Securing GPU via Region-based Bounds Checking

Jaewon Lee
jaewon.lee@gatech.edu
Georgia Institute of Technology
USA

Yonghae Kim
yonghae@gatech.edu
Georgia Institute of Technology
USA

Jiashen Cao
jiashenc@gatech.edu
Georgia Institute of Technology
USA

Euna Kim
euna.kim@gatech.edu
Georgia Institute of Technology
USA

Jaekyu Lee
jaekyu.lee@arm.com
Arm Research
USA

Hyesoon Kim
hyesoon@cc.gatech.edu
Georgia Institute of Technology
USA

ABSTRACT

Graphics processing units (GPUs) have become essential general-purpose computing platforms to accelerate a wide range of workloads, such as deep learning, scientific, and high-performance computing (HPC) applications. However, recent memory corruption attacks, such as buffer overflow, exposed security vulnerabilities in GPUs. We demonstrate that out-of-bounds writes are reproducible on an Nvidia GPU, which can enable other security attacks.

We propose GPUShield, a hardware-software cooperative region-based bounds-checking mechanism, to improve GPU memory safety for global, local, and heap memory buffers. To achieve effective protection, we update the GPU driver to assign a random but unique ID to each buffer and local variable and store individual bounds information in the bounds table allocated in the global memory. The proposed hardware performs efficient bounds checking by indexing the bounds table with unique IDs. We further reduce the bounds-checking overhead by utilizing compile-time bounds analysis, workgroup/warp-level bounds checking, and GPU-specific address mode. Our performance evaluations show that GPUShield incurs little performance degradation across 88 CUDA benchmarks on the Nvidia GPU architecture and 17 OpenCL benchmarks on the Intel GPU architecture with a marginal hardware overhead.

CCS CONCEPTS

• **Security and privacy → Vulnerability management**.

KEYWORDS

GPU, memory safety

ACM Reference Format:
Jaewon Lee, Yonghae Kim, Jiashen Cao, Euna Kim, Jaekyu Lee, and Hyesoon Kim. 2022. Securing GPU via Region-based Bounds Checking. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3470496.3527420

1 INTRODUCTION

Graphics processing units (GPUs) were originally invented to speed up graphics rendering, but the general-purpose adoption of GPUs led to accelerating various application domains—computer vision, computational finance, bio-molecular analysis [40], weather prediction [44], and crypto-currency mining—thanks to their enormous high-throughput computing capability. Moreover, GPUs have recently ignited the use of deep learning and artificial intelligence (AI) models, and virtualized GPUs are used to accelerate these workloads in cloud platforms. Shared Virtual Memory (SVM) in OpenCL 2.0 [23] and Unified Memory (UM) in Nvidia CUDA 6 [50] were introduced to improve data sharing between CPUs and GPUs in the same system. Also, Nvidia recently announced Grace CPU [58] to enable tighter integration between CPUs and GPUs via NVLink [51] in data-centers to boost complex AI and HPC workloads. When GPUs were used mainly for graphics in the past, they were considered passive, i.e., not too powerful to break system integrity. However, with the broad adoption of GPUs and tighter integration with CPUs, more GPU applications started to process sensitive and private data, which led to increased security concerns on GPUs.

Re-steering the control flow by overwriting a function pointer or return address, such as return-oriented program (ROP) [65] and jump-oriented program (JOP) [8], is a well-known exploit to compromise the system. By running a malicious gadget code on an altered control flow path, attackers can collect sensitive data or escalate the attacker's privilege to the system administrator. Recent work has shown that GPUs are also vulnerable to buffer overflow attacks [12, 14, 47]. For example, research by Miele [45] describes GPU attack scenarios in which the attacker manipulates the function pointer by using a buffer overflow and successfully forces the GPU kernel to execute the malicious function. The mind control attack [61] leverages a buffer overflow attack to reduce the prediction accuracy of machine learning workloads.

Researchers have proposed software-based GPU memory safety mechanisms [13, 14] that use a compiler-generated canary [9] by intercepting memory allocation functions to add secret bytes before and after each buffer. After a kernel completes its execution, canary bytes are checked to see if any write occurred. This approach incurs little hardware overhead but cannot guarantee strong memory safety because it cannot detect 1) illegal reads and 2) non-adjacent out-of-bounds (OOB) reads and writes that jump over the canary region. Buffer overflow detection tools [15, 31, 54, 63] also exist, but their performance overhead is shown to be too high to be used at runtime.

Various hardware-based memory safety mechanisms exist for CPUs [11, 21, 22, 35, 38, 48, 59, 78], but no such mechanism exists for GPUs to the best of our knowledge. Any hardware mechanism should not increase memory traffic. Otherwise, it will incur severe performance degradation due to the massively parallel execution capability of GPUs. Thus, our primary goal is to implement an efficient memory safety mechanism by utilizing the GPU's unique programming and execution models. Originated from traditional graphics programming models, GPU programming models adopt disciplined memory regions—global, local, heap, texture, and constant memory—and several addressing modes. To better use massively parallel hardware, the GPU programming model enforces the limited usage of memory buffers. All globally visible memory buffers used by a GPU kernel must be specified in the kernel argument, and dynamic memory allocations are rarely used. Consequently, a GPU kernel maintains only a few memory buffers during its execution.

In this paper, we propose GPUShield, which implements a region-based bounds-checking mechanism to improve GPU memory safety. We maintain the bounds information of each buffer in a newly introduced per-kernel Region Bounds Table (RBT). We modify the GPU driver such that it assigns a random but unique ID to each buffer and local variable and sets up RBT upon a GPU kernel launch. Also, the driver embeds the *encrypted* buffer ID in the unused upper bits of the buffer base address pointers. This pointer-tagging approach [21, 22, 35, 38, 78] is lightweight and efficient since it removes the need for bounds propagation and does not require any hardware changes.

We extend the memory system similar to a CPU-based mechanism, AOS [35], to perform hardware-based bounds checking. However, we cannot adopt the CPU mechanism directly because of its performance overhead. GPUs have too many memory operations, leaving little room for extra memory operations needed for bounds checking. Hence, we introduce the following three techniques. First, we perform bounds checking per workgroup/warp/wave-front, not per individual thread, by computing the minimum and maximum address range from all threads in a workgroup. Second, we perform compiler-based static analysis to effectively reduce the number of runtime bounds checking. Third, we leverage a GPU-specific memory addressing mode to embed the bounds information in eligible memory operations, eliminating the need for explicit bounds accesses in the GPU memory hierarchy.

The contributions of our paper are as follows:

- We propose GPUShield, a region-based bounds-checking mechanism, to provide spatial memory safety for GPUs. To the best of our knowledge, GPUShield is the first hardware-based bounds-checking proposal for GPUs.
- We show that out-of-bounds writes are exploitable to incur memory corruption and are observable by the CPU under the CUDA SVM environment running on an Nvidia GPU.
- We utilize the unique GPU programming and memory models to reduce the overhead of bounds-checking mechanisms. Also, compiler-based static analysis can reduce the number of runtime bounds checking. As a result, GPUShield incurs negligible runtime overhead across 88 Nvidia CUDA and 17 Intel OpenCL benchmarks with low hardware overhead.

2 BACKGROUND

2.1 GPU Execution and Memory Models

GPUs maintain their high-throughput capability via massively parallel hardware and a hierarchical execution model. A GPU comprises multiple shader cores, each of which has multiple processing elements (PEs). The total amount of work is defined as a multi-dimensional compute domain. Each element in the domain is called a *workitem* (or a CUDA thread). Multiple workitems form a *workgroup* (or a CUDA thread block), and each workgroup can be scheduled to any core, while all workitems in the same workgroup will be scheduled to the same core. A workgroup is split into multiple sub-workgroups (or CUDA warps), the basic scheduling unit in a core for execution. Thus, the number of workitems in a sub-workgroup usually matches the number of PEs in a core.

A GPU uses the Single Instruction Multiple Threads (SIMT) execution model, i.e., all workitems in a sub-workgroup will execute the same instruction. Some workitems can be individually masked to skip execution mostly due to branch divergence. A sub-workgroup executes instructions in order. When a sub-workgroup encounters a long latency instruction, such as memory loads, the scheduler will switch to the next available sub-workgroup. In this way, GPUs can hide execution latency with their thread-level parallelism (TLP).

Typical GPU programming models adopt various memory types to improve memory latency and bandwidth. Most memory buffers are placed in globally visible memory upon a kernel launch. Thread-local variables reside in registers. Arrays that are too large or arrays with dynamic indices will be placed in local memory, which resides in costly device memory. Some data used by a workgroup can be brought into shared memory to avoid expensive global memory accesses. Some read-only buffers can utilize constant and texture memory. GPUs also employ a multi-level cache hierarchy.

Recent Nvidia GPUs [57] support dynamic memory allocation, while OpenCL has not supported that feature yet. Dynamically allocated buffers reside in the device's heap memory and are mainly used to store per-workitem intermediate results. The heap memory is persistent during the lifetime of a GPU context and shared between the kernels in the same GPU context. However, the performance overhead of dynamic allocation is significant because massive threads allocate the memory buffers in parallel.

Earlier GPU programming models used a host-accelerator computing model that required programmers to explicitly transfer data from/to the host processor. To reduce the programmer's burden, recent GPUs have adopted SVM, which allows the CPU and GPU to efficiently share their virtual memory space [1, 23, 52, 53]. In an integrated GPU, SVM can share even physical memory space. However, in a discrete GPU, SVM (or UM in CUDA) provides automatic data migration via demand paging by the GPU driver's page fault handler and the Input Output Memory Management Unit (IOMMU) [5, 19, 34, 66].

GPUs use a disciplined memory model [37] with a linear memory space (in one or more dimensions) that is 1) explicitly passed as kernel input arguments or 2) defined as GPU device memory. Programmers need to provide detailed information—size and read-only attribute—about memory buffers. Unlike CPU memory objects that can be freely allocated and deallocated at runtime, GPU memory buffers are usually allocated before a kernel launch and deallocated

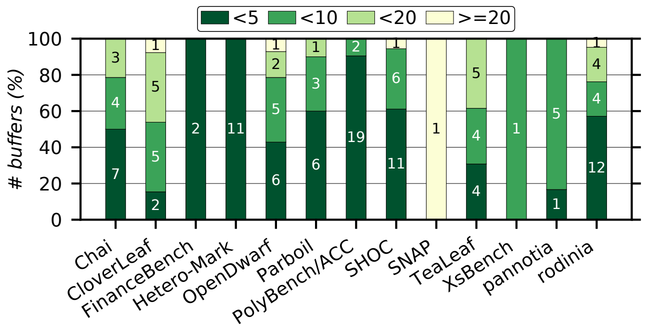


Figure 1: The distribution of the number of buffers in GPU benchmark suites (max: 34, avg: 6.5).

```
/* Method A. Binding table + offset:
      vaddr = BindingTable[BTI].base + offset */
LD dst, offset, BTI

/* Method B. Full virtual address: vaddr = src */
LD dst, src

/* Method C. Base address + offset: vaddr = base + offset */
LD dst, base, offset
```

Figure 2: GPU memory addressing methods.

after kernel completion. This memory model inevitably limits the number of buffers used in a GPU kernel. For example, in OpenCL 2.0 [32], the number of kernel arguments is limited to 128 in a 64-bit system, but a CPU program may spawn millions of memory objects. Figure 1 shows the distribution of the number of memory buffers in 145 GPU benchmarks across 13 suites. Most benchmarks use fewer than 10 buffers, and only five use more than 20.

