GPUArmor: A Hardware-Software Co-design for Efficient and Scalable Memory Safety on GPUs

MOHAMED TAREK IBN ZIAD, NVIDIA, USA
SANA DAMANI, NVIDIA, USA
MARK STEPHENSON, NVIDIA, USA
STEPHEN W. KECKLER, NVIDIA, USA
AAMER JALEEL, NVIDIA, USA

Memory safety errors continue to pose a significant threat to current computing systems, and graphics processing units (GPUs) are no exception. A prominent class of memory safety algorithms is *allocation-based* solutions. The key idea is to maintain each allocation's metadata (base address and size) in a disjoint table and retrieve it at runtime to verify memory accesses. While several previous solutions have adopted allocation-based algorithms (e.g., cuCatch [39] and GPUShield [18]), they typically suffer from high memory overheads or scalability problems. In this work, we examine the key characteristics of real-world GPU workloads and observe several differences between GPU and CPU applications regarding memory access patterns, memory footprint, number of live allocations, and active allocation working set. Our observations motivate GPUArmor[1], a hardware-software co-design framework for memory safety on GPUs. We show that a simple compiler analysis combined with lightweight hardware support can help prevent spatial and temporal memory violations on modern GPU workloads with 8% average run time overheads while cuCatch incurs nearly 36% overhead. In fact, GPUArmor achieves speed-of-light performance with negligible storage requirements. This result benefits both base and bounds solutions and memory tagging techniques, which we showcase with GPUArmor-Binary, a variation of GPUArmor that does not require recompilation, and achieves 2.2% slowdowns while significantly reducing storage overheads beyond traditional memory tagging approaches.

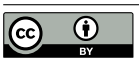
CCS Concepts: • **Computer systems organization → Heterogeneous (hybrid) systems**; *Architectures*; • **Security and privacy → Systems security**; • **Software and its engineering** → *Compilers*.

Additional Key Words and Phrases: GPU, memory safety, hardware-software co-design, bounds checking, memory tagging, spatial safety, temporal safety

ACM Reference Format:
Mohamed Tarek Ibn Ziad, Sana Damani, Mark Stephenson, Stephen W. Keckler, and Aamer Jaleel. 2026. GPUArmor: A Hardware-Software Co-design for Efficient and Scalable Memory Safety on GPUs. *ACM Trans. Arch. Code Optim.* 1, 1 (January 2026), 25 pages. https://doi.org/10.1145/3815783

[1]This article is an extension of the conference paper: Mohamed Tarek Ibn Ziad, Sana Damani, Aamer Jaleel, Stephen W. Keckler, and Mark Stephenson. 2023. cuCatch: A Debugging Tool for Efficiently Catching Memory Safety Violations in CUDA Applications. *Proc. ACM Program. Lang.*, 7, PLDI, Article 111 (June 2023), 24 pages. https://doi.org/10.1145/3591225. This article adds a comprehensive GPU workload characterization study with several novel observations, a complete hardware-software co-design (GPUArmor) with new ISA extensions, hardware structures, and compiler analysis, along with detailed implementation and evaluation addressing the performance and memory limitations of the conference paper.

Authors' Contact Information: Mohamed Tarek Ibn Ziad, NVIDIA, Westford, MA, USA, mtarek@nvidia.com; Sana Damani, NVIDIA, Santa Clara, CA, USA, sdamani@nvidia.com; Mark Stephenson, NVIDIA, Austin, TX, USA, mstephenson@nvidia.com; Stephen W. Keckler, NVIDIA, Austin, TX, USA, skeckler@nvidia.com; Aamer Jaleel, NVIDIA, Westford, MA, USA, ajaleel@nvidia.com.



ACM 1544-3973/2026/1-ART
https://doi.org/10.1145/3815783

1 Introduction

The lack of memory safety in C and C++ is a serious and long-standing issue on CPUs [35, 37, 41]. Modern GPU programming languages, such as CUDA and OpenACC, are extensions of C++ and also lack guarantees regarding the spatial and temporal validity of memory accesses. Consequently, it is unsurprising that recent research has shown that GPU software stacks are susceptible to the same vulnerabilities affecting CPUs for decades [6, 12, 20, 28]. For example, the Mind-Control attack [28] exploits a buffer overflow vulnerability on GPUs to reduce the prediction accuracy of machine learning workloads. Similarly, [12] and [30] identified several memory safety vulnerabilities in GPU applications. Thus, memory safety on GPUs is imperative.

Over the years, researchers and engineers have developed various memory safety solutions to mitigate the issues caused by unsafe programming languages. Techniques that have gained widespread adoption for protecting memory-unsafe languages on CPUs rely on *shadow memory* to store run-time metadata. For instance, the *AddressSanitizer* tool for CPUs associates a byte of shadow memory with every eight bytes of application memory [32]. At run-time, the tool verifies the validity of a pointer dereference by checking the value at the corresponding shadow memory location. AddressSanitizer uses a *direct addressing* hash table, which provides $O(1)$ access time [5]. Unfortunately, prioritizing speed causes AddressSanitizer to incur 12.5% (i.e., 1/8) memory bloat.

