2022 IEEE 40th International Conference on Computer Design (ICCD) | 978-1-6654-6186-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICCD56317.2022.00108

LAK: A Low-Overhead Lock-and-Key Based Schema for GPU Memory Safety

Chaochao Zhang, Rui Hou*
State Key Laboratory of Information Security, Institute of Information Engineering, CAS
and School of Cyber Security, University of Chinese Academy of Sciences.
Beijing, China
Email:{zhangchaochao, hourui}@iie.ac.cn

***Abstract*—Graphic processing units (GPUs) are becoming an essential computational resource in a widely range of domains. At the same time, the security of GPUs has emerged as a primary concern in security-sensitive applications. Memory corruption attacks are the major threat in computer security. With contemporary discrete GPUs supporting unified address space, which allows GPU kernel functions to access the same address space with CPU host applications, memory bugs (e.g., buffer overflow and use-after-free) in GPUs can be exploited by attackers to maliciously tamping CPU data or even hijacking control flow. However, modern GPUs lack memory protection support against memory corruption attacks concurrently available in CPUs, and as a result suffer from security threats, as we demonstrate.**

In this paper, we migrate the conventional CPU memory protection mechanisms to GPUs and point out that directly adopting the CPU memory protection design to GPUs incurs significant performance overhead. The key reason is that: (1) fine-grained multithreading property of GPUs leads to high memory traffic for security metadata accessing, which leads to significant memory bandwidth contention between regular application data and security metadata and degrades the GPU performance; (2) shared L2 data cache suffers significant interference when regular data shared cache with metadata, which leads to increased cache miss rate and worsens the bandwidth contention problem.

Based on our observation, we propose LAK, a low-overhead runtime memory safety solution for GPUs to provide comprehensive protection against memory corruption attacks. First, to provide a strong security guarantee, LAK employs a lock-and-key mechanism to enforce memory operations to only access allowed memory regions. Second, to mitigate bandwidth contention, which degrades GPU performance, LAK introduces two components: (a) a post-coalescing memory protection unit to reduce memory traffic of security metadata and (b) a highly-threaded dedicated L2 metadata cache to reduce the interference between metadata requests and data requests. Our evaluations show that LAK incurs a 19% performance degradation.

***Index Terms*—Memory Safety, Unified Memory, GPU**

I. Introduction

With steady increased computing density and continued improved programmability, Graphic Processing Units (GPUs) are becoming an essential computational resource in a widely range of application domains, such as high performance computing, deep learning, and cloud computing, because of their ability to provide better performance and lower energy compared to CPUs. Given the growing widely use of GPU to accelerate security-sensitive applications (e.g., financial computation applications, deep learning computing and crypto-graphic accelerations), some recent works have been proposed to secure GPU computation. For example, Gravition [25] and HIX [12] integrate trusted execution environments (TEE) onto GPUs to provide secure isolated computing environments. Kadam et al. [13], [11] presented serial works to thwart timing-based side-channel attacks that exploit memory coalescing in GPUs. PSSM [27] proposed a memory encryption design for partitioned GPU memory structure to defend against physical attacks.

Memory corruption bugs (e.g., buffer overflow and use after free) are the most common problem in CPU-based applications. Low-level languages like C/C++ are the root cause of these kinds of bugs due to the lack of memory safety in such languages. In the same vein, GPU applications written in CUDA language are also vulnerable to potential attacks that exploited memory bugs. Because CUDA [2] is a type unsafety programming language, which extends from C/C++, naturally, it inherits the uncontrolled memory management features from C/C++. Some recent works [3], [7], [8], [16], [17] showed that attackers can alter the CUDA application's behavior by hijacking its control flow. In particular, an attack exploits GPU-based buffer overflow to hijack control flow by tampering code pointer (e.g., function pointer, virtual table of C++ objects) with the knowledge of its relative locations in program's address space. While classic return orient programming (ROP [23]) attacks seem not to be feasible as the return address is stored at an unknown location and the underlying implementation is undocumented by NVIDIA, it can potentially threaten the GPUs security with the improvement of reverse engineering tools.

Additionally, to simplify GPU programming model and eliminate manual memory management, contemporary GPUs support unified memory, which provides a single memory address space that allows allocated data to be read or written from code running on either CPUs or GPUs. With GPU codes being able to access the same address space of CPU applications, GPU-based memory vulnerabilities can be exploited to easily corrupt CPU data. This may becomes a critical concern in view of such tight integration.

There currently exist few solutions to protect GPU against memory-corruption attacks. Oclgrind [21] and Cudagrind [5], similar to Valgraind for CPU applications, instrument

2576-6996/22/$31.00 ©2022 IEEE
DOI 10.1109/ICCD56317.2022.00108

OpenCL/CUDA code to find memory bugs, but both incur a heavy performance. To mitigate performance overhead, Erb et al. [8] present a canary scheme, similar to stackGuard in CPUs, to detect buffer overflows. By placing a unique value (called canary) outside of the buffer, the integrity of the canary can be checked after the kernel function returned and thus the overflow can be detected.

However, we find that the existing techniques for buffer overflow detecting in modern GPUs have two major shortcomings: (1) Aforementioned solutions offer limited security guarantees. Firstly, canary solutions cannot detect non-adjacent illegal memory access that jumps over canaries with the knowledge of predictable memory layout by counting canary size and buffer size. Secondly, they cannot detect use-after-free violations, which can be exploited easily due to lack of address space layout randomized (ASLR) in GPU runtime memory management. (2) The performance overhead of the aforementioned solutions are too high to be runtime solutions. Instead, their major usage is for testing and debugging before real world deployment. For instance, Oclgrind incurs performance overhead up to 300x [21]. Hence, we argue that implementing a low-overhead runtime memory safety solution is needed for GPUs to provide comprehensive protection against memory corruption attacks.