2.2 Memory Addressing in GPU

GPUs use similar memory addressing methods to CPUs but have a unique addressing mode that utilizes graphics structures. Figure 2 shows three addressing methods used by GPUs: A) binding table (BT) indirect access, B) a 32-bit or a 64-bit virtual address, and C) a base address with an offset.

The first method uses a BT to store various buffer information, such as the base address and the size. This method is specific to graphics, where the BT efficiently shares various objects—textures and buffers—across pipeline stages [24, 71]. Methods B and C are commonly used in CPU programming models. All addresses are computed using the base address of a memory region and an offset. Methods A and C are similar except that the base address is stored in BT or in the register.

We compare how GPUs use different addressing modes using a vector add kernel code written in OpenCL, as shown in Figure 3. Intel GPUs have four addressing models: Binding Table State (BTS), Shared Local Memory (SLM), and 32/64-bit Stateless models [26]. The BTS model (Method A) uses a 256-entry BT. Intel GPUs use the *send* instruction to access the memory [25, 27]. The GPU driver assigns buffer IDs based on the order specified in kernel arguments, e.g., a:0, b:1, and c:2 in Figure 3a. Each *send* instruction has a message descriptor (fourth operand), and the eight least significant bits (LSB)

```
1  void add(global int *a, global int *b, global int *c)
2  {
3      int id = (int)get_global_id(0);
4      c[id] = a[id] + b[id];
5  }
```

(a) Vector addition kernel example code.

```
1  send r20:w r16 0xC 0x04205E00 // &a + r16
2  add r32.0<1>:d r12.0<8;8,1>:d r9.2<0;1,0>:d
3  send r28:w r24 0xC 0x04205E01 // &b + r24
4  send r22:w r18 0xC 0x04205E00 // &a + r18
5  send r30:w r26 0xC 0x04205E01 // &b + r26
6  ...
7  sends null:w r32 r20 0x8C 0x04025E02 // [&c+r32] = r20
8  sends null:w r34 r22 0x8C 0x04025E02 // [&c+r34] = r22
```

(b) Intel assembly code.

```
1  s_load_dwordx4 s[0:3], s[6:7], 0x0
2  s_load_dwordx4 s[12:15], s[6:7], 0x10
3  ...
4  v_add_co_u32_e32 v2, vcc, s0, v0 // v0: thread ID
5  v_addc_co_u32_e32 v3, vcc, v3, v1, vcc
6  global_load_dword v4, v[2:3], off
7  v_mov_b32_e32 v3, s3
8  v_add_co_u32_e32 v2, vcc, s2, v0
9  v_addc_co_u32_e32 v3, vcc, v3, v1, vcc
10 global_load_dword v2, v[2:3], off
```

(c) AMD assembly code.

```
1  MOV R1, c[0x0][0x28]
2  S2R R6, SR_CTAID.X
3  MOV R7, 0x4
4  S2R R3, SR_TID.X
5  IMAD R6, R6, c[0x0][0x0], R3
6  IMAD.WIDE R2, R6, R7, c[0x0][0x168] //c[0x0][0x168] = &b
7  LDG.E.SYS R2, [R2]
8  IMAD.WIDE R4, R2, R7, c[0x0][0x160] //c[0x0][0x160] = &a
9  LDG.E.SYS R5, [R4]
10 ...
11 IMAD.WIDE R6, R6, R7, c[0x0][0x170] //c[0x0][0x170] = &c
12 STG.E.SYS [R6], R9
```

(d) Nvidia assembly code.

Figure 3: Vector add code by various GPUs. Memory accesses are highlighted, with emphasis on BTIs and source registers.

of the message descriptor are a binding table index (BTI) to index BT to fetch the base address. For example, send instructions in lines 1 and 4 in Figure 3b access the buffer ID 0, which is a. Then, the buffer address is computed with the memory offset stored in one of the source registers.

AMD GPUs do not use BT and mainly use Methods B and C [3]. They have 32-bit scalar registers that are shared by a sub-workgroup and per-thread 32-bit vector registers. The GPU driver allocates a memory segment for kernel arguments and stores the segment address in one of the scalar registers. For example, in Figure 3c, s[6:7] stores the address of the kernel argument segment. s_load_dwordx4 in line 1 loads the base addresses of a (at offset 0x0) and b (at offset 0x8) in s[0:1] and s[2:3], respectively. Per-thread offset is added to the base address, and Method B is used in lines 6 and 10.

Nvidia GPUs mainly use Method B, but Method C is often used for texture and surface memory [56]. Kernel arguments are stored in constant memory. For example, c[0][0x160] stores the base address of a, and the per-thread offset is added to access a[id], as shown in line 9 in Figure 3d.

```
__global__ void kernel_overflow(int * A, int *B)
{
  // Case 1. OOB within a 512B boundary: suppressed
  A[0x10] = 0xBAD;
  // Case 2. OOB within a 2MB boundary: overflow
  A[0x80] = 0xBAD;
  // Case 3. OOB crossing a 2MB boundary: kernel aborted
  A[0x80000] = 0xBAD;
}

void main()
{
  int *A, *B; // each buffer is 512B aligned.
  cudaMallocManaged(&A, sizeof(int)*0x10);
  cudaMallocManaged(&B, sizeof(int)*0x10);
  kernel_overflow<<<1,1>>>(A,B);
  cudaDeviceSynchronize();
}
```

Figure 4: SVM buffer overflow examples on Nvidia CUDA. All buffers are allocated in consecutive 512B-aligned addresses.

3 GPU SECURITY

3.1 Memory Safety in GPUs

GPUs have become a major computing component, but the security measure is not as high as in CPUs. GPUs have been considered isolated and incapable of affecting the system since IOMMU manages confined device memory regions and prevents illegal access from a device I/O bus. However, their general-purpose adoption and tighter integration with CPUs in the same system, thanks to SVM, led GPU applications to process security-critical information, such as encryption keys and photographs. Recent studies [39, 43, 79] show that IOMMU-based isolation can be compromised by an attacker-controlled peripheral device, allowing it full access to the entire physical memory.

Among existing GPU vulnerabilities, we focus on memory safety because of its significant impact on systems. Arguably, memory safety vulnerabilities are one of the major threats to modern computer systems. Recent reports [17, 46] show that over 70% of all security issues addressed in the industry stemmed from memory safety violations. Also, prior studies reveal that most of the existing memory safety errors are exploitable in GPUs. Miele [45] presented that buffer overflows are feasible in Nvidia CUDA GPUs for statically (in the stack) and dynamically (in the heap) allocated buffers.[1] Di et al. [12] showed that stack overflow could influence the execution of other threads, and integer and function pointer overflow in a C/C++ struct data type is exploitable on GPUs. Recently, the mind control attack [61] demonstrated a practical attack on the deep learning frameworks by leveraging the buffer overflow vulnerabilities in GPU kernels to achieve arbitrary code execution, degrading the prediction accuracy of ML applications.

SVM buffers are also exploitable by overflow attacks. Figure 4 shows an example code that performs out-of-bounds writes. We identify that these illicit writes can be easily performed on an Nvidia CUDA GPU under the SVM environment with CPUs. We find that 1) out-of-bounds writes within a 512B boundary are suppressed, i.e., no side effect, because of a default 512B address alignment, 2) out-of-bounds writes within a 2MB memory are allowed, and 3) accesses crossing their 2MB boundary result in a kernel abortion with an illegal memory access error. Given that Nvidia GPUs use a 2MB-size page, we speculate that they use a 2MB-granularity protection scheme. Out-of-bounds writes are observable from the CPU side and can be used as a basis for other security attacks.

[1]Unlike CPU stack, a GPU stack in Nvidia CUDA GPUs resides in the off-chip local memory. A heap memory is allocated in the global memory.

Table 1: GPU memory types and their vulnerabilities.

Type	Scope	Location	Overflow Possibility
Register	Thread	On-chip	No
Local (stack)	Thread	Off-chip	Yes [12, 45, 61]
Shared	Workgroup	On-chip	Yes
Global	Application	Off-chip	Yes [14, 61]
Heap	Application	Off-chip	Yes [12, 13, 45, 62]
Constant	Application	Off-chip	No (read only)
Texture/Surface	Application	Off-chip	No (read only)
SVM	Application	Off-chip	Yes

Table 1 summarizes different GPU memory types and their known vulnerabilities. Unlike vulnerabilities in other memory types, access violations in the local memory occur *between different local variables within the same thread*, and one thread cannot access another thread's variable. This is because each local variable is organized in the local memory such that consecutive threads access consecutive 32-bit words, i.e., the same local variable from consecutive threads will reside in spatially adjacent memory [57]. The effective address is computed using the base address of a local variable and an index, which is a function of a thread-id.

3.2 Threat Model

We assume that adversaries can attack GPU kernels by injecting malicious inputs to invoke buffer overflow and perform illegal accesses to a victim kernel's memory regions. We do not limit the platforms on which GPU programs are executed. Thus, victim kernels can run on a personal desktop, data-center, or cloud environment, where clients or residents may simultaneously request multiple jobs that invoke GPU kernels on the shared GPUs. We assume that target GPU binaries are owned by victims, and native binary forging is prohibited by typical access control policies in operating systems (OSes). We also assume that the GPU driver is trustworthy using a kernel module signing facility [41]. Also, GPU hardware components are considered reliable. Hence, we consider the side- and covert-channel attacks that use resource contentions on hardware units [28, 29] or measure power consumption activity [42] out of the scope of this paper.

4 PRIOR ART ON CPU MEMORY SAFETY

Various CPU mechanisms have been proposed to secure computer systems against memory corruption attacks. Although software mechanisms typically provide strong safety, their significant performance overhead hinders their adoption as a runtime solution. We consider only hardware-based mechanisms since we aim to devise a low-overhead runtime GPU defense mechanism. In this section, we group these mechanisms into three categories and detail each category. Then, we describe the requirements for the GPU memory safety mechanisms to be practical.

