A Study of Overflow Vulnerabilities on GPUs

Bang Di, Jianhua Sun(✉), and Hao Chen

College of Computer Science and Electronic Engineering,
Hunan University, Changsha 410082, China
{dibang,jhsun,haochen}@hnu.edu.cn

Abstract. GPU-accelerated computing gains rapidly-growing popularity in many areas such as scientific computing, database systems, and cloud environments. However, there are less investigations on the security implications of concurrently running GPU applications. In this paper, we explore security vulnerabilities of CUDA from multiple dimensions. In particular, we first present a study on GPU stack, and reveal that stack overflow of CUDA can affect the execution of other threads by manipulating different memory spaces. Then, we show that the heap of CUDA is organized in a way that allows threads from the same warp or different blocks or even kernels to overwrite each other's content, which indicates a high risk of corrupting data or steering the execution flow by overwriting function pointers. Furthermore, we verify that integer overflow and function pointer overflow in *struct* also can be exploited on GPUs. But other attacks against format string and exception handler seems not feasible due to the design choices of CUDA runtime and programming language features. Finally, we propose potential solutions of preventing the presented vulnerabilities for CUDA.

Keywords: GPGPU · CUDA · Security · Buffer overflow

1 Introduction

Graphics processing units (GPUs) were originally developed to perform complex mathematical and geometric calculations that are indispensable parts of graphics rendering. Nowadays, due to the high performance and data parallelism, GPUs have been increasingly adopted to perform generic computational tasks. For example, GPUs can provide a significant speed-up for financial and scientific computations. GPUs also have been used to accelerate network traffic processing in software routers by offloading specific computations to GPUs. Computation-intensive encryption algorithms like AES have also been ported to GPU platforms to exploit the data parallelism, and significant improvement in throughput was reported. In addition, using GPUs for co-processing in database systems, such as offloading query processing to GPUs, has also been shown to be beneficial.

With the remarkable success of adopting GPUs in a diverse range of real-world applications, especially the flourish of cloud computing and advancement

G.R. Gao et al. (Eds.): NPC 2016, LNCS 9966, pp. 103–115, 2016.
DOI: 10.1007/978-3-319-47099-3_9

in GPU virtualization [1], sharing GPUs among cloud tenants is increasingly becoming norm. For example, major cloud vendors such as Amazon and Alibaba both offer GPU support for customers. However, this poses great challenges in guaranteeing strong isolation between different tenants sharing GPU devices.

As will be discussed in this paper, common well-studied security vulnerabilities on CPUs, such as the stack and heap overflow and integer overflow, exist on GPUs too. Unfortunately, with high concurrency and lacking effective protection, GPUs are subject to greater threat. In fact, the execution model of GPU programs consisting of CPU code and GPU code, is different from traditional programs that only contains host-side code. After launching a GPU *kernel* (defined in the following section), the execution of the GPU code is delegated to the device and its driver. Therefore, we can know that the GPU is isolated from the CPU from the perspective of code execution, which means that the CPU can not supervise its execution. Thus, existing protection techniques implemented on CPUs are invalid for GPUs. On the other hand, the massively parallel execution model of GPUs makes it difficult to implement efficient solutions tackling security issues. Unfortunately, despite GPU's pervasiveness in many fields, a thorough awareness of GPU security is lacking, and the security of GPUs are subject to threat especially in scenarios where GPUs are shared, such as GPU clusters and cloud.

