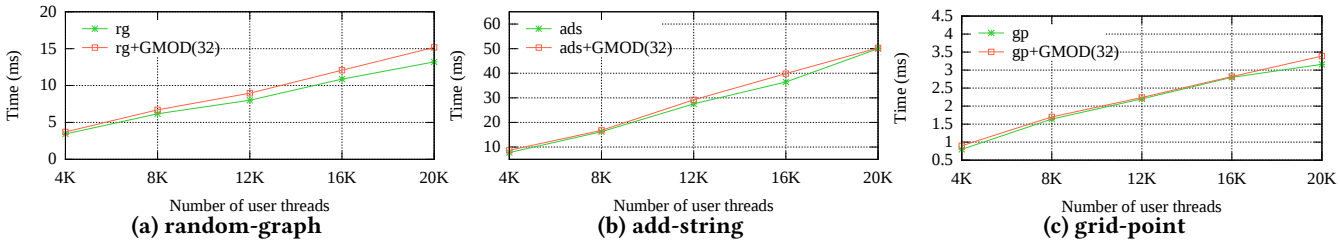
The first experiment evaluates the situation that the buffer overflow occurs in a single kernel. The second experiment considers GPU buffer overflow between two sequentially launched kernels. The third one evaluates how one kernel can influence the execution flow of another concurrently running kernel. The last two experiments are a typical usage mode of sharing GPU in cloud or HPC environment. The three experiments focus on fine-grained, dynamic memory management. To show the effectiveness of GMOD for coarse-grained, dynamic memory management, we replace `malloc` and `free` in the three experiments with cudaMalloc and cudaFree. In all experiments, once GMOD finds the overflow, GMOD stops the execution of user kernels and outputs overflow information such as the user address where overflow happens.

Table 3 shows the results. "kernel time" in the table is the execution time of the attacked user kernel. In all cases, GMOD successfully detects overflow and the detection latency is very short, much shorter than kernel execution time.

7 RELATED WORK

Buffer overflow detection on CPU. Many static analysis tools [13, 28] use bounds checking to detect buffer overflows by analyzing source code statically. This approach suffers from high false positive or false negative rate. Canary was firstly proposed in StackGuard [7], which tackles stack smashing attacks by placing a canary word before return address on stack. Address Space Layout Randomization (ASLR) [4] randomizes addresses of stack and heap variables for each execution, such that buffer overflow attacks can not be achieved reliably. Cruiser [34] is a concurrent heap buffer overflow detector on CPUs which is similar to GMOD but runs on CPUs.

Security issues on GPU. The study in [5] shows that adversaries can retrieve other processes's data stored in GPU memory by analyzing the memory dump of GPU devices. Maurice et. al [14] highlight possible information leakage of GPUs in virtualized and cloud environments. In [19], Pietro et al. present a detailed analysis of information leakage in



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

Figure 11: Study the performance impact of `mallocN`.

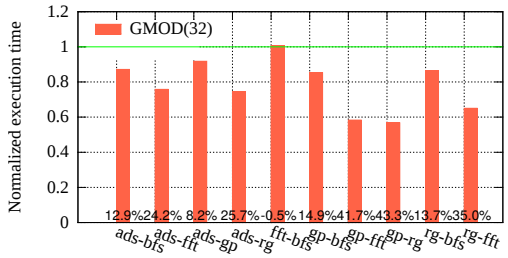


Figure 12: Performance (execution time) of co-run benchmarks with GMOD. Performance is normalized by that without GMOD.

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 proposed solutions to run multiple kernels from different users concurrently on GPUs. In [21], Ravi et al. present a framework to enable applications executing within virtual machines to transparently share one or more GPUs. Pai et al. [17] 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. Our GMOD complements the above work by considering GPU security.

GPU memory overflow. Miele [15] 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 [9], 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 [16], 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% [3].

clARMOR [10] is another GPU buffer overflow detector using a canary-based design. It offers runtime protection with reasonable overhead. Although both clARMOR and GMOD use the canary-based design, they are fundamentally different. There is no simple method to slightly change clARMOR to become GMOD. 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`). GMOD 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 detection on the performance of user kernels. GMOD introduces a series of techniques to avoid performance overhead.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we present the design and implementation of a dynamic GPU memory overflow detector GMOD, which performs always-on monitoring on dynamically allocated buffers from concurrent user kernels. GMOD can effectively identify buffer overflows with ignorable performance overhead. Because GMOD is based on canary, it can not detect unauthorized read access from untrusted users. We leave this as future work. It is also interesting and challenging to extend GMOD to more complex scenarios, where the GPU and CPU share virtual or physical memory space, and memory operations are issued from both sides simultaneously.

ACKNOWLEDGMENTS

This work is partially supported by the National Science Foundation of China under grants 61572179, 61772183, and 61602166, and U.S. National Science Foundation (CNS-1617967, CCF-1553645 and CCF-1718194). We thank anonymous reviewers for their valuable feedback.

REFERENCES

[1] 2000. S. Shield. Website. (2000). http://www.angelfire.com/sk/stackshield/.

[2] K. Avijit, P. Gupta, and D. Gupta. 2004. TIED, LibsafePlus: Tools for Runtime Buffer Overflow Protection. In *Proceedings of the 13th USENIX Security Symposium (SECURITY '04)*. USNIX Association, 45–56.