Table 2: Comparing GPUShield with previous memory safety mechanisms.

Mechanism	Unit	Protection	No Register Extensions	No Duplicated Memory usage	No Extra Check Ops	Bandwidth Increase	Perf. Overhead
REST [67]	CPU	Canary		✓	✓	-	Low
Califorms [64]	CPU	Canary	✓	✓	✓	-	Low
ARM MTE [4], SPARC ADI [60]	CPU	Tag	✓	✓	✓	-	Low
Intel MPX [59]	CPU	Bounds checking		✓		High	High
HardBound [11], Watchdog [48]	CPU	Bounds checking				High	Moderate
CHERI [74–76][a]	CPU	Bounds checking		✓	✓	High	Moderate
In-Fat Pointer [78]	CPU	Bounds checking	✓	✓		High	Moderate
AOS [35]	CPU	Bounds checking	✓	✓	✓	High	Moderate
No-Fat [21]	CPU	Bounds checking	✓	✓	✓	-	Low
C3 [38]	CPU	Bounds checking	✓	✓	✓	-	Low
clArmor [14], GMOD [13]	GPU	Canary	✓	✓	✓	-	High
CUDA-MEMCHECK [54]	GPU	Bounds checking	✓	✓		High	High
GPUShield	GPU	Bounds checking	✓	✓	✓	Low	Low

[a]CHERI capability models support other security features, such as highly scalable software compartmentalization.

4.1 Canary-based Protection

This class inserts a canary, i.e., secret bytes, around objects to protect. Access to a canary is considered malicious. REST [67] embeds random tokens into programs and detects illegal accesses by having a token detector at the cache hierarchy. Califorms [64] enhances security by providing intra-object protection while alleviating memory overhead by utilizing padding bytes inserted for address alignment. This class achieves low performance overhead but has limited security coverage since non-adjacent accesses jumping over canary bytes cannot be detected.

Recent software-based proposals for GPUs [10, 13, 14] belong to this category. clArmor [14] intercepts OpenCL malloc calls to add a canary around allocated buffers. After a kernel finishes, it checks the canary to detect any bytes written beyond each buffer region. GMOD [13] runs concurrent guard threads that regularly monitor buffer overflows. However, in addition to their weak security guarantee, instantaneous error detection is difficult to achieve.

4.2 Memory Tagging

This approach places tags into pointers and enforces tag checking upon each pointer dereferencing. A memory access is allowed only when a pointer's tag matches that of the memory region being accessed. For example, Arm Memory Tagging Extension (MTE) [4] embeds a 4-bit tag into a pointer and associates it with a memory region pointed to by the pointer. Even though memory tagging approaches [4, 60] typically incur a marginal slowdown, the limited size of tag bits reduces the security coverage because of a high probability of false positives. Having more tag bits can enhance the security level, but it requires extensions of all relevant hardware units, such as tag cache, tag checker, and cache metadata.

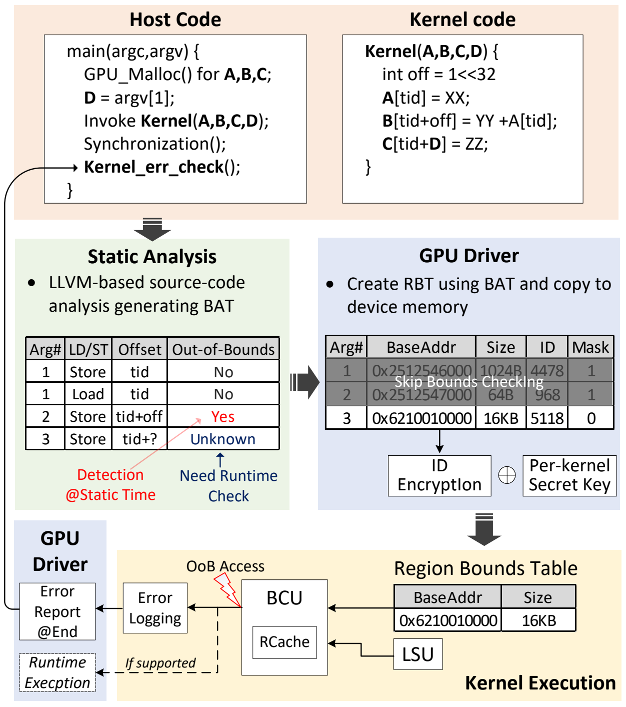
4.3 Bounds Checking

Bounds-checking mechanisms provide the strongest security guarantees. Intel MPX [59] associates bounds information with pointers and enforces bounds checking for memory accesses, but it incurs significant performance overhead due to bounds propagation and hierarchical bounds addressing. Some proposals [11, 48, 74–76] implement a hardware-assisted fat pointer, where a pointer includes bounds and permission metadata. Such an implementation requires register extensions (up to 256 bits) to hold metadata. On the other hand, to maintain pointer metadata in memory, HardBound [11] and Watchdog [48] implement shadow memory that mirrors allocated memory pages in a program and stores metadata in locations mapped to pointers' addresses.

Pointer tagging mechanisms [21, 22, 35, 38, 78] place a pointer tag in unused upper bits of pointers and use the tag to look up object metadata in memory or perform an object-based tag match. AOS [35] stores secret keys, instead of metadata, in the unused upper bits of pointers to avoid register extension and shadow memory. AOS uses the keys to index a bounds table stored in memory. It also develops microarchitectural extensions to eliminate explicit bounds-checking instructions. In-Fat pointer [78] stores object metadata along with in-memory type metadata and guarantees sub-object granularity protection. No-FAT [21] utilizes a binning allocator to expose memory allocation size to the architectural state and performs bounds checking using the implicit allocation bounds. For temporal safety, No-FAT places a pointer tag in the top bits of a pointer and verifies it upon a memory access. C3 [38] proposes a stateless memory-safety technique with a radix-bound pointer encoding scheme, removing the need for additional metadata.

4.4 Challenges in GPU Memory Safety

We compare the hardware-based mechanisms mentioned above in Table 2, in terms of hardware requirements, bandwidth increase, and performance overhead, with other software-based GPU mechanisms [13, 14]. Any practical memory safety mechanism for GPUs needs the following requirements. First, extending registers, as in some bounds-checking mechanisms [11, 48, 74–76], is not acceptable. Recent GPUs maintain huge register files to allow a massive number of concurrently running threads. For example, Nvidia Ampere GPU has a 256KB register file per core [55]. Extending already enormous register files can incur significant hardware overhead.



```
main(argc,argv) {
  GPU_Malloc() for A,B,C;
  D = argv[1];
  Invoke Kernel(A,B,C,D);
  Synchronization();
  Kernel_err_check();
}
```

```
Kernel(A,B,C,D) {
  int off = 1<<32
  A[tid] = XX;
  B[tid+off] = YY +A[tid];
  C[tid+D] = ZZ;
}
```

Arg#	LD/ST	Offset	Out-of-Bounds
1	Store	tid	No
1	Load	tid	No
2	Store	tid+off	Yes
3	Store	tid+?	Unknown

Arg#	BaseAddr	Size	ID	Mask
1	0x2512546000	1024B	4478	1
2	0x2512547000	64B	968	1
3	0x6210010000	16KB	5118	0

BaseAddr	Size
0x6210010000	16KB

Figure 5: System overview of GPUShield.

Second, GPU applications demand much higher memory bandwidth than CPU applications, so cache and memory bandwidth consumption have to be carefully managed. If a bounds-metadata access increases bandwidth usage, this will lead to significant performance loss due to massively parallel GPU computing capability.

Third, attacks become more sophisticated, so safety mechanisms with high coverage, such as memory tagging and bounds checking, are preferred to canary mechanisms.

All things considered, we aim to devise an efficient hardware-based GPU bounds-checking mechanism (for higher coverage) without using a fat pointer (for less hardware overhead) or shadow memory (for bandwidth savings).

5 GPUSHIELD

5.1 System Overview

Figure 5 shows the system overview. We maintain bounds metadata for each global buffer, local variable, and the entire heap memory. The metadata is stored in the newly introduced RBT (§5.2.3) in the GPU global memory, which is indexed by the buffer ID. The compiler statically analyzes all memory pointers in an application to identify the associated buffer and their types (§5.3). The compiler also performs static bounds checking for simple access types to filter out unnecessary runtime bounds checking (§5.3). Findings are stored in a *bounds-analysis table (BAT)* and passed to the GPU driver (§5.4), which assigns a random but unique ID for each buffer and embeds the encrypted ID in the pointer that holds the buffer base address. The driver also initializes the RBT.

```
struct Bounds {
  /* valid and readonly fields are physically stored in
     the upper bit of base_addr */
  //bool valid, readonly;
  uint64_t base_addr; // 48-bit virtual address
  uint32_t size;
};
```

Figure 6: Bounds metadata format.

When a sub-workgroup executes a memory instruction, we perform bounds checking in a new microarchitectural structure, the bounds-checking unit (BCU) (§5.5), located next to the load-store unit (LSU). The BCU comprises RBT cache (RCache) hierarchy and address range comparison logic.

5.2 Region-based GPU Bounds Checking

5.2.1 Protection Coverage. GPUShield implements efficient region-based bounds checking to protect the following GPU memory types: 1) host-allocated buffers that are passed as kernel arguments, 2) local variables, and 3) the entire heap memory chunk. As explained in Section 2.1, a GPU kernel can support up to 128 kernel arguments, so we need to maintain at most 128 entries in the bounds table for host-allocated buffers. In addition, we treat each local variable (i.e., kernel variables allocated in the local memory) as a separate buffer. Note that a kernel cannot maintain too many local variables because 1) local memory has a size limitation (512KB per thread), and 2) the size of constant memory that stores the base addresses of local variables is also limited (64KB shared by all threads).

For the heap memory, we adopt coarser grain protection. Recent Nvidia GPUs support each thread to allocate buffers at runtime, but the performance overhead of dynamic allocation would be significant when numerous threads allocate the memory buffers in parallel.[2] In this case, the number of dynamic buffers can be enormous, considering that a GPU can support numerous concurrent threads. The size of each dynamic buffer can vary for each thread, so protecting individual dynamic buffers in the heap memory is practically infeasible. For heap protection, we treat the entire heap memory as a buffer and maintain its bounds information in a single entry in the bounds table. The maximum size of the heap memory is preset by using the `cudaDeviceSetLimit` function with argument `cudaLimitMallocHeapSize` before a context creation.

5.2.2 Bounds Metadata. We maintain per-buffer bounds metadata, as shown in Figure 6. The `base_addr` stores the 48-bit[3] base address of a buffer whose size is stored in the 32-bit `size`. The `valid` and `readonly` fields are 1-bit each and embedded in the `base_addr`. Each buffer has a unique ID, assigned by the GPU driver.