From the above discussion, we know that GPUs may become a weakness that can be exploited by adversaries to execute malicious code to circumvent detection or steal sensitive information. For example, although GPU-assisted encryption algorithms achieve high performance, information leakage such as private secrete key has been proven to be feasible [2]. In particular, in order to fully exert the computing power of GPUs, effective approaches to providing shared access to GPUs has been proposed in the literature [3,4]. However, without proper mediation mechanisms, shared access may cause information leakage as demonstrated in [2]. Furthermore, we expect that other traditional software vulnerabilities on CPU platforms would have important implications for GPUs, because of similar language features (CUDA programming language inherits C/C++). Although preliminary experiments has been conducted to show the impact of overflow issues on GPU security [5], much remains unclear considering the wide spectrum of security issues. In this paper, we explore the potential overflow vulnerabilities on GPUs. To the best of our knowledge, it is the most extensive study on overflow issues for GPU architectures. Our evaluation was conducted from multiple aspects, which not only includes different types of attacks but also considers specific GPU architectural features like distinct memory spaces and concurrent kernel execution. Although we focus on the CUDA platform, we believe the results are also applicable to other GPU programming frameworks.

The rest of this paper is organized as follows. Section 2 provides necessary background about the CUDA architecture. In Sect. 3, we perform an extensive evaluation on how traditional overflow vulnerabilities can be implemented on GPUs to affect the execution flow. Possible countermeasures are discussed in Sect. 4. Section 5 presents related work, and Sect. 6 concludes this paper.

2 Background on CUDA Architecture

CUDA is a popular general purpose computing platform for NVIDIA GPUs. CUDA is composed of device driver (handles the low-level interactions with the GPU), the runtime, and the compilation tool-chain. An application written for CUDA consists of host code running on the CPU, and device code typically called *kernels* that runs on the GPU. A running kernel consists of a vast amount of GPU *threads*. Threads are grouped into *blocks*, and blocks are grouped into *grids*. The basic execution unit is *warp* that typically contains 32 threads. Each thread has its own program counters, registers, and local memory. A block is an independent unit of parallelism, and can execute independently of other thread blocks. Each thread block has a private per-block shared memory space used for inter-thread communication and data sharing when implementing parallel algorithms. A grid is an array of blocks that can execute the same kernel concurrently. An entire grid is handled by a single GPU.

The GPU kernel execution consists of the following four steps: (i) input data is transferred from the host memory to GPU memory through the DMA; (ii) a host program instructs the GPU to launch a kernel; (iii) the output is transferred from the device memory back to the host memory through the DMA.

CUDA provides different memory spaces. During execution, CUDA threads may access data from multiple memory spaces. Each thread maintains its own private local memory that actually resides in global memory. Automatic variables declared inside a kernel are mapped to local memory. The on-chip shared memory is accessible to all the threads that are in the same block. The shared memory features low-latency access (similar to L1 cache), and is mainly used for sharing data among threads belonging to the same block. The global memory (also called device memory) is accessible to all threads, and can be accessed by both GPU and CPU. There are two read-only memory spaces accessible by all threads, i.e. constant and texture memory. Texture memory also offers different addressing models, as well as data filtering, for some specific data formats. The global, constant, and texture memory spaces are optimized for different memory usages, and they are persistent across kernel launches by the same application.

3 Empirical Evaluation of GPU Vulnerabilities

In this section, we first introduce the testing environment. Then, we discuss specific vulnerabilities for stack overflow, heap overflow, and others respectively, with a focus on the heap overflow because of its potential negative impact and significance in scenarios where multiple users share GPU devices. Due to the proprietary nature of the CUDA platform, we can only experimentally confirm the existence of certain vulnerabilities. And further exploration about inherent reasons and such issues is beyond the scope of this paper, which may require a deeper understanding of the underlying implementation of CUDA framework and hardware device intricacy.

```
  typedef unsigned long(*pFdummy)(void);
__device__ __noinline__ unsigned long normal1() {
  printf("Normal\n");
  return 0;
}
__device__ __noinline__ unsigned long malicious() {
  printf("Attack!\n");
  return 0;
}
__device__ int overf[100];
//==========================================================
  for(int i = 0; i < length; i++) {overf[i] = input[i];}
  unsigned int buf[16];
  pFdummy fp[8];
  fp[0]=normal1; fp[1]=normal2; fp[2]=normal3; fp[3]=normal4;
  fp[4]=normal5; fp[5]=normal6; fp[6]=normal7; fp[7]=normal8;
  for(int i = 0; i < length; i++) {buf[i] = overf[i];}
  fp[5];
```

Fig. 1. A code snippet of stack overflow in device.