[3] T. M. Baumann and J. Gracia. 2013. Cudagrind: Memory Usage Checking for CUDA. In *Tools for High Performance Computing 2013, Proceedings of the 7th International Workshop on Parallel Tools for High Performance Computing*. Springer, 67–78.

[4] S. Bhatkar, D. C. DuVarney, and R. Sekar. 2003. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium (SECURITY '03)*. USENIX Association, 291–301.

[5] S. Breß, S. Kiltz, and M. Schäler. 2013. Forensics on GPU Coprocessing in Databases - Research Challenges, First Experiments, and Countermeasures. In *Proceeding of Workshop on Databases in Biometrics, Forensics and Security Applications (DBforBFS), BTW-Workshops*. Köllen-Verlag, 115–129.

[6] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, October 4-6, 2009, Austin, TX, USA*.

[7] C. Cowan. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, January 26-29, 1998*.

[8] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The Scalable Heterogeneous Computing (SHOC) benchmark suite.

[9] B. Di, J. H. Sun, and H. Chen. 2016. A Study of Overflow Vulnerabilities on GPUs. In *Proceedings of the 13th International Conference on Network and Parallel Computing (NPC '16)*. Springer, 103–115.

[10] C. Erb, M. Collins, and J. L. Greathouse. 2017. Dynamic buffer overflow detection for GPGPUs. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*. ACM, Austin, TX, USA, 61–73.

[11] GMOD. 2018. GMOD. Website. (2018). https://github.com/aimlab/GMOD.

[12] IBM. 2006. ProPolice detector. https:www.trl.ibm.com/projects/security/ssp/. (2006).

[13] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. L. Wang. 2002. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (USENIX ATC '02)*. USENIX Association, Berkeley, CA, USA, 275–288.

[14] C. Maurice, C. Neumann, O. Heen, and A. Francillon. 2014. Confidentiality Issues on a GPU in a Virtualized Environment. In *Proceedings ot the 18th International Conference on Financial Cryptography and Data Security (FC '14)*. 119–135.

[15] A. Miele. 2016. Buffer overflow vulnerabilities in CUDA: a preliminary analysis. *Journal of Computer Virology and Hacking Techniques* 12, 2 (2016), 113–120.

[16] NVIDIA. 2017. CUDA-MEMCHECK. (2017). https://developer.nvidia.com/cuda-memcheck.

[17] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. 2013. Improving GPGPU Concurrency with Elastic Kernels. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, 407–418.

[18] B. Perens. 1987. Electric Fence. Website. (1987). http://linux.die.net/man/3/efence.

[19] R. D. Pietro, F. Lombardi, and A. Villani. 2016. CUDA Leaks: A Detailed Hack for CUDA and a (Partial) Fix. *ACM Transactions Embedded Computing Systems* 15, 1 (2016), 15:1–15:25.

[20] The PaX project. 2017. http://pax.grsecurity.net/. (2017).

[21] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar. 2011. Supporting GPU Sharing in Cloud Environments with a Transparent Runtime Consolidation Framework. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing (HPDC '11)*. ACM, 217–228.

[22] M. Song, Y. Hu, H. Chen, and T. Li. 2017. Towards Pervasive and User Satisfactory CNN across GPU Microarchitectures. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[23] J. A. Stratton, C. Rodrigues, I. J. Sung, N. Obeid, L. W. Chang, N. Anssari, G. D. Liu, and W. W. Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* (2012).

[24] D. H. Tian, Q. Zeng, D. H. Wu, P. Liu, and C. Z. Hu. 2012. Kruiser: Semi-synchronized Non-blocking Concurrent Kernel Heap Buffer Overflow Monitoring. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*.

[25] Yash Ukidave, Fanny Nina Paravecino, Leiming Yu, Charu Kalra, Amir Momeni, Zhongliang Chen, Nick Materise, Brett Daley, Perhaad Mistry, and David R. Kaeli. 2015. NUPAR: A Benchmark Suite for Modern GPU Architectures.

[26] A. V.Adinetz. 2014. Halloc GPU memory allocator. (2014). https://github.com/canonizer/halloc.

[27] M. Vinkler and V. Havran. 2015. Register Efficient Dynamic Memory Allocator for GPUs. *Computer Graphics Forum* 8 (2015), 143–154.

[28] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. 2000. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of Network and Distributed System Security Symposium (NDSS '00)*. Internet Society, 3–17.

[29] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. 2014. Concurrent Analytical Query Processing with GPUs. *Proc. VLDB Endow.* 7, 11 (2014).

[30] wiki. 2017. Code Red. https://en.wikipedia.org/wiki/Code_Red_(computer_worm). (2017).

[31] wiki. 2017. Morris Worm. https://en.wikipedia.org/wiki/Morris_worm. (2017).

[32] wiki. 2017. Slammer. https://en.wikipedia.org/wiki/SQL_Slammer. (2017).

[33] H. C. Wu, D. Li, and M. Becchi. 2016. Compiler-Assisted Workload Consolidation for Efficient Dynamic Parallelism on GPU. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*. 534–543.

[34] Q. Zeng, D. H Wu, and P. Liu. 2011. Cruiser: concurrent heap buffer overflow monitoring using lock-free data structures. In *ACM Conference on Programming Language Design and Implementation*.