5.2.3 Region Bounds Table. We store the bounds metadata of each buffer in a newly introduced per-kernel RBT. Since a GPU kernel uses a few buffers (6.5 buffers on average in Figure 1), a small-size

[2]We have conducted a study of GPU dynamic memory allocation and discovered that the performance overhead of using CUDA built-in `malloc()` ranges from a 4.9 to 63.7× slowdown. We use Nvidia RTX2080 with the following parameters: blocks per grid: 1K to 16K, threads per block: 1024, buffer size: 16 bytes.

[3]The size of the virtual address (VA) depends on the host CPU and the compute capability for Nvidia GPUs, e.g., 40-bit (Maxwell) and 47-bit (Pascal) under x86-64 [57]. Regardless, our mechanism can apply to different VA sizes.

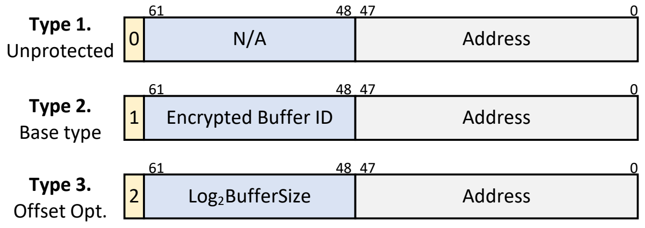


Figure 7: Pointer types used by GPUShield.

RBT can cover all bounds metadata. RBT is a 16384-entry direct-mapped structure indexed by a 14-bit buffer ID. The GPU driver allocates and updates RBT in the GPU global memory space upon a kernel launch (§5.4).

5.2.4 Embedded Metadata in Pointer. To access bounds metadata, inspired by pointer tagging [21, 22, 35, 38, 78], we use unused upper bits in the memory address to store a buffer ID. In this way, we can remove the need for hardware extension to store buffer IDs, and the embedded buffer ID will be propagated with any pointer arithmetic instruction and later used for bounds checking. This approach is less intrusive than extending pointers or registers to store bounds metadata.

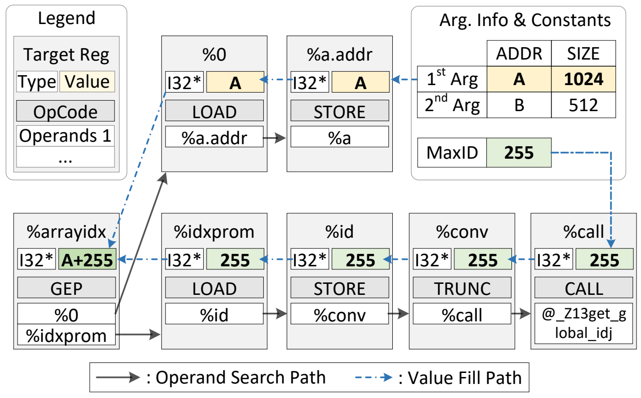
However, the pointer value is not naturally protected, so pointer forging attacks can exploit this method. Since the GPU driver assigns buffer IDs in ascending order based on the kernel argument order, GPUShield can be vulnerable against pointer forging if we use the originally assigned IDs. Considering only a smaller number of buffers (at most 34 buffers, as shown in Figure 1) with their IDs known from the source code, an attacker can maneuver the ID in a pointer to exercise an illegal write to the victim's benign buffer. To prevent this forging attack, we assign a 14-bit *random but unique* ID to each buffer.

Using random buffer IDs can obfuscate attackers but is still vulnerable since the original ID is exposed in the pointer. If the same kernel runs multiple times, an attacker can infer metadata embedded in pointers. Against this attack, we *encrypt* buffer IDs before storing them in pointers using the per-kernel encryption key. Without knowing the encryption key, attack attempts will lead to incorrect RBT access, resulting in bounds-checking failures and subsequent faults. Note that a new encryption key will be used for each kernel launch.

5.2.5 Pointer Formats. GPUShield uses three pointer types, shown in Figure 7. The two most significant bits (the `C` field) indicate the type of pointer, and the remaining 14 bits hold bounds information. When `C` is 0, which is set after compiler-based static analysis (§5.3), bounds checking will not be performed. When `C` is 1, this indicates that the pointer uses Method B addressing mode (full virtual address) in Figure 2, and the encrypted buffer ID is stored in the pointer. When `C` is 2, the pointer uses an optimized addressing Method C (base address with an offset) with the embedded buffer size information, which is explained in Section 5.3.3.

```
1 store i32 * %a, i32 * %a.addr, align 8
2 %call = call i64 @_Z13get_global_idj(i32 0) #4
3 %conv = trunc i64 %call to i32
4 store i32 %conv, i32* %id, align 4
5 %0 = load i32 *, i32 * %a.addr, align 8
6 %idxprom = load i32, i32 * %id, align 4
7 %arrayidx = getelementptr inbounds i32,i32*%0,i32 %idxprom
```

(a) LLVM IR example.



(b) Compiler-based data-flow analysis.

Figure 8: LLVM-based static bounds checking.

5.3 Compiler-based Static Analysis

We analyze all memory pointers by using the LLVM compiler framework [36] to identify the pointer type and perform static bounds checking.

5.3.1 Pointer Type Identification. The compiler first identifies the type of each pointer, i.e., which memory it will access. In particular, we aim to identify pointers that access global memory (host-allocated buffers and dynamic memory) and local memory.

5.3.2 Static Bounds Checking. Figure 8 shows how we perform the analysis using an LLVM intermediate representation (IR) example. To identify the pointers for which bounds checking can be done at compile-time, we perform a data-flow analysis. First, we look for `GetElementPtr` (`GEP`) instructions, which indicate the address of memory operations. A `GEP` instruction (line 7 in Figure 8a) requires one destination (`%arrayidx`) and two source operands, a base address (`%0`) and an index (`%idxprom`). From this instruction, we construct an *operand tree.* We search the producer (dependent) instructions of all operands through the instruction dependency chain. In this way, we can find the base address (`%a.addr` in line 1) of the `GEP` instruction.

Then, we perform reverse traversal on the *operand tree* from the leaf node to the root. At this time, we try to fill the value of the operand, which we can identify from the host code analysis or the given maximum value. For example, the maximum return value of `get_global_id` function in line 2 is `CL_DEVICE_MAX_WORK_GROUP_SIZE`. When the search reaches the root node, i.e., one of the operands of `GEP`, we perform bounds checking if the value can statically be known. Otherwise, we rely on dynamic bounds checking.

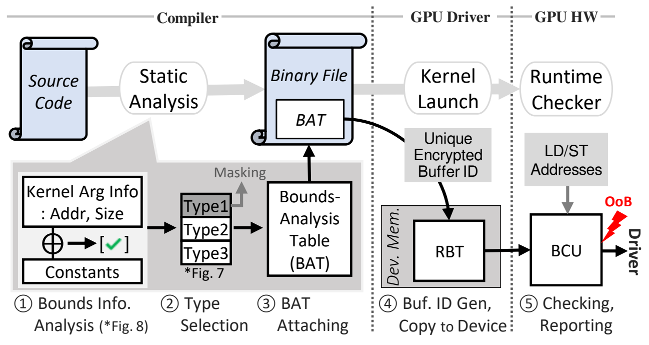


Figure 9: GPUShield operation flow.

After the static bounds checking is completed, we log the findings (pointer types) in the *bounds-analysis table* (see Section 5.4). We report overflow errors to the user immediately after this step. For pointers whose bounds checking was successfully done, we set them to Type 1 (see Figure 7) to avoid runtime bounds checking. Other pointers will become Type 2. Finally, the binary-format bounds-analysis table is attached to the binary and later used by the GPU driver upon kernel launch.

5.3.3 Optimization. We can optimize the Method C addressing in Figure 2 to bypass RBT accesses. Instead of a buffer ID, we embed the buffer size in a pointer and compare the address offset with the embedded size. If the offset is negative or exceeds the size, these are out-of-bounds memory accesses.

However, naïvely storing the size limits this optimization to buffers whose size is less than 16KB. Instead, we can enforce all buffers to always be aligned with the power-of-two byte address and extend support for any buffer sizes using log2 of the size. When the buffer size is not the power-of-two, we can add padding bytes until the next power-of-two address. Thus, this approach inevitably incurs memory fragmentation. This optimization is similar to Guarded Pointer [6], but we can add a canary in the padding bytes [13, 14] to detect out-of-bounds writes to this region after the kernel execution, enabling a capability-based addressing [6, 16, 75] for this type of pointer without extending pointers and registers.

It is worth noting that we can rarely find addressing Method C from the analysis, except for Intel GPUs. As explained in Section 2.2, Method A (send instructions) will be the same as Method C if we can use registers, instead of BT, to store the base address. Consequently, we make these pointers Type 3.

5.4 Kernel Setup by GPU Driver

The GPU driver is responsible for setting up GPU kernels upon their launch. Figure 9 depicts the overall procedure of how the GPU driver obtains information from the compiler and sets up GPU kernels in GPUShield.

The GPU driver uses the *bounds-analysis table* that is attached in the binary from the compiler analysis for bounds checking. The GPU driver also allocates RBT in the GPU memory and copies bounds metadata into RBT. To prevent illegal accesses to RBT from different kernels, the driver stores the physical address of RBT for all cores the kernel will be running on and makes the corresponding pages inaccessible. RBT accesses in GPU cores will bypass the address translation, while all other normal accesses require the translation and fail if they try to access RBT because of the permission.

```
void Memory::UpdateBnds(Device *pDevice, uint64_t base, int sz,
    bool ronly)
{
  int id = assign_buffer_id();
  m_RBT[id].base_addr = (base | (1<<63) | (ronly<<62));
  m_RBT[id].size = sz;
  pDevice->VkInstance()->PalPlatform()->updateBnds(m_RBT
    );
}

// AMD OpenSource Vulkan Driver
VkResult Memory::Memory(...)
{
  ...
  m_RBT = new Bounds[16384];
  // Allocate Bound table data on GPU Memory
  Pal::GpuMemoryResourceBoundTableData(m_RBT);
  ...
}

VkResult Memory::Create(Device * pDevice,
    VkMemoryAllocateInfo* pAllocInfo, ...)
{
  createInfo.size = pAllocInfo->allocationSize;
  // allocate physical memory in Driver
  CreateGpuMemory(...,createInfo, &pMemory);
  ...
  // Generate per-kernel key
  uint64_t kernel_secret_key = key_generation();
  uint64_t kernel_secret_id = id_generation();
  ...
  // Update the Bounds Table
  UpdateBnds(pDevice, pMemory->base, pMemory->m_size, 1)
  ...
}
```

Figure 10: GPU kernel driver code example to set up RBT.