To this end, a straightforward solution is adopting conventional CPU-based memory protection mechanisms for GPUs. However, since GPU architectures deviate significantly from multi-core CPUs, directly migrating CPU memory protection solutions to GPUs can degrade GPU performance dramatically. The main reason is that GPU is designed for highly multi-threaded computation which leads to massive memory requests for security metadata, such as memory tags, boundary info, etc. Consequently, the memory bandwidth contention between regular application data and security metadata, can potentially hurt GPU performance.

To perform a thorough analysis on performance impact of adopting CPU based memory protection techniques to GPUs directly, we first implement the state-of-the-art CPU based memory protection mechanism (i.e. Arm MTE [22] and Intel MPX [19]) on a GPU simulator (Section III). We make following key observations from our analysis. (1) Security metadata traffic is high due to large amounts of concurrently issued memory instructions, which is explored in contemporary GPUs to hide long memory latency. Thus, a large portion of memory bandwidth is consumed by metadata. As a result, more warps are stalled due to the additional queuing delay, leaving the warp scheduler often not have enough ready warps to issue, which can hurt the overall GPU performance. (2) Significant interference between regular application data and security metadata in L2 data cache, leads to increased cache miss rate, which further worsens the bandwidth contention problem. Alternatively, a dedicated partitioned L2 metadata cache can potentially eliminate such interference, but the design complexity and high are overhead of this operation will not suitable to GPUs due to on-die area limitation and energy efficiency.

Base on our observations, we propose **LAK**, a low-overhead runtime memory safety solutions for GPUs to provide protections against memory corruption attacks. Given the high performance impact from memory bandwidth contention due to additional memory requests of metadata, the overarching idea is to reduce metadata traffic, and to mitigate metadata interference at L2 shared data cache. First. We propose **L**ock-**A**nd-**K**ey mechanism (memory objects are tagged by a lock upon allocation, and pointer must have the correct key to be dereferenced) to provide strong security guarantee by enforcing memory operations to only access allowed memory regions. Comparing to pointer boundary validation mechanism, lock-and-key schema can reduce metadata overhead with adjustable tag granularity (e.g., 16 bytes per 4 bits in Arm MTE [22]). Second, our post-coalescing memory protection unit performs lock-and-key checks on coalesced memory accesses to reduce memory traffic. Third, our highly-threaded metadata fetcher provides a non-blocking mechanism to address the need of concurrently metadata requests, also including a dedicated cache to reduce the consumption of off-chip memory bandwidth and long memory access latency of metadata.

This paper makes the following contributions:

- We show that naive adoption of CPU-based memory protection mechanisms in GPUs, results significant performance overhead. Further, we point out that memory bandwidth contention and cache interference between regular data and metadata are the key performance bottlenecks.
- To mitigate memory bandwidth contention, we introduce a post-coalescing memory protection unit to reduce memory traffic of metadata, and a highly-threaded metadata cache to eliminate cache interference.
- To our best knowledge, this is the first work to provide a runtime memory protection for GPUs against memory corruption attacks. And our evaluations show that our design only incurs 14% performance overhead with memory protection support.

The rest of paper is organized as follows. Section II describes the background of memory corruption attacks in GPUs and specifies the thread model. Section III presents naive design. Section IV discusses our observations and motivations of our design. Section V proposes our design. Sections VI evaluates the performance and memory overhead. Section VII presents the related works, and Section VIII concludes our work.

II. Background

A. Threat Model and Scope of Our Work

We assume that GPU's memory is also vulnerable to the conventional memory corruption attacks considered in CPUs. We assume the CUDA program running on GPU to have one or more memory vulnerabilities that an attacker can exploit to read and write arbitrary memory locations. Furthermore, we assume that the adversary can access to the source code

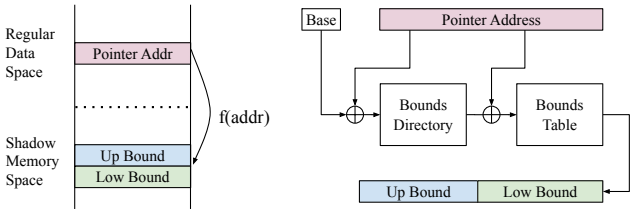
```
1  typedef void (*Dummy)(char* , int );
2  __device__ void Dummy(char* input, int length){
3    /*implementation*/
4  }
5  __global__ void kernel(char* input, int length){
6    int idx=blockDim.x*blockIdx.x+threadIdx.x;
7    Dummy fptr;
8    char buf[BUF_LEN];
9
10   /*Attacker overwrites function pointer
11   via controlled input and length*/
12   memcpy(buf, input, length);
13
14   //control flow hijacked
15   fptr(input+(length*idx), length);
16 }
```

(a) stacked based buffer overflow

```
1  class Dummy(){
2    public:
3    void __device__ virtual func(char* input, int length);
4  }
5  __global__ void kernel(char* input, int length){
6    int idx=blockDim.x*blockIdx.x+threadIdx.x;
7    char* buf = (char*)cumalloc(sizeof(char)*BUF_LEN);
8    Dummy *obj = new Dummy;
9
10   /*Attacker overwrites the class V-table,
11   which is stored in first 64-bit of obj*/
12   memcpy(buf, input, length);
13
14   //control flow hijacked
15   obj->func(input+(length*idx), length);
16 }
```

(b) heap based buffer overflow

Fig. 1. Memory Corruption Attack on GPUs



(a) Direct Mapping Schema (b) Two-level Table Schema

Fig. 2. Comparison of metadata Management

of the CUDA program, thus he is able to glean all source-level information, e.g., the relative distance between memory vulnerability and target variable. Finally, we assume the processes chip forms the trust computing boundary(TCB), and all hardware components are assumed to be out of the reach for attackers.