3.1 Experiment Setup

The machine conducting the experiment has a Intel Core i5-4590 CPU clocked at 3.30 GHz, and the GPU is NVIDIA GeForce GTX 750Ti (Maxwell architecture) that has compute capability 5.0. The operating system is Ubuntu 14.04.4 LTS (64 bit) with CUDA 7.5 installed. *nvcc* is used to compile CUDA code, and NVIDIA visual profiler is adopted as a performance profiling tool. CUDA-GDB allows us to debug both the CPU and GPU portions of the application simultaneously. The source code of all implemented benchmarks is publicly available at https://github.com/aimlab/cuda-overflow.

3.2 Stack Overflow

In this section, we investigate the stack overflow on GPUs by considering different memory spaces that store adversary-controlled data, and exploring all possible interactions among threads that are located in the same block, or in different blocks of the same kernel, or in distinct kernels.

The main idea is as follows. The adversary formulates malicious input data that contains the address of a malicious function, and assign it to variable a that is defined in global scope. Two stack variables b and c are declared in a way to make their addresses adjacent. If we use a to assign values to b to intentionally overflow b and consequently corrupt the stack variable c that stores function pointers. Then, when one of the function pointers of c is invoked, the execution flow would be diverted to the adversary-controlled function. Note that there is a difference of the stack between the GPU and CPU. In fact, the storage allocation of GPU stack is similar to the heap, so the direction of overflow is from low address to high address.

We explain how a malicious kernel can manipulate a benign kernel's stack with an illustrating example that is shown in Fig. 1. In the GPU code, we define 9 functions containing 1 malicious function (used to simulate malicious behavior) and 8 normal functions (only one is shown in Fig. 1, and the other 7 functions are the same as the function *normal1* except the naming). The *__device__* qualifier declares a function that is executed on the device and callable from the device only. The *__noinline__* function qualifier can be used as a hint for the compiler not to inline the function if possible. The array *overf[100]* is declared globally to store data from another array *input[100]* that is controlled by the malicious kernel. Given the global scope, the storage of *overf[100]* is allocated in the global memory space, indicating both the malicious kernel and benign kernel can access. In addition, two arrays named *buf* and *fp* are declared one after another on the stack to ensure that their addresses are consecutively assigned. The *fp* stores function pointers that point to the normal functions declared before, and the data in *overf[100]* is copied to *buf* (shown at line 17) to trigger the overflow. The *length* variable is used to control how many words should be copied from *overf* to *buf* (shown at line 17). It is worth noting that the line 12 is only executed in the malicious kernel to initialize the *overf* buffer. If we set *length* to 26 and initialize *overf* with the value1 **0x590** (address of the *malicious* function that can be obtained using *printf("%p",malicious)* or CUDA-GDB [5]), the output at line 18 would be string "**Normal**". This is because with value 26, we can only overwrite the first 5 pointers in *fp* (*sizeof(buf) + sizeof(pFdummy) * 5 == 26*). However, setting *length* to 27 would cause the output at line 18 to be "**Attack!**", indicating that *fp[5]* is successfully overwritten by the address of the *malicious* function. This example demonstrates that current GPUs have no mechanisms to prevent stack overflow like stack canaries on the CPU counterpart (Fig. 2).

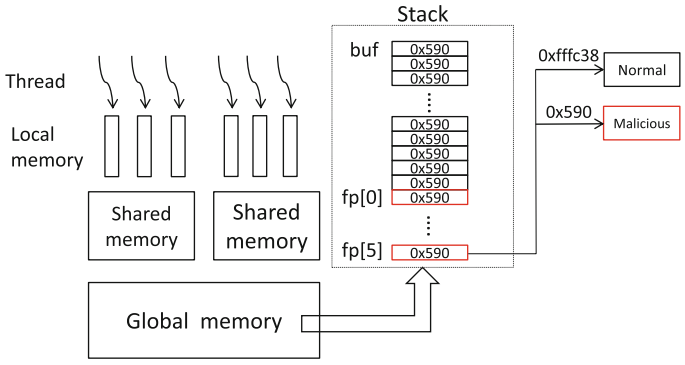

Fig. 2. Illustration of stack overflow.