The driver assigns a random but unique ID to each buffer and local variable and embeds it in the pointer that holds the base address, for example, constant memory (Nvidia) or registers (AMD), as explained in Section 2.2. Because of security concerns, we encrypt the ID before embedding it (see Section 5.2.4). The encryption key will be stored in the GPU cores to decrypt buffer IDs before performing bounds checking. For the heap memory, the driver assigns the ID and allocates a single entry in RBT. Upon dynamic buffer allocations, this preassigned ID will be embedded into the pointer. Figure 10 shows how to set up RBT using AMD opensource Vulkan driver [2].[4] The `Memory::create` function is called to set up the GPU device memory.

5.5 Bounds-Checking Unit

We introduce a new microarchitectural structure, the bounds-checking unit (BCU), to perform bounds checking. The BCU is located next to the load-store unit (LSU) and comprises the RBT cache (RCache) hierarchy, address gathering unit, and address range checking logic.

[4]We choose AMD's Vulkan driver source code as the base platform to explain how a GPU driver can be modified since it is the latest open-source GPU driver. Both Intel and Nvidia do not release their driver source code.

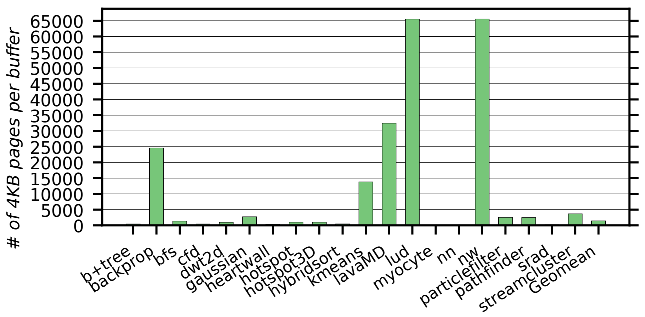


Figure 11: Number of 4KB pages per buffer in Rodinia suite.

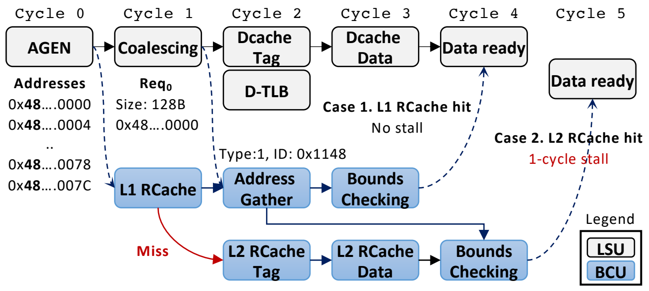


Figure 12: BCU pipeline stages with one coalesced 128B request.

For efficient bounds checking, we introduce the RCache hierarchy next to the L1 data cache. Storing all bounds metadata in the L1 data cache only costs several cache lines, but this will double cache bandwidth pressure since every memory access now needs one additional L1 cache access. The L1 RCache is a first-in, first-out (FIFO) queue and performs parallel tag lookups and data reads upon its access. The L2 RCache is a 64-entry fully associative structure, which can sufficiently cover all buffers in all evaluated benchmarks. The L2 RCache is physically split into tag and data arrays, similar to other cache structures.

We store virtual base addresses in RBT so that bounds checking in a core can overlap with address translation. Otherwise, bounds checking will be serialized with the address translation, lengthening the critical path. We store 14-bit buffer IDs in the RCache tag array, and each data array entry has the following fields:

- Base address (48-bit): the virtual base address of the buffer.
- Size (32-bit): the size of the buffer.
- Read-only (1-bit): to indicate if the buffer is read-only.
- Kernel ID (12-bit): the kernel ID.

When a sub-workgroup executes a memory instruction, BCU performs bounds checking along with the LSU pipeline. We first try to get bounds data from an RCache. The L2 RCache is large enough to avoid any miss during kernel execution other than initial misses. Even if the L2 RCache is not big enough for some cases, GPU kernels access most memory regions with only a few memory buffers. Hence, the TLB misses will occur much more frequently than L2 RCache misses. For example, Figure 11 shows that one buffer touches 1425 4KB pages on average in Rodinia suite [7], while we estimate 6.6 pages on average for SPEC CPU2006 benchmarks. Thus, the RCache miss latency will overlap with the TLB miss latency. Initial L2 RCache misses will be serviced from RBT using the physical address of RBT stored in the GPU core (§5.4) and a buffer ID as an offset while conforming to the underlying memory consistency and coherency models.

We can dramatically reduce the dynamic power consumption of the L2 RCache with a much smaller L1 RCache. Figure 1 shows that 55.9% of benchmarks have fewer than five buffers. Also, the *lock-step execution model* of the GPU improves the temporal locality of bounds metadata accesses. If a sub-workgroup encounters a long-latency instruction, such as branch and memory, the GPU core switches to another sub-workgroup to hide latency. As a result, a workgroup in the same GPU core tends to execute spatially adjacent memory instructions. This execution model makes a small (4-entry) L1 RCache effective.

L1 RCache hits entirely avoid the L2 RCache accesses. Upon an L1 RCache miss, we access the L2 RCache tag array. After finding a matching tag from the tag array, the corresponding bounds data will be fetched in the next cycle and added to the L1 RCache. Note that the RCaches will be flushed upon a kernel termination or a context switch.

5.5.1 *BCU Pipeline Stages.*

Figure 12 shows the combined LSU and BCU pipeline stage diagram. When a sub-workgroup executes a memory instruction, the instruction is sent to LSU. Virtual addresses are generated by an address generation unit (AGU), and the address coalescing unit (ACU) merges adjacent addresses into a small number of larger-size memory transactions. Based on the memory access pattern, one memory instruction from a sub-workgroup can generate a single coalesced transaction or multiple transactions. The data TLB (D-TLB) and the L1 data cache (Dcache) tag are accessed in parallel, and the Dcache line will be read out in the next cycle in the case of D-TLB and Dcache tag hits.

The BCU pipeline comprises address gathering, RCache access, and address range checking. The AGU sends the *encrypted* buffer ID and the pointer type from a pointer to the BCU. Type 1 pointers bypass bounds checking. For Type 2 pointers, the buffer ID is *decrypted* using the per-kernel encryption key in the BCU, and the L1 RCache will be accessed. An L1 RCache miss initiates an L2 RCache access. Thus, an RCache access has one- (L1 hit) or three-cycle (L2 hit) latency. The BCU performs bounds checking for Type 3 pointers without RCache accesses.

The address gathering pipeline computes the *minimum* and *maximum* address pair for bounds checking from the addresses the ACU sends. The BCU pipeline incurs one cycle penalty only if a memory instruction results in a single transaction that hits in Dcache but misses in the L1 RCache, as shown in Figure 12. For all other cases, bounds-checking latency will be hidden by the LSU pipeline latency.

5.5.2 *Handling Bounds-Checking Failure.*

When the BCU detects a bounds-checking failure, i.e., illegal memory access, it can immediately raise a fault if a GPU supports a precise exception. Otherwise, the BCU logs the error and returns zero for loads or drops stores silently. Memory safety violations can be reported 1) at the end of a kernel execution or 2) at runtime by signaling the host using

Table 3: Area and power overhead by GPUShield.

Structure	# of Entry	SRAM (Byte)	Area (mm^2)	Leakage (μW)	Dynamic (mW)
Comparators	-	-	0.0064	17.51	20.41
L1 RCache	4	53.5[a]	0.0060	26.40	22.93
L2 RCache tag	64	112	0.0166	256.71	55.39
L2 RCache data	64	744	0.0568	499.13	104.63
Total	-	909.5	0.0858	799.75	203.36

[a]4 entries × (14b ID + 48b base addr. + 32b size + 1b read only + 12b kernel ID)

Table 4: Security coverage by GPUShield.

Type	Coverage
Host-allocated buffers	Isolation guaranteed per each buffer
Local memory	Isolation guaranteed between threads
Heap memory	Isolation guaranteed between kernels

an SVM buffer that the CPU and GPU share before a GPU kernel finishes its execution.

5.6 Hardware Overhead

To estimate the area and power overhead of GPUShield, we synthesize the additional comparator logic and caches using a Synopsys design compiler [70] at the 1 GHz clock frequency. We implement the comparator using Verilog and use SRAM models generated from OpenRAM [18] for caches. We use 45nm FreePDK libraries [68] for synthesis. Table 3 shows that GPUShield incurs only modest hardware overhead. The total overhead across all GPU cores is 14.2KB and 21.3KB for Nvidia and Intel GPUs, respectively, based on the configurations in Table 5.

5.7 Security Coverage

Table 4 summarizes the security coverage by GPUShield described in Section 5.2.1. Most prior studies exploit memory buffers specified as kernel arguments, i.e., host-allocated buffers, to initiate overflow attacks. For example, a recent work, the mind control attack [61], demonstrates an attack scenario on the existing Deep Neural Network (DNN) server system. It consists of three phases: setup, search, and downgrade. During the setup phase, the attack utilizes the buffer overflow by injecting a malicious payload through kernel arguments, which causes memory overwrites in a global memory buffer that contains a function pointer or return address. Then, it hijacks the control flow, enabling ROP [65] to run arbitrary code. Since we protect each host-allocated buffer, GPUShield prohibits the initial setup phase, and further steps become unfeasible. In this way, GPUShield can mitigate this type of attack.

However, the effectiveness of GPUShield can be limited against more futuristic, sophisticated, fine-grained attacks. In our current design, supporting fine-grained protection for dynamically allocated buffers may impose additional performance overhead and require extra hardware support. For example, having too many buffers can cause RCache thrashing, incurring pipeline stalls and memory bandwidth increase. Alternatively, the protection could be extended 1) by replacing dynamic buffers with pre-allocated buffers, which also improves performance while potentially consuming more memory, or 2) by using software-based bounds checking. We leave supporting fine-grained protection as future work.

6 DISCUSSIONS

6.1 RBT Attacks

The GPU driver allocates and updates per-kernel RBT in the GPU memory upon kernel launch (§5.4). We prevent illicit RBT accesses from different kernels by storing the physical address of RBT in all cores that the kernel will run and making corresponding pages inaccessible.

An attacker may try to manipulate a victim application's pointers via a pointer-forging attack. As explained in Section 5.2.4, we mitigate this attack by using a 14-bit random encrypted buffer ID with a per-kernel encryption key. Reading an invalid RBT entry leads to a fault, so brute-force attacks are not feasible considering encryption entropy.

6.2 Concurrent GPU Kernel Executions

Recent GPU architectures allow multiple kernels to run concurrently on the same GPU [77]. GPUShield can be extended to support multi-kernels in the following ways:

1) Inter-core sharing: Multiple kernels from the same GPU application can simultaneously run on the same GPU by splitting cores, i.e., one kernel on the first half of the cores and the other on the other half. Since RBT is maintained per kernel, and a core is occupied by one kernel, GPUShield can work as is without incurring any performance overhead.