The scope of this paper covers the protection of memory corruption attacks on GPUs. Defending against GPU side channel attacks, such as timing-based side channel attacks are out of the scope of this work. Also, GPU memory encryption as well as PCIe bus protection scheme, have been studied in previous work [27], [26], and therefore we leave out of scope any attacks that exploit memory bus snooping or physical access to the off-chip GDDR memory.

B. Memory Corruption Attack on GPUs

Fig. 1 shows an example of stack exploitation in CUDA code. The function kernel() is vulnerable to a buffer overflow. Line 15 in the listed kernel function contains the buffer overflow bug wherein the validity of data length is not checked before the input data to be copied into the destination buffer. The resulting exploit can then be used to tamper adjacent memory objects (i.e., function pointer, *fptr*, in line 8). Thus, an attacker can invoke any device function in the code memory by overwriting the function pointer in *fptr*. Furthermore, it is possible to use cuda-gdb or cuobjdump to figure out what are the addresses of the victim functions the attackers willing to invoke.

Theoretically, the conventional return oriented programming (ROP) is possible. However, practically speaking, classic ROP attacks seem not to be feasible as the return address is stored at an unknown location and the underlying implementation is undocumented by NVIDIA. Furthermore, any attempts to jump out of code segment are failed, and we infer that some security mechanisms (i.e., non-executable data) are deployed in GPUs. Thus, simple code injection attacks are not possible.

Consider the following code where we define a C++ class Dummy with a virtual method: func (the qualifier device denotes a function that is called by threads running on the GPU). The global function kernel dynamically allocated an array buffer and a class Dummy object in heap. Similarly, the kernel function is also vulnerable to a heap overflow as the array buffer can be overridden to overwrite the adjacent class Dummy object.

Similar to conventional dynamic memory management in C/C++ runtime, CUDA runtime dynamically allocates memory blocks (malloc) or objects (new) with a predictable location in heap. In this case, the allocated string buffer and the class instance are always in a fixed memory location. Moreover, the relative distance between both is constant, resulting the array buffer can be overridden to overwrite the adjacent class Dummy object. In addition, the address of virtual table is stored in the first 64-bits value of class instance. Thus, attackers can exploit the overflow vulnerability to overwrite the address of class object's virtual table with the address of a "fake" virtual table, which can be stored in the attacker-controlled string buffer. Then the GPU's control flow is hijacked as the device threads are forced to call unintended functions.

III. Baseline Design

To solve the problem of memory error based attacks in GPUs, it is natural to migrate well-studied CPU based defense techniques into GPUs directly. To analyze the performance impact of such design, we first implement the state-of-the-art CPU based memory protection mechanisms on a GPU architecture simulated using a cycle accurate GPU simulator-GPGPU-Sim4.0. Then we evaluate the performance overhead of those designs with sufficient workloads, and address the performance bottlenecks by performing a detailed analysis. The simulation methodology is presented in Section VI-A.

State-of-the-art CPUs provide memory protection by checking the validity of each memory access with the help of secure metadata. For instance, Intel Memory Protection Extensions (Intel MPX) performs dynamic bounds-checking on each pointer dereferencing, Arm Memory Tagging Extension (Arm MTE) implements a lock-and-key scheme to secure memory access. Both memory protection techniques require the associated metadata to be addressed and fetched from memory on each memory access, causing significant performance degradation for those memory-intensive applications. Furthermore, the secure metadata storage often incurs significant memory overhead. The design choice is a trade off between performance and memory overhead. To mitigate memory overhead, Intel MPX stores bounds metadata in memory with a two-level

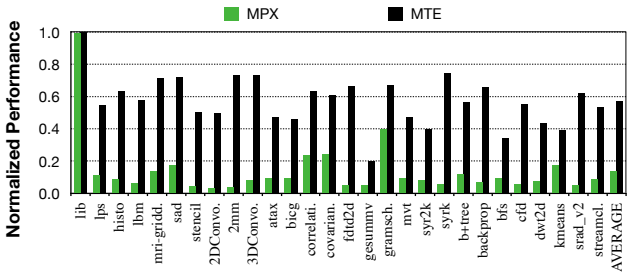


Fig. 3. Performance comparison of naive designs

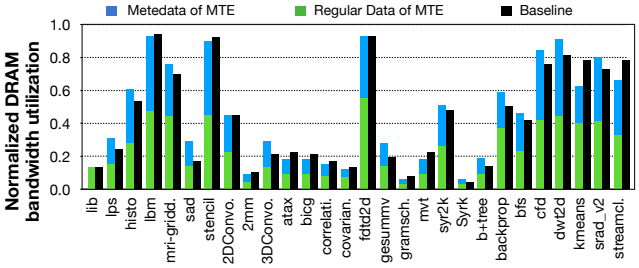


Fig. 4. Breakdown of bandwidth utilization on metadata and regular data.

bounds table. However, the address of metadata is calculated with a two-level address translation scheme, similar to virtual address translation, as shown in Fig. 2b. This multi-state translation requires two memory loads additionally, resulting in high performance overhead (50% on average). Conversely, Arm MET stores metadata in a shadow memory location calculated with directing mapping address translation scheme. Thus only one additional memory access for metadata fetching. However, shadow memory implementations mirror regular data space to the shadow space, as shown in Fig. 2b, which results memory fragmentation and memory overhead.