It is straightforward to extend our experiments to other scenarios. For example, by locating the array *overf* in the shared memory, we can observe that the attack is feasible only if the malicious thread and benign thread both reside in the same block. While if *overf* is in the local memory, other threads haves

no way to conduct malicious activities. In summary, our evaluation shows that attacking a GPU kernel based on stack overflow is possible, but the risk level of such vulnerability depends on specific conditions like explicit communication between kernels.

3.3 Heap Overflow

In this section, we study a set of heap vulnerabilities in CUDA. We first investigate the heap isolation on CUDA GPUs. Then, we discuss how to corrupt locally-allocated heap data when the malicious and benign threads co-locate in the same block. Finally, we generalize the heap overflow to cases where two kernels are run sequentially or concurrently.

```
1  // ============== a virtual table of the device code ================
2  class Vtable {
3  public:
4    __device__ virtual unsigned long v1() {printf("Normal\n");return 0;}
5    __device__ virtual unsigned long v2() {printf("Normal\n");return 0;}
6    __device__ virtual unsigned long v3() {printf("Normal\n");return 0;}
7    __device__ virtual unsigned long v4() {printf("Normal\n");return 0;}
8  };
9  //====================== malicious function ==========================
10 __device__ __noinline__ unsigned long malicious() {
11   printf("Attack!\n");
12   return 0;
13 }
14 //============= a snippet code of memory isolation of heap ===========
15   __shared__ unsigned long *buf;
16   if(threadIdx.x == 0)
17   buf = (unsigned long *) malloc(sizeof(unsigned long) * 8);
18   Vtable *fp = new Vtable;
19   if(threadIdx.x == 0)
20     for(int i = 0; i < length; i++) {buf[i] = input[i];}
21   if(threadIdx.x == 1)
22     printf("%lx", buf[0]);
23 //============ a snippet code of exploiting of 'global' heap =========
24   unsigned long *buf;
25   buf = (unsigned long *) malloc(sizeof(unsigned long) * 8);
26   Vtable *fp = new Vtable;
27   printf("malicious %p\n", malicious);
28   for(int i = 0; i < length; i++) {buf[i] = input[i];}
29   res=fp->v1(); res=fp->v2(); res=fp->v3(); res=fp->v4();
```

Fig. 3. A code snippet of heap overflow

Heap Isolation. Similar to the description of stack overflow, we also use a running example to illustrate heap isolation from two aspects. First, we consider the case of a single kernel. As shown in Fig. 3 (from line 15 to 22), suppose we have two independent threads t_1 and t_2 in the same block, and a pointer variable *buf* is defined in the shared memory. We can obtain similar results when *buf* is defined in the global memory. For clarity, we use buf_1 and buf_2 to represent the

buf in t_1 and t_2. buf_1 is allocated by calling *malloc* as shown at lines 16 and 17. Our experiments show that t_2 can always access buf_1 (line 21 to 22) unless *buf* is defined in the local scope. Second, we consider the case of two kernels (not shown in the figure). Kernel *a* allocates memory space for *buf*, and assigns the input value to it. If *a* returns without freeing the memory of *buf*, another kernel *b* can always read the content in *buf* if *b* also has a variable *buf* defined in either shared memory or global memory (no memory allocation for *buf* in *b*). This is because the GPU assigns the same address to *buf* for *b*, which makes it possible to access the not freed content of *buf* in *b*. In summary, for globally-defined heap pointer, the memory it points to can be freely accessed by threads that are not the original allocator. It is not the case for locally-defined heap pointers, but it may still be possible if we can successfully guest the addresses of local heap pointers (we leave this to future work). Most importantly, when a heap pointer is globally visible and the corresponding memory is not properly managed (freed), arbitrary memory accesses across kernels would be possible.