2) Intra-core sharing: Recent GPUs support fine-grained core slicing, i.e., multiple kernels can share the same core. Although per-kernel RBT is not affected, kernels will share the same L1 and L2 RCaches in a core. Since RCaches already have a kernel ID field, GPUShield will work without a problem. To mitigate potential performance degradation due to the reduced effective RCache size, we can double and partition RCaches, i.e., bank-level partitioning. Kernels will access their partitions based on the warp scheduler position [77].

3) Fine-grained context switching [73] will not affect the bounds-checking capability of GPUShield. As explained in Section 5.5, RCaches are flushed upon context switching or kernel termination. If the TLB is also flushed, any RBT miss latency will be amortized with TLB-miss latencies.

6.3 Number of Memory Buffers in GPU

The maximum number of buffers that GPUShield can protect is bounded by the number of unused bits in virtual addresses. Although typical GPU kernels have fewer than 20 buffers, as shown in Figure 1, one could also argue that having multiple concurrently running kernels, too many local variables, or future programming models might increase the number of active buffers. Since bounds metadata is maintained per kernel, having multiple kernels would not cause this problem. The dynamic parallelism supported by device kernel invocation from a parent thread has a very strict rule for memory usages in a kernel, so it would not increase the number of active buffers. As also discussed in Section 5.2.1, the number of

```
1 __kernel void kmeans_kernel_swap(...)
2 {
3   unsigned int tid = get_global_id(0);
4   if (tid < npoints) // SW bounds checking
5     for (int i = 0; i < nfeatures; i++)
6       feat_swap[i*npoints+tid] = feat[tid*nfeatures+i];
7 }
```

Figure 13: Software bounds checking in OpenCL kmeans.

Table 5: Configuration of the simulated system.

	Nvidia-GPU Configuration
Core	16 SMs, 1.6 GHz, 1024 threads per SM, 256KB register files per SM
Private L1 Cache	16KB, 4-way, LRU
Private L1 TLB	64 entries per core, fully associative, LRU
	Intel-GPU Configuration
Core	24 Cores, 1GHz, 7 HW threads per core, integrated GPU model
Private L1 Cache	32KB, 4-way, LRU
Private L1 TLB	64 entries per core, fully associative, LRU
	Memory Configuration
Shared L2 Cache	2MB total, 16-way, LRU
Shared L2 TLB	1024 entries total, 32-way associative, LRU
Memory	2KB row buffer, FRFCFS policy, 16 channels

local variables cannot be too high. A future programming model that increases the number of kernel arguments would require more significant hardware changes for various other changes. More importantly, since the GPU driver is responsible for maintaining buffer ID generation and RBT allocation, we could enhance the driver algorithm to adapt to programming model changes. For example, when the driver detects that the number of remaining buffer IDs is running low, GPUShield can enforce two adjacent buffers to share the same buffer ID and the merged bounds metadata. In sum, GPUShield is capable of handling these cases.

6.4 Replacing Software Bounds Checking

Bounds-checking code is often placed in the program [20] to prevent overflows. Bounds checking in *If* clauses is quite common, as shown in Figure 13 line 3. This *If*-statement is executed by all workitems. We measure that the performance overhead could be up to 76% from 1) increased instruction counts and 2) potential control-flow divergence if overflow exists, i.e., the number of threads exceeds the buffer size. GPUShield can perform hardware-based bounds checking instead of adding bounds-checking code in a program. Note that we do not implement this optimization in GPUShield and leave it as future work.

7 EVALUATION METHODOLOGY

We use MacSim [33], a cycle-level microarchitecture simulator. The simulator supports virtual memory. We evaluated both the Nvidia and Intel GPU architectures. Table 5 shows the simulation parameters used in evaluations. We used both CUDA and OpenCL applications for simulations. For CUDA, among 88 benchmarks from Rodinia [7], Parboil [69], GraphBig [49], and CUDA-SDK,

Table 6: Evaluated benchmarks (the italic font indicates the RCache-sensitive benchmarks).

Domain (Abbr.)	**Benchmarks**
Machine learning (ML)	mm, *convolution (ConvSep)*, kmeans, backprop
Linear Algebra (LA)	sad, spmv, stencil, *Scalarprod*, vectoradd, dct, Reduction
Graph-traversal (GT)	*between centrality (bc), bfs-dtc, graph coloring (gc-dtc), sssp-dwc*, lavaMD, gaussia, *nn*
Graph-iterative (GI)	pagerank, kcore, traingle count
Phys. and modeling (PS)	cutcp, tpacf, blacksholes, mersennetwister, sorting, *MergeSort*
Image and media (IM)	mri-q, *sobolQRNG*, Dct, DwtHarr, hotspot, *lud*, LineOfSight, Dxtc, *Histogram*, HSOpticalFlow
Data mining (DM)	*streamcluster*, *nw*
OpenCL	backprop, bfs, Bitonicsort, GEMM, Image, lavaMD, MedianFilter, cfd, MonteCarlo, pathfinder, svm, hotspot, hotspot3D, hybridsort, kmeans, nn, streamcluster

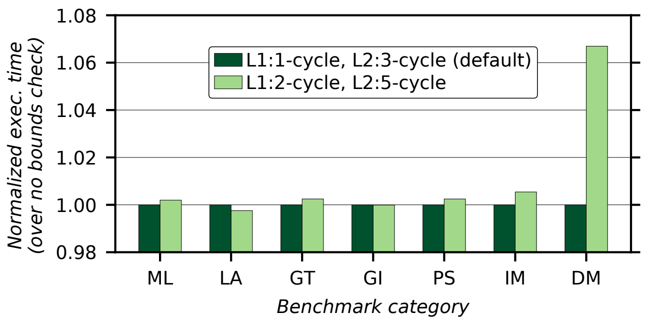


Figure 14: Performance results per category (normalized to the no bounds checking baseline). The legend shows the L1 and L2 RCache latencies, e.g., L1:1 L2:3 has 1- and 3-cycle latency for the L1 and L2 RCaches, respectively.

we categorized benchmarks per domain, as shown in Table 6. We collected OpenCL traces for Intel GPUs using GT-Pin [30].

8 RESULTS

8.1 Nvidia GPU Architecture Evaluations

Figure 14 shows the performance overhead of GPUShield compared to the baseline (no bounds checking). We show per-category average performance results. In the default GPUShield configuration, we use a 4-entry, 1-cycle L1 RCache and a 64-entry, 3-cycle L2 RCache. As explained in Section 5.5.1, we run into one cycle performance penalty only upon an L1 RCache miss with the L1 data cache hit. Even if we encounter pipeline bubbles introduced by bounds checking, abundant thread-level parallelism (TLP) in GPUs can tolerate the penalty to some degree. As a result, we observe that benchmarks in all categories do not show performance degradation, and most benchmarks exhibit close to 100% L1 RCache hit rate.

We measured the latency sensitivity of the L1 and L2 RCaches. Most benchmarks do not show any performance degradation if the

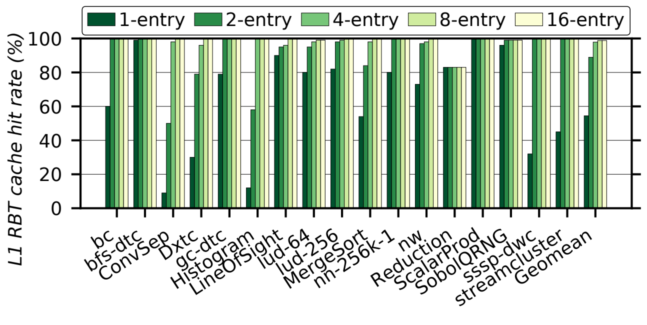


Figure 15: L1 RCache size sensitivity result.

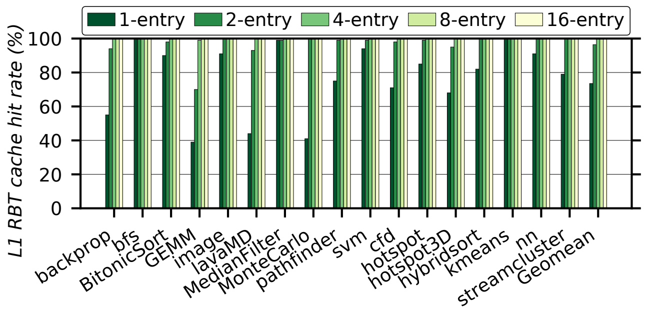


Figure 16: L1 RCache hit rate on Intel GPU evaluations.

L1 RCache latency is less than three cycles, which is one cycle less than the LSU pipeline. We identify that `streamcluster` in the `DM` category shows the worst performance degradation. Unlike other benchmarks, it has a massive number of memory requests, which mostly hit in the L1 Dcache, and adding one cycle pipeline bubble degraded its performance significantly.

Figure 15 shows the L1 RCache hit rate of 17 RCache-sensitive benchmarks as we increase the size from 1 to 16 entries. For most benchmarks, 4-entry sufficiently covers all buffers, resulting in close to a 100% hit rate, which is not surprising since GPU kernels typically have a small number of buffers, as shown in Figure 1. Furthermore, sub-workgroups in the same GPU core can exploit strong temporal locality in the buffer accesses (see Section 5.5).

8.2 Intel GPU Architecture Evaluations

We also evaluated GPUShield on the Intel GPU architecture. Unlike Nvidia GPUs, the Intel GPU architecture has fewer hardware threads (7) but uses vectorization to sustain high throughput. Similar to the results on the Nvidia architecture in the previous section, Figure 16 shows that most of the benchmarks show a near 100% hit rate with a 4-entry L1 RCache, leading to negligible performance degradation.

8.3 Benefits of Static Code Analysis

GPUShield incurs little performance overhead, but we can further reduce it with static bounds checking. While static analysis costs a few tens of milliseconds of compilation time, it can significantly improve performance for applications whose dominant memory access patterns use a base address with thread ID or block ID indexing. Figure 17 shows the performance benefits on the Nvidia GPU architecture when we apply compiler-based static bounds checking. Note that we choose two configurations in which RCache latencies are longer than the baseline because the baseline does not incur performance degradation.

First, static bounds checking (`+static` in the legend) reduces the performance overhead even with one cycle extra latency in the L1 RCache pipeline (`L1:2`). Second, we identify that applying static bounds checking completely removes the necessity of runtime bounds checking for some benchmarks with simple addressing. However, graph benchmarks—`bc`, `bfs-dtc`, `gc-dtc`, `sssp-dwc`, and `nw`—use many indirect memory accesses, which limit the applicability of static time analysis. This result affirms the need for dynamic bounds checking even with the GPU's distinct programming and memory models.