We model the GPU memory protection schema based on above design, whose metadata management makes use of one of two variants for memory access validation: (1) direct mapping schema, a previously proposed design that mirrors a pointer to the associated metadata with a constant offset (Fig. 2a); (2) two-level table schema, a design that stores the metadata in a multi-level bound table (we assume a two-level bound table in this paper, Fig. 2b). In both variants, the secure metadata shares a private per-core L1 cache and shared L2 cache. For direct mapping schema, we extend PTX instruction set with LDG and STG instructions. LDG/STG instructions allow getting or setting tags in memory with one memory request. We modified the load and store PTX instructions to raise an exception on a mismatch between the pointers and memory tags. For the two-level table schema, we extend PTX instructions set with bndldx, bndstx and bndcx instructions. bndldx/bndstx instructions are used to load/store pointer bounds in two-level bounds tables respectively, and bndcx instructions are used to compare the pointer value against the lower and upper bounds. Note that two sequential memory requests are required to address bounds metadata.

Fig. 3 compares the performance of both naive designs (MTE, depicted in Fig .2a, and MPX, depicted in Fig .2b) to baseline GPU without memory protection support (see Table I for our GPGPU-Sim configuration, and Section VI-A for our methodology). In the figure, the design directly adopted from Arm MTE is labeled as MTE, and the design directly adopted from Intel MPX is labeled as MPX. We find that both designs incur a significant performance overhead (43% and 79% on average) compared to the baseline GPU without memory protection support. The performance degradation is even higher for memory-intensive benchmarks like kmeans (63%) and gesummv (80%). Moreover, with more memory requests for metadata addressing and fetching, the average performance degradation of MPX is more than tripled to MTE. It indicates that the memory requests for load/store metadata from/to GDDR memory is one of the major performance bottleneck. Furthermore, the memory bandwidth contention between regular data and security metadata is another reason, that hurts the performance of GPUs. In the next section, we perform a thorough experimental analysis to address the shortcomings of our naive designs.

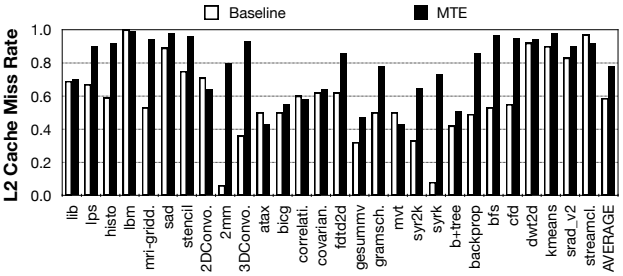


Fig. 5. Interference on the shared L2 data cache miss rate

IV. Performance Degradation Diagnosis

A. Memory bandwidth contention

As presented in section III, the frequency memory requests for fetching/storing security metadata from/to off-chip memory, imply the memory contention between security metadata and regular application data, which degrades GPU performance. Fig. 4 displays the breakdown of off-chip memory bandwidth used by security metadata (left bar, top, blue) as well as bandwidth used by regular application data (left bar, bottom, green), both normalized to the maximum bandwidth available. We can see that, with memory bandwidth contention due to security metadata requests, the bandwidth consumption of regular application data degrades almost half (average 23% of utilization) of total utilized memory bandwidth (average 46% utilization), compare to the baseline bandwidth utilization with no memory protection adopted (average 41%). To better understand the performance impact on bandwidth contention, we measure the average GPU stall cycles caused by security metadata requests and regular application data requests. The results show that the increased pipe stalled cycles, caused by

metadata memory requests, are almost the same as regular application data.

B. Interference in L2 data cache

When security metadata shares L2 cache with regular application data, the memory bandwidth contention discussed in Section III is aggravated due to the high cache miss rate. To better understand the performance impact of this interference, we present how the L2 cache miss rate changes when security metadata is housed in L2 cache. Fig. 5 compares the L2 cache miss rate of baseline GPUs without memory protection support to the miss rate of our naive MPX design. We choose MPX to measure, as each pointer dereferencing in MPX design requires three memory requests, two for metadata addressing and one for metadata fetching, which leads the interference to be exacerbated.

From the Fig. 5, we can make the following observation that the shared security metadata increases the L2 cache miss rate significantly (from 59% to 78% on average), and potentially results to high memory bandwidth consumption. For those low memory intensive benchmarks, like 2mm (10%) and syrk (4%), the shared L2 cache miss rate increases significantly. For instance, 2mm increases from 6% to 80%, and syrk from 8% to 73%. The reason is that each pointer dereferencing needs corresponding metadata, which displaces cache entries housing regular application data. For those memory intensive benchmarks, like lbm (94%), fdtd2d (93%) and streamcluste (78%), it typically have high L2 cache miss rate (e.g., lbm 99%, fdtd2d 92%, and streamcluste 97%). Consequently, the high L2 cache miss rate coupled with high off-chip memory traffic, leads to bandwidth contention, which can hurt the overall GPU performance.

V. DESIGN OF LAK

A. Lock-and-Key Protection Schema

The virtual memory supported in modern GPUs allows us to share a GPU among concurrently-executing applications. To guarantee the applications running in different virtual addresses are isolated from each other, the GPU runtime driver ensures that each application uses its own page table. However, modern GPUs lack of intra-process memory protection for memory corruption defenses to protect sensitive data.