Heap Exploitation. In this experiment, we present that because the heap memory for different threads or kernels is allocated contiguously, overflowing one thread's local buffer may lead to the corruption of another thread's heap data.

We first consider heap memory allocated in local scope. As shown in Fig. 3 (from line 24 to 29), like before, suppose we have two threads t_1 and t_2 in the same block. For t_1, we use *malloc* and *new* to allocate memory for buf_1 and fp_1 respectively (we use these notations to clarify our discussion). fp_1 just stores the **start address** of the four virtual functions (the addresses is contained in the VTABLE). buf_1 and fp_1 are declared locally. t_2 is the same as t_1. After initializing t_1 and t_2, the memory layout of buf_1, fp_1, buf_2, and fp_2 looks like that shown in Fig. 4. Assume t_2 has malicious intention. The *input* in t_1 consists of normal data, and the *input* in t_2 consists of four addresses of the *malicious* function (**0x138**), and the remaining contents of *input* are the address of buf_2 (**0x50263f920**). When *length* is less than 11 (not 9 due to alignment), both t_1 and t_2 will print the string "**Normal**". However, when *length* is set to 11, the virtual table address in fp_1 would be modified to the start address of buf_2 where the four addresses of the *malicious* function are stored. So the output of t_1 will be the string "**Attack!**". When *length* is set to 21 (this value is relative to fp_1), both t_1 and t_2 will invoke the *malicious* function. Similarly, assuming t_1 is the malicious thread, both t_1 and t_2 will output "**Normal**" when *length* is less than 21. By assigning 21 to *length*, only t_1 will print "**Attack!**". And when the value of *length* is 31, both t_1 and t_2 will invoke the *malicious* function.

Based on the analysis above, we can conclude that the memory allocated from the heap in CUDA is globally accessible from different GPU threads without proper access control, and no protection is provided by the runtime system to prevent buffer overflow of heap-allocated memory. The addresses of heap pointers are often guessable, which makes it easy for a adversary to conduct attacks. Because of the closed-source nature of CUDA, further investigation about the implementation-level details is left as future work. In the following, we extend our analysis to the scenario where two kernels are considered when experimenting buffer overflow.

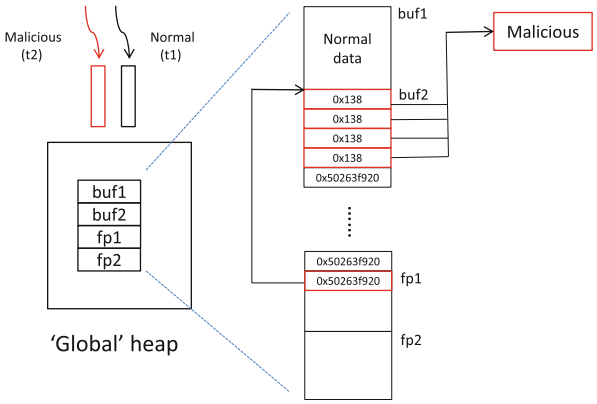

Fig. 4. A heap overflow: t_1 and t_2 represent the benign and malicious thread respectively.

Heap Exploitation Between Kernels. First, we consider the overflow between sequentially launched kernels. The experiment setup is similar to the previous section except that the host launches serially two kernels, $kernel_1$ (launched first) and $kernel_2$. $kernel_1$ simulates behaviors of an adversary and initializes the thread t_1. $kernel_2$ represents a benign user. t_1's input data consist of four addresses of malicious function, and its remaining contents are the address of buf_1. $kernel_1$ intentionally overflows buf_1 by using a large value of $length_1$, and then terminates. $kernel_2$ allocates memory for buf_2 and fp_2, but does not assign any value to them. When accessing one element in fp_2 as a function call, we will observe the output of string "**Attack!**". This is because the GPU assigns the same addresses of pointer variables to the second kernel, and the contents used in the first kernel are remained in GPU memory unchanged.