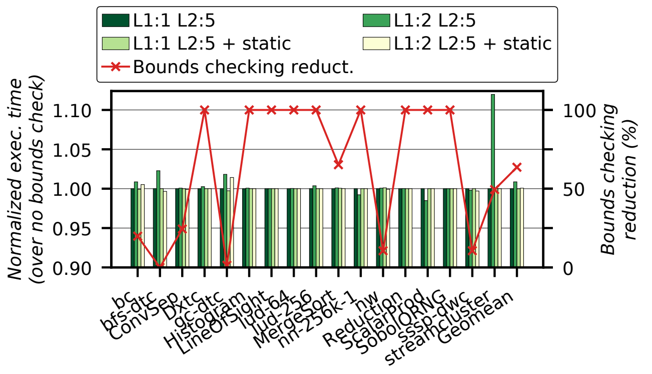


Figure 17: The effect of static time bounds-checking filtering.

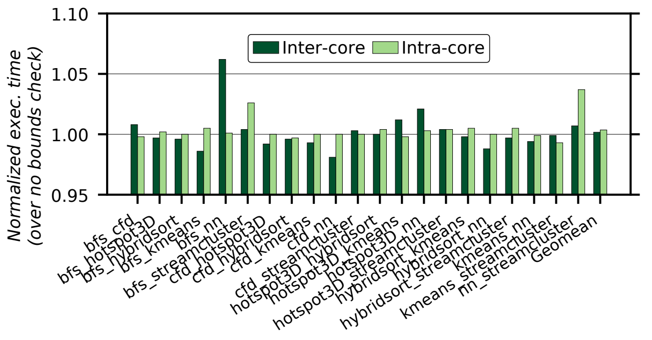


Figure 18: Multi-kernel execution results.

8.4 Multi-kernel Execution Results

To see how GPUShield performs when we run multiple kernels simultaneously on the same GPU, we run 21 combinations of two benchmarks on the Intel GPU architecture. We test two modes described in Section 6.2: 1) inter-core (one kernel runs on core 0 to 11, and the other runs on core 12-23) and 2) intra-core (kernels can share any core). Figure 18 shows that the average performance overhead of multi-kernel execution is under 0.3% for both cases. When we run memory-intensive benchmarks, such as `bfs`, `nn`, and

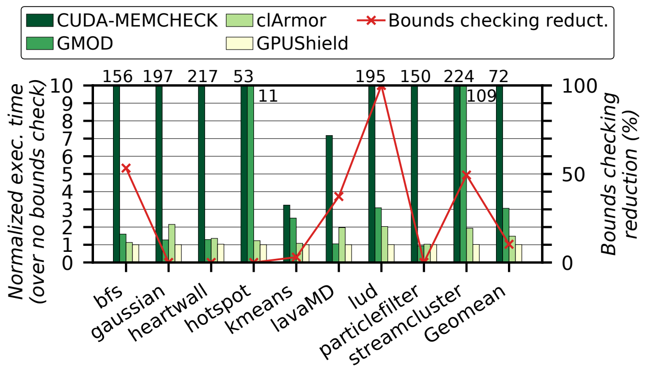


Figure 19: Software bounds-checking performance overhead and bounds-checking reduction ratio by static analysis.

`streamcluster`, we could see 6.2% performance degradation (`bfs` and `nn` in inter-core mode) over no bounds-checking case. We can mitigate this performance loss using static bounds checking.

8.5 Performance Overhead of GPU Buffer Overflow Detection Tools

Various overflow detection tools also exist. For example, Nvidia's CUDA-MEMCHECK [54] is a runtime memory error detector for out-of-bounds and misaligned accesses. clArmor [14] and GMOD [13] are canary-based buffer overflow detectors.

Figure 19 shows the performance results of these tools across Rodinia benchmarks [7]. CUDA-MEMCHECK, clArmor, and GMOD incur 72.3×, 3.1×, and 1.5× overhead on average, respectively, while GPUShield shows only a 0.8% slowdown. As expected, canary-based mechanisms (GMOD and clArmor) perform better than CUDA-MEMCHECK. `streamcluster` shows the highest overhead by both CUDA-MEMCHECK and GMOD (224× and 109.2×, respectively). This is because 1) it has a high percentage of load/store instructions (31.22%) that are instrumented for bounds checking by CUDA-MEMCHECK, and 2) GMOD has a software structure that enforces users to call constructor/destructor upon all kernel launches, which incurs significant performance degradation when an application frequently invokes a kernel, e.g., `streamcluster` conducts 1000 kernel invocations.

The high performance overhead in software tools is due to 1) just-in-time (JIT) binary instrumentation, 2) extra instrumented instructions, 3) extra load instructions for bounds metadata, and 4) software-based bounds-checking operations. NVBit [72], which takes a similar approach to CUDA-MEMCHECK, shows up to 20% overhead from JIT binary instrumentation and 112× for running instrumented binary for memory profiling. Compile-time binary instrumentation would reduce the performance overhead, but the overhead will still be much higher than CPU because additional register usage and increased memory footprint may reduce GPU occupancy, i.e., less throughput.

Static analysis used in GPUShield can be applied to these software schemes to alleviate the performance overhead. For example, we expect the performance of `bfs`, `lud`, and `streamcluster` to significantly improve with this optimization thanks to the high runtime bounds checking reduction rate (53.3%, 100%, and 49.4%, respectively). However, not all benchmarks can benefit from this optimization. For example, graph applications have many indirect memory accesses, which solely rely on runtime bounds checking. GPUShield would still outperform software tools with static bounds checking optimization thanks to efficient hardware-based bounds checking.

9 CONCLUSIONS

This paper proposed GPUShield, which is the first hardware-based GPU bounds-checking mechanism to provide spatial memory safety. We first compared addressing methods by various GPUs. We also demonstrated that out-of-bounds writes can be easily done on Nvidia CUDA SVM. Thanks to GPU's unique programming and execution models, GPUShield can implement an efficient region-based bounds-checking mechanism. Our evaluations showed that GPUShield incurs negligible performance overhead. We also described how compiler-based static bounds checking can be performed to reduce unnecessary runtime bounds checking. With increasingly complex GPU applications introduced by the wide adoption of GPUs, we believe in the value of our proposed mechanism for secure GPU computing.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their insightful feedback to improve the paper. We also thank Sunpyo Hong, Aamer Jaleel, Sam Jijina, Andrei Bersatti, and other HPArch members for their support and suggestions. We gratefully acknowledge support from Intel.

REFERENCES

[1] AMD. 2012. AMD Graphics Cores Next (GCN) Architecture. https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf.
[2] AMD. 2018. AMD Vulkan® Open Source Driver. https://github.com/GPUOpen-Drivers/AMDVLK.
[3] AMD. 2020. RDNA 1.0 Instruction Set Architecture Reference Guide. https://developer.amd.com/wp-content/resources/RDNA_Shader_ISA.pdf.
[4] Arm. 2020. Arm®Architecture Reference Manual Armv8, for Armv8-A architecture profile. https://developer.arm.com/docs/ddi0487/fb/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile.
[5] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J. Rossbach, and Onur Mutlu. 2018. MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency. In *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, New York, NY, USA, 503–518.
[6] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. 1994. Hardware Support for Fast Capability-Based Addressing. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, New York, NY, USA, 319–327.
[7] 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)*. IEEE, Piscataway, NJ, USA, 44–54.
[8] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-Oriented Programming without Returns. In *Proceedings of the 17th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Association for Computing Machinery, New York, NY, USA, 559–572.
[9] Crispin Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium (Security)*. USENIX Association, USA, 1–15. https://www.usenix.org/conference/7th-usenix-security-symposium/stackguard-automatic-adaptive-detection-and-prevention

[10] Datalogisk Institut. 2020. The Futhark Programming Language. https://futhark-lang.org.

[11] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. 2008. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. In *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, New York, NY, USA, 103–114.

[12] Bang Di, Jianhua Sun, and Hao Chen. 2016. A Study of Overflow Vulnerabilities on GPUs. In *IFIP International Conference on Network and Parallel Computing (NPC)*. Springer International Publishing, Cham, 103–115.

[13] Bang Di, Jianhua Sun, Dong Li, Hao Chen, and Zhe Quan. 2018. GMOD: A Dynamic GPU Memory Overflow Detector. In *Proceedings of the 27th ACM International Conference on Parallel Architecture and Compilation Techniques (PACT)*. Association for Computing Machinery, New York, NY, USA, 1–13.

[14] Christopher Erb, Mike Collins, and Joseph L Greathouse. 2017. Dynamic buffer overflow detection for GPGPUs. In *Proceedings of the 15th International Symposium on Code Generation and Optimization (CGO)*. IEEE, Piscataway, NJ, USA, 61–73.

[15] Christopher Erb and Joseph L. Greathouse. 2018. ClARMOR: A Dynamic Buffer Overflow Detector for OpenCL Kernels. In *Proceedings of the International Workshop on OpenCL (IWOCL)*. Association for Computing Machinery, New York, NY, USA, Article 15, 2 pages. https://github.com/ROCm-Developer-Tools/clARMOR.

[16] Robert S. Fabry. 1974. Capability-based addressing. *Communications of the ACM* 17, 7 (1974), 403–412.

[17] Google. 2017. Google Queue Hardening. https://security.googleblog.com/2019/05/queue-hardening-enhancements.html.

[18] Matthew R. Guthaus, James E. Stine, Samira Ataei, Brian Chen, Bin Wu, and Mehedi Sarwar. 2016. OpenRAM: An Open-Source Memory Compiler. In *Proceedings of the 35th International Conference on Computer-Aided Design (ICCAD)*. Association for Computing Machinery, New York, NY, USA, 1–6.

[19] Yuchen Hao, Zhenman Fang, Glenn Reinman, and Jason Cong. 2017. Supporting address translation for accelerator-centric architectures. In *Proceedings of the 23rd IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, Piscataway, NJ, USA, 37–48.

[20] Troels Henriksen. 2021. Bounds Checking on GPU. *International Journal of Parallel Programming* 49, 6 (2021), 761–775.

[21] Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, Ryan Piersma, and Simha Sethumadhavan. 2021. No-FAT: Architectural Support for Low Overhead Memory Safety Checks. In *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE Press, Piscataway, NJ, USA, 916–929.

[22] Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, and Simha Sethumadhavan. 2021. ZeRØ: Zero-Overhead Resilient Operation Under Pointer Integrity Attacks. In *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Piscataway, NJ, USA, 999–1012.

[23] Intel. 2014. OpenCL 2.0 Shared Virtual Memory Overview. https://software.intel.com/content/www/us/en/develop/articles/opencl-20-shared-virtual-memory-overview.html.