Unlike previous-proposed GPU memory safety techniques that do not provide protection against memory corruption attacks, LAK provides memory protection by enforcing valid memory access with the help of lock-and-key schema. In this schema, memory objects are tagged by additional bits of metadata (also called lock) upon allocation, and each valid pointer to that memory object is modified to contain the matching key. On every pointer dereferencing, the access is permitted if the key matches the lock, otherwise, an exception is raised on mismatch, as shown in Fig. 7. In order to avoid larger pointer requirement, the bits in the unused upper part of a virtual address are used as the key. The size of a key should be large enough to have a sufficient number of different key values (i.e., ARM MTE provides 4 bits to have 16 key values), but is limited by the number of unused upper bits in a virtual address. Meanwhile, the size of each tagged memory chunk, called tag granule, is a balance between the tag storage overhead and the alignment overhead (ARM MTE tagged memory chunks by adding 4 bits of lock to each 16 bytes of physical memory).

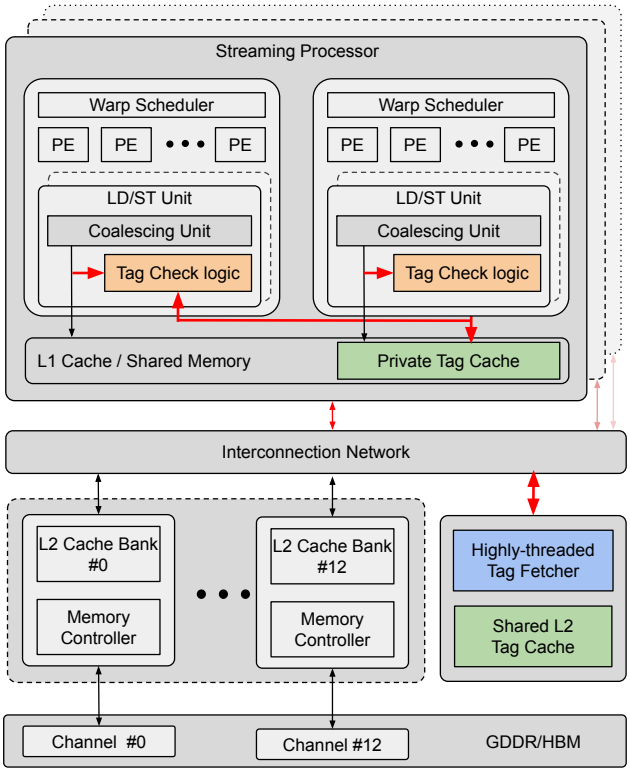


Fig. 6. Overview architecture.

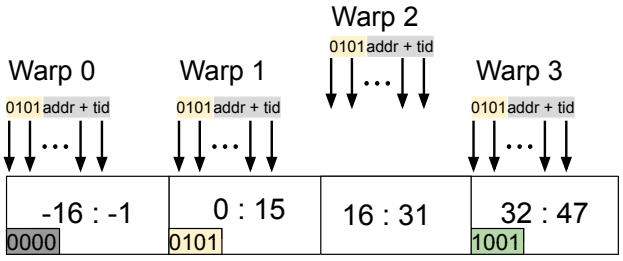


Fig. 7. shows an example of lock and key access to memory.

Typically, GPUs do not zero to deallocated memory pages, as this is considered to be not worth the overhead. Thus attackers could reveal a victim program's data by exploring such vulnerabilities. Detecting use of uninitialized memory if often not trivial, however, if the memory is already tagged, it comes almost free for such detection. As Fig. 7 shows, once a memory chunk is freed, the corresponding lock is set to be '0000' instead of initializing the memory chunks to zero. Any pointer dereferencing with mismatched key is not allowed

(warp 0). Also, buffer overflow can be detected, because the pointer's address key ('0101') does not match the memory lock ('1001') which is belonged to other pointer, as the warp 3 showed in Fig. 7.

B. Post-coalescer Memory Protection Unit

Inter-warp coalescer is commonly used for commercial GPUs to reduce memory bandwidth consumption by agglomerating the memory requests of active threads within each warp, whose requesting data size can range from 32 to 128-byte. Section IV-A demonstrates the need to mitigate memory bandwidth contention issues due to additional security metadata requests, which induce long-latency stalls causing GPU performance to degrade significantly. LAK addresses this need with a post coalescer tag check logic (yellow in Fig. 6), called post-coalescer MPU. To reduce memory bandwidth consumption of metadata requests, LAK moves memory protection unit from before to after inter-warp coalescer.

For various benchmarks, Fig. 8 presents memory operations per billion instructions for pre-coalescer memory instructions (left, black) and post-coalescer memory instructions (right, red), both normalized to their pre-coalecer memory instructions for regular application data. The 'AVERAGE' bars represent the average reduction of those benchmarks on arithmetic mean. As Fig. 8 shows that, with help of memory coalescing unit, the number of memory operations for security metadata is reduced for an 87% reduction. The performance gap between the practical coalesce on memory access (87% on average) and the one with prefect coalesce (97% theoretically 32 operates to one), is caused by the memory divergence in benchmarks.

C. Highly-threaded Metadata Fetcher

The high security metadata traffic is because GPU's basic fine-grained multithreading design, which is essential for tolerating long latency memory accesses. To facilitate the execution of massive scalar threads in the single-instruction multi-data (SIMD) pipeline, threads are scheduled at the granularity of a warp, which consists of 32 threads executing a single instruction in a lockstep manner (i.e., each thread in the warp executes the same instruction concurrently). Because of this characteristics, threads within the same warp can simultaneously issue a memory instruction with different addresses. This coupled with the fact, that GPUs contain several streaming multiprocessors (SMs), and each SM supports concurrent warps executing, means that the security metadata traffic will be high.