Second, we analyze the situation of two concurrently running kernels as shown in Fig. 5. This is the typical case of sharing GPUs in cloud or cluster environment, where different users may access the GPU at the same time. $kernel_1$ and $kernel_2$ must be in different streams. *cudaMemcpyAsync* is called to receive data from the host that allocates *page-locked* memory. The *sleep()* function is used to perform a simple synchronization between kernels to make our experiments more deterministic. t_1 in $kernel_1$ mimics a malicious user's behavior. When buf_2 and fp_2 are initialized (from line 9 to 12) and at the same time t_1 has finished the execution of the *for* loop (at line 20), we will observe the output of string "**Attack!**" from t_2 if it continues to run after the pause at line 13. Based on these observations, we can conclude that multiple kernels, regardless of serially or concurrently running, have the opportunity to intentionally manipulate the heap memory of each other.

```
1  //==================== sleep function ======================
2  __device__ void sleep(int64_t num_cycles) {
3    int64_t cycles = 0;
4    int64_t start = clock64();
5    while (cycles < num_cycles)
6            cycles = clock64() - start;
7  }
8  //==================== normal kernel ======================
9    unsigned long *buf;
10   buf = (unsigned long *) malloc(sizeof(unsigned long) * 8);
11   Vtable *fp = new Vtable;
12   for(int i = 0; i < length; i++) {buf[i] = input[i];}
13   sleep(10000000);
14   res=fp->v1(); res=fp->v2(); res=fp->v3(); res=fp->v4();
15 //==================== malicious kernel ====================
16   unsigned long *buf;
17   buf = (unsigned long *) malloc(sizeof(unsigned long) * 8);
18   Vtable *fp = new Vtable;
19   sleep(1000000);
20   for(int i = 0; i < length; i++) {buf[i] = input[i];}
21   res=fp->v1(); res=fp->v2(); res=fp->v3(); res=fp->v4();
```

Fig. 5. A code snippet of concurrent kernel execution.

3.4 Other Vulnerabilities

In this section, we discuss the issues of *struct* and integer overflow that are demonstrated in one example. In this case, the attacker can exploit the characteristics of integer operations. Because of the two's complementary representation of integers, integer operations may produce undesirable results when an arithmetic operation attempts to produce a numeric value that is too large to be representable with a certain type.

In this experiment (Fig. 6), we define two variables *input[10]* and *length*, which stores user data and data size respectively. The device code for functions *normal* and *malicious* are the same as defined in Fig. 5. In addition, we define a *struct unsafe*, which contains an array *buf[6]* and a function pointer of type *normal*. The *init()* (line 6) function is used to initialize the structure defined above. The *if* statement (line 12) performs array-bounds check to prevent out-of-bound access. Suppose that the address of the *malicious* function is **0x43800000438** that is assigned to *input[10]* as input data by a malicious user. The variable *length* whose type is **unsigned char** is set to 0 by the attacker. The *if* branch would be executed because the value of *length* (0) is smaller than 6. But, the value of *length* will be 255 after it is decremented by one, which causes that the assignment at line 15 overflows the array *buf* and corrupt the function pointer in *struct unsafe*. This experiment shows that *struct* and integers can both be exploited in CUDA. In particular, both overflow and underflow of integer arithmetic operation are possible, which exposes more opportunities for the adversaries.