[24] Intel. 2015. Introduction to Resource Binding in Microsoft DirectX* 12. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-resource-binding-in-microsoft-directx-12.html.

[25] Intel. 2016. In-Depth Discussion of Intel® Processor Graphics. https://software.intel.com/content/www/us/en/develop/blogs/micro49-tutorial-on-intel-processor-graphics-microarchitecture-and-isa.html.

[26] Intel. 2017. Intel® Iris® Plus Graphics and UHD Graphics Open Source Programmer's Reference Manual. Volume 7: 3D-Media-GPGPU. https://01.org/sites/default/files/documentation/intel-gfx-prm-osrc-kbl-vol07-3d_media_gpgpu.pdf.

[27] Intel. 2020. Intel® Iris® Plus Graphics and UHD Graphics Open Source Programmer's Reference Manual. Volume 2a - Command Reference: Instructions (Command Opcodes). https://01.org/sites/default/files/documentation/intel-gfx-prm-osrc-icllp-vol02a-commandreference-instructions_2.pdf.

[28] Gurunath Kadam, Danfeng Zhang, and Adwait Jog. 2018. Rcoal: mitigating gpu timing attack via subwarp-based randomized coalescing techniques. In *Proceedings of the 24th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, Piscataway, NJ, USA, 156–167.

[29] Gurunath Kadam, Danfeng Zhang, and Adwait Jog. 2020. Bcoal: Bucketing-based memory coalescing for efficient and secure gpus. In *Proceedings of the 26th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, Piscataway, NJ, USA, 570–581.

[30] Melanie Kambadur, Sunpyo Hong, Juan Cabral, Harish Patil, Chi-Keung Luk, Sohaib Sajid, and Martha A. Kim. 2015. Fast Computational GPU Design with GT-Pin. In *Proceedings of the 2015 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, USA, 76–86.

[31] Khronos Group. 2014. WebCL Validator. https://github.com/KhronosGroup/webcl-validator.

[32] Khronos Group. 2015. The OpenCL Specification. https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf.

[33] Hyesoon Kim, Jaekyu Lee, Nagesh B. Lakshminarayana, Jaewoong Sim, Jieun Lim, Tri Pho, Hyojong Kim, and Ramyad Hadidi. 2012. MacSim: A CPU-GPU Heterogeneous Simulation Framework User Guide. https://github.com/gthparch/macsim.

[34] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. 2020. Batch-Aware Unified Memory Management in GPUs for Irregular Workloads. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, New York, NY, USA, 1357–1370.

[35] Yonghae Kim, Jaekyu Lee, and Hyesoon Kim. 2020. Hardware-based Always-on Heap Memory Safety. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, Piscataway, NJ, USA, 1153–1166.

[36] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO): Feedback-Directed and Runtime Optimization*. IEEE Computer Society, USA, 75–86.

[37] Jaekyu Lee, Dong Hyuk Woo, Hyesoon Kim, and Mani Azimi. 2015. GREEN cache: Exploiting the disciplined memory model of openCL on GPUs. *IEEE Transactions on Computers (TC)* 64, 11 (2015), 3167–3180.

[38] Michael LeMay, Joydeep Rakshit, Sergej Deutsch, David M. Durham, Santosh Ghosh, Anant Nori, Jayesh Gaur, Andrew Weiler, Salmin Sultana, Karanvir Grewal, and Sreenivas Subramoney. 2021. Cryptographic Capability Computing. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Association for Computing Machinery, New York, NY, USA, 253–267.

[39] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. 2019. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *Proceedings of the 28th USENIX Security Symposium (Security)*. USENIX Association, USA, 1257–1272.

[40] Tao Liao, Yongjie Zhang, Peter M Kekenes-Huskey, Yuhui Cheng, Anushka Michailova, Andrew D McCulloch, Michael Holst, and J Andrew McCammon. 2013. Multi-core CPU or GPU-accelerated multiscale modeling for biomolecular complexes. *Computational and Mathematical Biophysics* 1 (2013), 164–179.

[41] Linux kernel development community. 2020. Kernel module signing facility. https://www.kernel.org/doc/html/v4.15/admin-guide/module-signing.html#:~:text=module%20signing%20facility-,Overview,signed%20with%20an%20invalid%20key.

[42] Chao Luo, Yunsi Fei, Pei Luo, Saoni Mukherjee, and David Kaeli. 2015. Side-channel power analysis of a GPU AES implementation. In *Proceedings of the 33rd IEEE International Conference on Computer Design (ICCD)*. IEEE, Piscataway, NJ, USA, 281–288.

[43] A. Markettos, Colin Rothwell, Brett Gutstein, Allison Pearce, Peter Neumann, Simon Moore, and Robert Watson. 2019. Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals. In *Proceedings of 2019 Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, USA, 1–15.

[44] John Michalakes and Manish Vachharajani. 2008. GPU acceleration of numerical weather prediction. *Parallel Processing Letters* 18, 04 (2008), 531–548.

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

[46] Matt Miller. 2019. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf.

[47] Sparsh Mittal, SB Abhinaya, Manish Reddy, and Irfan Ali. 2018. A survey of techniques for improving security of gpus. *Journal of Hardware and Systems Security* 2, 3 (2018), 266–285.

[48] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety. In *Proceedings of the 39st Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, USA, 189–200.

[49] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Association for Computing Machinery, New York, NY, USA, Article 69, 12 pages.

[50] Nvidia. 2013. Unified Memory in CUDA 6. https://developer.nvidia.com/blog/unified-memory-in-cuda-6/.

[51] Nvidia. 2014-2021. NVLink and NVSwitch. https://www.nvidia.com/en-us/data-center/nvlink.

[52] Nvidia. 2016. Nvidia Tesla P100. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[53] Nvidia. 2016. Nvidia Tesla V100. http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[54] Nvidia. 2020. CUDA-MEMCHECK User Manual. https://docs.nvidia.com/cuda/pdf/CUDA_Memcheck.pdf.

[55] Nvidia. 2020. Nvidia A100 Tensor Core GPU Architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

[56] Nvidia. 2020. Parallel Thread Execution ISA. https://docs.nvidia.com/cuda/pdf/ptx_isa_7.1.pdf.

[57] Nvidia. 2021. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#dynamic-global-memory-allocation-and-operations.

[58] Nvidia. 2021. NVIDIA GRACE CPU. https://www.nvidia.com/en-us/data-center/grace-cpu.

[59] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2018. Intel MPX Explained: A Cross-Layer Analysis of the Intel MPX System Stack. *Proceedings of the ACM Measurement and Analysis of Computing Systems (POMACS)* 2, 2, Article 28 (June 2018), 30 pages.

[60] Oracle. 2015. Hardware-assisted checking using Silicon Secured Memory (SSM). https://docs.oracle.com/cd/E37069_01/html/E37085/gphwb.html

[61] Sang-Ok Park, Ohmin Kwon, Yonggon Kim, Sang Kil Cha, and Hyunsoo Yoon. 2021. Mind control attack: Undermining deep learning with GPU memory exploitation. *Computers & Security* 102 (2021), 102115.

[62] Can Peng, Chenlin Huang, Daokun Hu, Di Bang, Jianhua Sun, Hao Chen, and Xionghu Zhong. 2019. Address Randomization for Dynamic Memory Allocators on the GPU. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, Piscataway, NJ, USA, 570–577.

[63] James Price and Simon McIntosh-Smith. 2015. Oclgrind: An Extensible OpenCL Device Simulator. In *Proceedings of the 3rd International Workshop on OpenCL (IWOCL)*. Association for Computing Machinery, New York, NY, USA, Article 12, 7 pages.

[64] Hiroshi Sasaki, Miguel A. Arroyo, M. Tarek Ibn Ziad, Koustubha Bhat, Kanad Sinha, and Simha Sethumadhavan. 2019. Practical Byte-Granular Memory Blacklisting Using Califorms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Association for Computing Machinery, New York, NY, USA, 558–571.

[65] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86). In *Proceedings of the 14th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Association for Computing Machinery, New York, NY, USA, 552–561.

[66] Seunghee Shin, Guilherme Cox, Mark Oskin, Gabriel H. Loh, Yan Solihin, Abhishek Bhattacharjee, and Arkaprava Basu. 2018. Scheduling Page Table Walks for Irregular GPU Applications. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Piscataway, NJ, USA, 180–192.

[67] Kanad Sinha and Simha Sethumadhavan. 2018. Practical Memory Safety with REST. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE Press, Piscataway, NJ, USA, 600–611.

[68] James E. Stine, Ivan Castellanos, Michael Wood, Jeff Henson, Fred Love, W. Rhett Davis, Paul D. Franzon, Michael Bucher, Sunil Basavarajaiah, Julie Oh, and Ravi Jenkal. 2007. FreePDK: An open-source variation-aware design kit. In *Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education (MSE’07)*. IEEE, Piscataway, NJ, USA, 173–174.

[69] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen mei W. Hwu. 2012. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. Technical Report. IMPACT, UIUC. http://impact.crhc.illinois.edu/parboil/parboil.aspx

[70] Synopsys. 2020. DC Ultra. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html.

[71] The Khronos®Vulkan Working Group. 2020. Vulkan®1.2.160 - A Specification (with all registered Vulkan extensions). https://www.khronos.org/registry/vulkan/specs/1.2-extensions/html/vkspec.html#shader-binding-table.

[72] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. 2019. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Association for Computing Machinery, New York, NY, USA, 372–383.

[73] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2016. Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *Proceedings of the 22nd IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, Piscataway, NJ, USA, 358–369.

[74] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert M. Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel W. Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. 2019. CHERI concentrate: Practical compressed capabilities. *IEEE Transactions on Computers (TC)* 68, 10 (2019), 1455–1469.

[75] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA)*. IEEE Press, Piscataway, NJ, USA, 457–468.

[76] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. 2019. CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Association for Computing Machinery, New York, NY, USA, 545–557.

[77] Qiumin Xu, Hyeran Jeon, Keunsoo Kim, Won Woo Ro, and Murali Annavaram. 2016. Warped-slicer: efficient intra-sm slicing through dynamic resource partitioning for GPU multiprogramming. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Piscataway, NJ, USA, 230–242.

[78] Shengjie Xu, Wei Huang, and D. Lie. 2021. In-fat pointer: hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, New York, NY, USA, 224–240.

[79] Zhiting Zhu, Sangman Kim, Yuri Rozhanski, Yige Hu, Emmett Witchel, and Mark Silberstein. 2017. Understanding The Security of Discrete GPUs. In *Proceedings of the 10th Workshop on General Purpose GPUs (GPGPU-10)*. Association for Computing Machinery, New York, NY, USA, 1–11.