To address the need of high traffic of security metadata, a naive design choice is to share the existing GPU's partitioned memory architecture (i.e., sectored caches, banked register files) with security metadata to improve metadata throughput by exploiting parallelism memory accesses and to reduce bandwidth consumption by enabling GPU threads to read/write small amounts of metadata on demand. However, as presented in Section IV-B, a shared L2 data cache housing security metadata and regular application data housing both security metadata and regular application data inside a same data cache,

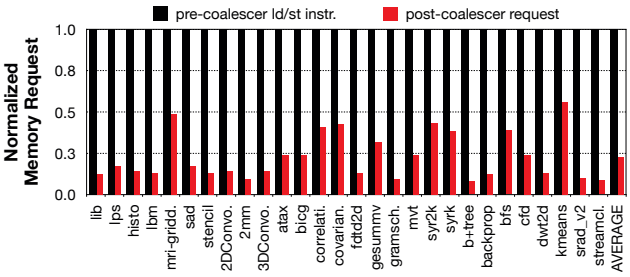


Fig. 8. Pre-coalescer memory requests vs. Post-coalescer memory requests.

TABLE I
GPU CORE CONFIGURATION

Parameter	Configurations
Shader Core	30 Cores, 1365*MHz*, GTO scheduler
L1 Cache	64 KB/SM, 4 bank, 2 latency
L2 Cache	3MB total , 16-way associative, 128 KB each bank and 2 banks per memory partition, LRU
DRAM	3GB GDDR5 timing, 3500 *HMz*, 100-cycle latency, burst length 8, 12 partitions, FR-FCFS scheduler

that can hurt the GPU performances. The main reason is that metadata can interfere with and displace cache entries housing regular application data, resulting a significant miss rate on regular application data. Alternatively, a dedicated partitioned L2 cache used for metadata keeping, could potentially eliminate interference at L2 cache between metadata requests and regular data requests. Unfortunately, the complexity and high area overhead of this separation will be not suitable to a GPU execution environment.

Categorization	Benchmark Name	Bandwidth utilization	Benchmark Name	Bandwidth utilization
Non memory intensive	LIB	13%	sad	17%
	2mm	10%	correlation	17%
	covariance	13%	gesummv	19%
	gramschmidt	8%	syrk	4%
	b+treei	14%		
Medium memory intensive	LPS	24%	2DConvolution	45%
	3DConvolution	21%	bicg	21%
	atax	22%	mvt	22%
	syr2k	48%	backprop	50%
	bfs	42%		
Memory intensive	histo	53%	stencil	92%
	lbm	94%	fdtd2d	93%
	cfd	76%	dwt2d	81%
	kmeans	78%	srad_v2	73%
	streamcluste	78%	mri-gridding	70%

Fig. 9. Benchmarks are categorized by bandwidth utilization .

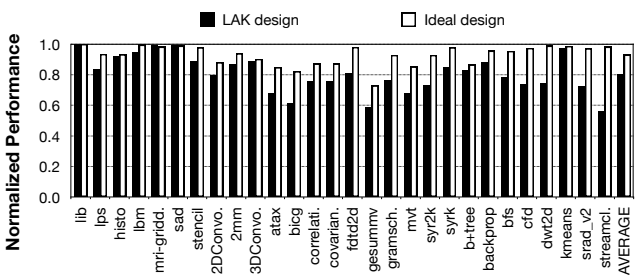


Fig. 10. Performance comparison of ideal design and our design .

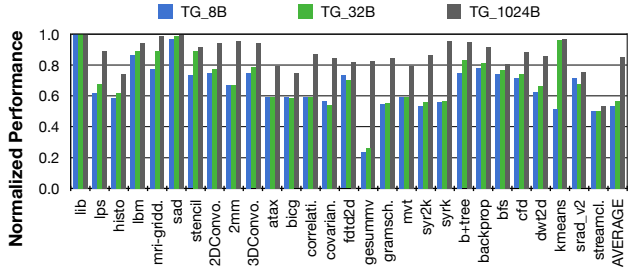


Fig. 11. Performance comparison with different tag granularity.

VI. EVALUATION

A. Methodology

We model our proposed schemes with GPGPU-Sim v4.0. Table I elaborates the details of our baseline GPU configuration, which is based on the NVIDIA Turing architecture.

Workload. Our benchmarks are from the Ispass [4], Rodinia-3.1 [6], Parboil [24] and Polybench [10] suites. We select 26 memory intensive workloads, since computation intensive workloads are relatively insensitive to metadata load/store overheads. We classify these benchmarks based on their bandwidth utilization when running on the baseline GPU without secure memory support, as shown in Fig. 9. For benchmarks with low simulation time, we simulate the entire benchmarks; for benchmarks with long simulation time, we simulate the first one billion instructions as as the input sizes are too large to simulate.

B. Overall Performance

We evaluate our memory safety design and compare it with an ideal design, which is modeled with an infinite sized metadata cache and minimal latency (1 cycle) for metadata addressing and fetching. Fig. 10 shows the performance results (normalized to the baseline GPUs without memory protection support). We make two observations from Fig. 10. First, although with infinite sized metadata cache and minimal latency on metadata fetching, the ideal design (impossible to implement) still degrades the GPU performance by 6% on average. We identify that the performance overhead is mainly due to the extra instructions for memory checking and metadata addressing and fetching. Second, compare to naive design (described in section III labeled as 'Naive'), our design

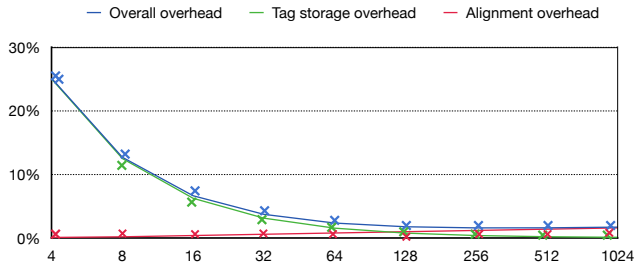


Fig. 12. Memory overhead comparison of different metadata granularity.