Format string and exception handling vulnerabilities have been studied on CPUs. However, our experiments show that they currently are not exploitable

```
// ================== a snippet of the device code ==================
struct unsafe {
  unsigned long buf[6];
  void (*normal)();
};
__device__ __noinline__ void init(struct unsafe *data) {
  data->normal = normal;
}
__global__ void test_kernel(unsigned long *input, unsigned char length) {
  struct unsafe cu;
  init(&cu);
  if (length < 6) {
    length = length - 1;
    for (int i = 0; i < length; i++)
      cu.buf[i] = input[i];
  }
  cu.normal();
}
```

Fig. 6. A snippet code of *struct* and integer vulnerabilities

on the GPU due to the limited support in CUDA. For format string, the formatted output is only supported by devices with compute capability 2.x and higher. The in-kernel *printf* function behaves in a similar way to the counterpart in the standard C-library. In essence, the string passed in as format is output to a stream on the host, which makes it impossible to conduct malicious behavior through *printf* to expose memory errors on the GPU. For the exception handling in C++, it is only supported for the host code, but not the device code. Therefore, we can not exploit the security issues of exception handling on CUDA GPUs currently.

4 Discussions and Countermeasures

This section discusses potential countermeasures that can prevent or restrict the impact of the described vulnerabilities. Basically, most discussed weaknesses have limited impact on current GPUs if considered from the security point of view. For example, stack overflow and in-kernel heap overflow can only corrupt the memory data from within the same kernel. It is difficult to inject executable code into a running kernel to change its control flow under the GPU programming model. Of course, it may not be the case when more functionalities are integrated to make developing GPU applications more like the CPU counterparts [6]. However, exploitable vulnerabilities for concurrent kernels pose real and practical risks on GPUs, and effective defenses are required for a secure GPU computing environment.

For security issues of the heap, calling the *free()* function, to a certain extent, can offer necessary protection for heap overflows. When an attacker wants to exploit vulnerabilities to corrupt the heap memory that belongs to other threads and has been freed, error message is reported and the application is terminated.

But this is hard to guarantee in environments where multiple kernels are running concurrently. The bounds check can be used to prevent overflow for both the stack and heap, but the performance overhead is non-trivial if frequently invoked at runtime. In addition, integer overflow may be leveraged to bypass the bounds check. Therefore, a multifaceted solution is desired to provide full protection against these potential attacks. For example, standalone or hybrid dynamic and static analysis approaches can be designed to detect memory and integer overflows. Ideally, such protections should be implemented by the underlying system such as the runtime system, compiler, or hardware of GPU platforms, as many countermeasures against memory errors on CPUs, such as stack canaries, address sanitizers, and address randomization, have been shown to be effective and practical. Unfortunately, similar countermeasures are still missing on GPUs.

CUDA-MEMCHECK [7] can identify the source and cause of memory access errors in GPU code, so it can detect all the heap related overflows in our experiments. But it fails to identify the stack overflows. CUDA-MEMCHECK is effective as a testing tool, but the runtime overhead makes it impractical to be deployed in production environments. Most importantly, when multiple mutually untrusted users share a GPU device, dynamic protection mechanisms are indispensable. Thus, we argue that research endeavors should be devoted to the design of efficient approaches to preventing security issues for GPUs.

A wide range of defensive techniques has been proposed to prevent or detect buffer overflows on CPUs. Compiler-oriented techniques including canaries, bounds checking, and tagging are especially useful in our context. Given that CUDA's backend is based on the LLVM framework and Clang already supports buffer overflow detection (AddressSanitizer) through compiler options, we believe these existing tools can be leveraged to implement efficient defenses against GPU-based overflow vulnerabilities.

5 Related Work

Recently, using GPUs in cloud environment has been shown to be beneficial. The authors of [4] present a framework to enable applications executing within virtual machines to transparently share one or more GPUs. Their approach aims at energy efficiency and do not consider the security issues as discussed in this work. For GPU-assisted database systems, the study in [8] shows that data in GPU memory is retrievable by other processes by creating a memory dump of the device memory. In [9], it is demonstrated feasible to recover user-visited web pages in widely-used browsers such as Chromium and Firefox, because these web browsers rely on GPUs to render web pages but do not scrub the content remained on GPU memory, leading to information leakage. The paper [10] highlights possible information leakage of GPUs in virtualized and cloud environments. They find that GPU's global memory is zeroed only when ECC (Error Correction Codes) is enabled, which poses high risk of private information exposure when ECC is disabled or not available on GPUs. In [2], the authors present a detailed analysis of information leakage in CUDA for multiple memory spaces

including the global memory, shared memory, and register. A real case study is performed on a GPU-based AES encryption implementation to reveal the vulnerability of leaking private keys.

However, all existing studies on GPU security have less considered vulnerabilities that are more extensively studied on CPUs. The paper [5] presents a preliminary study of buffer overflow vulnerabilities in CUDA, but the breadth and depth are limited as compared to our work. For example, we put more emphasis on heap overflows between concurrently running kernels, which we believe deserves special attentions in future research of securing shared GPU access.

6 Conclusion

In this paper, we have investigated the weakness of GPUs under malicious intentions with a focus on the CUDA platform. We believe the experiments conducted in this work are also applicable to other GPU frameworks such as OpenCL to reveal the potential security vulnerabilities. Particularly, we have confirmed the existence of stack and heap overflows through a diversified set of experiments, and also uncovered how integer overflow can be used in overwriting a function pointer in a *struct*. Other security issues such as format string and exception handling are also investigated. Although direct exploitation of these potential vulnerabilities is not feasible on current CUDA platform, care must be taken when developing applications for future GPU platforms, because GPU programming platforms are evolving with a fast pace. We hope this study can not only disclose real security issues for GPUs but stimulate future research on this topic especially for scenarios where GPU devices are shared among untrusted users.

Acknowledgment. This research was supported in part by the National Science Foundation of China under grants 61272190, 61572179 and 61173166.

References

1. Shi, L., Chen, H., Sun, J., Li, K.: vCUDA: GPU-accelerated high-performance computing in virtual machines. IEEE Trans. Comput. **61**(6), 804–816 (2012)
2. Pietro, R.D., Lombardi, F., Villani, A.: CUDA leaks: a detailed hack for CUDA and a (partial) fix. ACM Trans. Embedded Comput. Syst. **15**(1), 15 (2016)
3. Pai, S., Thazhuthaveetil, M.J., Govindarajan, R.: Improving GPGPU concurrency with elastic kernels. In: Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, 16–20 March 2013, pp. 407–418 (2013)
4. Ravi, V.T., Becchi, M., Agrawal, G., Chakradhar, S.T.: Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework. In: Proceedings of the 20th ACM International Symposium on High Performance Distributed Computing, HPDC 2011, San Jose, CA, USA, 8–11 June 2011, pp. 217–228 (2011)

5. Miele, A.: Buffer overflow vulnerabilities in CUDA: a preliminary analysis. J. Comput. Virol. Hacking Techn. **12**(2), 113–120 (2016)
6. Silberstein, M., Ford, B., Keidar, I., Witchel, E.: GPUfs: integrating a file system with GPUs. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS 2013, pp. 485–498. ACM (2013)
7. NVIDIA: CUDA-MEMCHECK. https://developer.nvidia.com/cuda-memcheck
8. Breß, S., Kiltz, S., Schäler, M.: Forensics on GPU coprocessing in databases - research challenges, first experiments, and countermeasures. In: Datenbanksysteme für Business, Technologie und Web (BTW), - Workshopband, 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 11–15 March 2013, Magdeburg, Germany. Proceedings, pp. 115–129 (2013)
9. Lee, S., Kim, Y., Kim, J., Kim, J.: Stealing webpages rendered on your browser by exploiting GPU vulnerabilities. In: 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, 18–21 May 2014, pp. 19–33 (2014)
10. Maurice, C., Neumann, C., Heen, O., Francillon, A.: Confidentiality issues on a GPU in a virtualized environment. In: Christin, N., Safavi-Naini, R. (eds.) FC 2014. LNCS, vol. 8437, pp. 119–135. Springer, Heidelberg (2014). doi:10.1007/978-3-662-45472-5_9