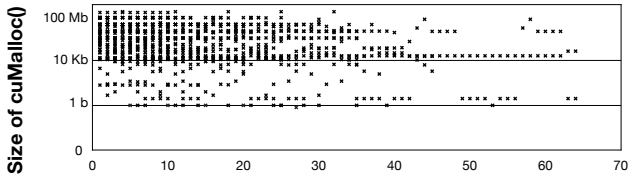


Fig. 13. Memory object distributions of cuMalloc function.

improves the performance significantly. The average performance overhead is reduced from 53% to 19%. The main reason is that the post-coalescing memory protection unit significantly reduces the memory bandwidth for accessing metadata, and shared metadata cache in highly-threaded metadata fetcher reduces L2 cache interference. However, our design shows a performance gap compared to the ideal design (about 9% on average). Because, a portion of metadata transfers over the off-chip memory, which results the SIMD pipeline to be stalled due to long memory access latency.

Performance Impact on Tag Granularity To better understand the performance impacts of tag granularity, we measure the performance overhead with different tag granularity and show results (normalized to ideal design) in Fig. 11. From Fig. 11, we can make the following observations. First, the coarse-grained tag is beneficial. For example, compared with 32-bytes tag granularity (a tag is used to mark an aligned 8 bytes memory chunk, as labeled as 'TG-8B' in Fig. 11), the performance overhead is reduced from 54% to 85% for 1024-bytes tag granularity and 62% for 32-bytes tag granularity. The reason is that an coarse-grained tag can reduce metadata miss rate in both L1 shared data cache and delicate L2 metadata, thus the memory bandwidth for metadata is saved.

C. Memory Overhead

The major source of memory overhead are come from the memory tag storage (extra N-bits per every tagged memory chunk) and over-aligning memory objects from the alignment to tag granularity. We present the memory overhead with different tag granularity in Fig. 12. From the Fig. 12, we can make the following observations. First, the extra tag storage mainly contributes high memory overhead, especially for fine-grained tag such as the granularity of 4-bytes (25%) and 8-bytes (13%). second, the alignment overhead impacts negligibly on memory overhead. The main reason is that the GPU

applications perform most of their memory allocation en mass. Thus, this en mass allocation allows CUDA runtime to perform efficient alignment with continuous objects in memory. This characteristic further illustrates in Fig. 13. As Fig. 13 shows that the size of allocated objects in GPU memory are well distributed.

VII. RELATED WORK

A. Memory bug detection tools on GPUs

Several memory bug detection tools have been proposed in the context of GPUs. Oclgrind [21] is an analysis tool used to search memory error (e.g., buffer overflow) for OpenCL kernels, like Valgrind, which is a popular memory analysis tools designed for CPUs. Similar to Valgrind, Cudagrind [5] can be used to detect memory error in CUDA code. Both. Of Oclgrind and Cudagrind suffer from significant runtime slowdown (roughly 3X-300X) due to GPU code instrumentation on CPUs. Alternatively, CUDA-MEMCHECK [9] is another memory access analysis tool released by NVIDIA with less overhead.

Recently, Erb et al. [8] proposed a canary scheme build on OpenCL warpper to search buffer overflow vulnerabilities. As the integrity of canary is checked only after kernel function returned, while the runtime overhead is significant reduced, the buffer overflows occurred within kernel running window cannot be detected. Moreover, the aforementioned solutions are mainly used for GPU codes testing and debugging, not in deployment builds. Comparatively, our design provides low-overhead runtime protections for GPUs with even stronger security guarantee.

B. GPU memory security

Recent works point out that GPUs are vulnerable to a variety of memory attacks. For memory information leakage, CUDA leak [20], Sangho et al. [15] and zhu et al. [28] demonstrated information leakage through residual data in deallocated memory objects in GPUs. For memory corruption attacks, Miele [16] presented a preliminary study of buffer overflow vulnerabilities in CUDA codes and demonstrated memory corruption attacks on GPUs. On the other hand, to ensure memory confidentiality and integrity on GPUs, Na et al. [18] proposed common counter to encrypt/decrypt GPU memory. To deduce performance overhead, PSSM [27] proposed by Yuan et al. generates encryption metadata using partition-local address instead of virtual or physical address, as a result, the metadata redundant problem is addressed. Later on, Yuan et al. [26] further improve the performance by eliminating the freshness checks on read-only data, and propose coarse-grain message authentication code to reduce metadata memory bandwidth. However, the aforementioned memory encryption solutions cannot prevent memory corruption attacks, instead they are designed for physical attack. These approaches can be combined with our LAK design to provide stronger protections on GPUs.

A concurrent work to our paper is GPUShield [14] where the authors propose a bounds-checking based mechanism to improve GPU memory safety. GPUShield assigns a unique ID to security-sensitive regions (e.g., buffers, local variables) and employs a bounds table to store corresponding bounds information. Typically, bounds-checking based mechanisms incur significant performance overhead due to bounds propagation and addressing. To minimize performance impact, GPUShield extends GPU compiler to perform static bounds checking to filter out unnecessary runtime bounds checking, and performs runtime bounds checking in warp-level. Compared to GPUShield, LAK utilizes memory-tagging mechanism, which eliminates additional memory access for bounds management. Despite, the limited size of tags reducing security guarantee, the tagging information propagates along with the pointer address, eliminating propagation overhead. Furthermore, the design of LAK's tag is compatible with conventional memory tag (i.e., similar to Arm memory tag mechanism in this paper), thus the tag propagated with address pointer could be shared with CPU side memory-tag based protections, to ensure both CPU side and GPU side secure in Shared Virtual Memory in OpenCL [1] and Unified Memory in Nvidia CUDA [2].

VIII. CONCLUSIONS

In this paper, we propose a low-overhead memory safety solutions for GPUs against memory corruption attacks. We perform a thorough analysis of performance bottlenecks when directly adopting CPU-based memory safety solutions to GPUs, and identify that security metadata leads memory contention and cac he interference. Based on our observations, we employ lock-and-key schema to provide strong security guarantee, integrate a post-coalescing memory protection unit to reduce memory contention, and introduce a delicate metadata cache to eliminate cache interference. Our results shows that LAK achieves minima performance overhead while providing memory safety support for GPUs.

IX. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments. This work is supported by the National Natural Science Foundation of China (Grant No. 62125208). Rui Hou is the corresponding author.

REFERENCES

[1] "Intel. 2014. opencl 2.0 shared virtual memory overview." in *https://software.intel.com/content/www/us/en/develop/articles/opencl-20-shared- virtual-memory-overview.html.*

[2] "Nvidia corporation 2013. unified memory address space in cuda 6," in *https://developer.nvidia.com/blog/unified-memory-in-cuda-6/.*

[3] "Mind control attack: Undermining deep learning with gpu memory exploitation," vol. 102, 2021, p. 102115. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404820303886

[4] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 163–174.

[5] T. M. Baumann and J. Gracia, "Cudagrind: Memory-usage checking for cuda," in *Parallel Tools Workshop*, 2013.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[7] B. Di, J. Sun, and H. Chen, "A study of overflow vulnerabilities on gpus," in *Network and Parallel Computing*, G. R. Gao, D. Qian, X. Gao, B. Chapman, and W. Chen, Eds. Cham: Springer International Publishing, 2016, pp. 103–115.
[8] C. Erb, M. Collins, and J. L. Greathouse, "Dynamic buffer overflow detection for gpgpus," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017, pp. 61–73.
[9] G. Gerfin and V. Venkataraman, "Debugging experience with cuda-gdb and cuda-memcheck," in *Presented at GPU Technology Conference (GTC)*, 2012.
[10] S. Grauer-Gray and J. Cavazos, "Optimizing and auto-tuning belief propagation on the gpu." 01 2010, pp. 121–135.
[11] K. Gurunath, Z. Danfeng, and J. Adwait, "Bcoal: Bucketing-based memory coalescing for efficient and secure gpus," in *2020 IEEE International Symposium on High Performance Computer Architecture, HPCA*, 2020, pp. 570–581.
[12] I. Jang, A. Tang, T. Kim, S. Sethumadhavan, and J. Huh, "Heterogeneous isolated execution for commodity gpus," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 455–468. [Online]. Available: https://doi.org/10.1145/3297858.3304021
[13] G. Kadam, D. Zhang, and A. Jog, "Rcoal: Mitigating gpu timing attack via subwarp-based randomized coalescing techniques," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 156–167.
[14] J. Lee, Y. Kim, J. Cao, E. Kim, J. Lee, and H. Kim, "Securing gpu via region-based bounds checking," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 27–41. [Online]. Available: https://doi.org/10.1145/3470496.3527420
[15] S. Lee, Y. Kim, J. Kim, and J. Kim, "Stealing webpages rendered on your browser by exploiting gpu vulnerabilities," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 19–33.
[16] A. Miele, "Buffer overflow vulnerabilities in cuda: a preliminary analysis," in *Journal of Computer Virology and Hacking Techniques*, 2016, pp. 113–120.
[17] S. Mittal, A. S B, M. Yedulla, and I. Ali, "A survey of techniques for improving security of gpus," vol. 2, 09 2018.
[18] S. Na, S. Lee, Y. Kim, J. Park, and J. Huh, "Common counters: Compressed encryption counters for secure gpu memory," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 1–13.
[19] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel mpx explained: An empirical study of intel mpx and software-based bounds checking approaches," in *CoRR abs/1702.00719 (2017).http://arxiv.org/abs/1702.00719*, 2017.
[20] R. D. Pietro, F. Lombardi, and A. Villani, "Cuda leaks: A detailed hack for cuda and a (partial) fix," *ACM Trans. Embed. Comput. Syst.*, vol. 15, no. 1, jan 2016. [Online]. Available: https://doi.org/10.1145/2801153
[21] J. Price and S. McIntosh-Smith, "Oclgrind: An extensible opencl device simulator," in *Proceedings of the 3rd International Workshop on OpenCL*, ser. IWOCL '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2791321.2791333
[22] K. Serebryany, "Arm memory tagging extension and how it improves c/c++ memory safety," *login Usenix Mag.*, vol. 44, 2019.
[23] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07, New York, NY, USA, 2007, p. 552–561.
[24] J. Stratton, C. Rodrigues, I. Sung, N. Obeid, L. Chang, N. Anssari, G. Liu, and W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," 2012.
[25] S. Volos, K. Vaswani, and R. Bruno, "Graviton: Trusted execution environments on gpus," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 681–696.
[26] S. Yuan, A. Awad, A. W. B. Yudha, Y. Solihin, and H. Zhou, "Adaptive security support for heterogeneous memory on gpus," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 213–228.
[27] S. Yuan, Y. Solihin, and H. Zhou, "Pssm: Achieving secure memory for gpus with partitioned and sectored security metadata," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 139–151. [Online]. Available: https://doi.org/10.1145/3447818.3460374
[28] Z. Zhu, S. Kim, Y. Rozhanski, Y. Hu, E. Witchel, and M. Silberstein, "Understanding the security of discrete gpus," in *Proceedings of the General Purpose GPUs*, ser. GPGPU-10. Association for Computing Machinery, 2017, p. 1–11.