GPUShield [18] and cuCatch [39], two state-of-the-art GPU memory protection solutions, also use direct addressing tables to facilitate fast verification of memory operations. GPUShield's hash table is manageably sized because the universe of its keys is small, thereby limiting the scalability of the technique [18]. For example, GPUShield cannot handle five of the workloads we studied because they have more than 128 live allocations. cuCatch is scalable, but like AddressSanitizer, it suffers a 12.5% memory bloat [39]. Both schemes use a key (*e.g.*, a pointer identifier [18] or a data address [39]) to retrieve metadata (*i.e.*, the base and bounds) for the allocation to which the key maps in the hash table. The choice of direct addressing tables is presumably based on the conventional wisdom that memory operations are frequent and tend to be on the critical path, so any additional latency from a memory protection scheme directly translates to run time overhead.

This paper presents GPUArmor, a memory protection scheme that challenges the wisdom of using direct addressing tables for GPU memory protection. Initially, we planned for GPUArmor to provide straightforward hardware acceleration for cuCatch, a comprehensive pure software solution for GPU memory protection that delivers base and bounds protection for *global*, *shared*, and *local* memory spaces. We sought to implement hardware instructions for the two intrinsics that cuCatch requires: LOADMETA and MEMCHECK [39]. Notably, LOADMETA is the intrinsic function in cuCatch that fetches the metadata from the hash table for the allocation associated with a data address.

Within well-defined ABI boundaries (i.e., within a function or kernel), cuCatch finds the set of root pointers from which all other pointers within the boundary are derived, and eagerly converts them to fat pointers. Figure 1 illustrates how cuCatch functionally transforms the kernel, `GPUKernel`, to protect the pointer dereference `a[tid]` on line (8). An analysis determines that `buf1` and `buf2` are the root pointers for the kernel, and inserts LOADMETA functionality at the kernel's entry to lookup the metadata for `buf1` and `buf2` from the hash table, effectively turning them into fat pointers. The fat pointer metadata propagates through pointer arithmetic (the highlighted instructions in blocks `BB1` and `BB2`), which the MEMCHECK function in `BB3` uses to verify that `a + tid` is temporally valid and within the bounds of the root pointer from which 'a' is derived.

To reduce run time overheads of cuCatch, we introduce two new ISA-level instructions for cuCatch's intrinsics along with their microarchitectural implementation. In an effort to reduce the cuCatch metadata storage bloat, we analyzed real-world GPU workloads. We discovered unique

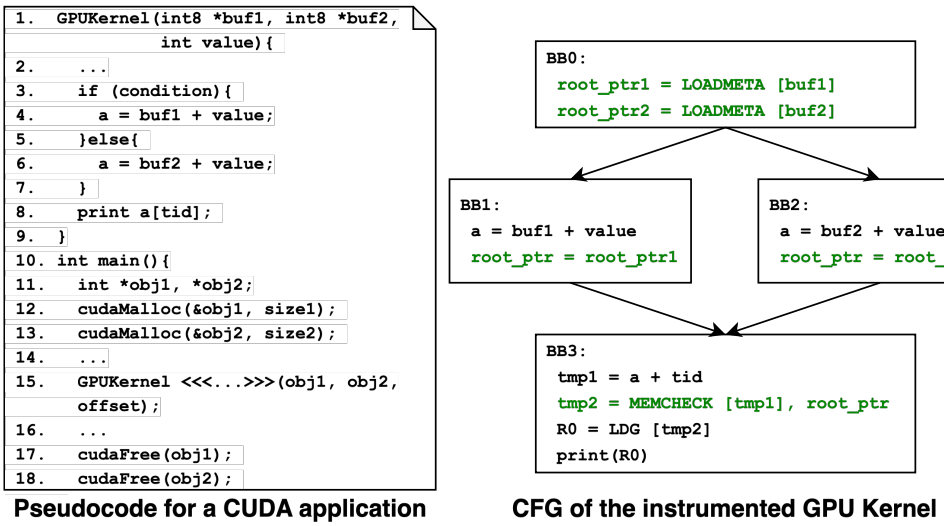


Fig. 1. GPUArmor compiler instrumentation.

properties about GPU applications in terms of their memory access patterns, memory footprint, allocation working set size, and the number of live allocations. In particular, while the size and number of live allocations tend to be large, the allocation working set is fairly small. These studies motivated a small hardware-managed structure to store recently fetched metadata, which we call the Metadata Lookaside Buffer (MLB). The MLB is extremely effective, and with it, GPUArmor can consider different points in the *time-space tradeoff* of hash tables [31] for LOADMETA's operation, paying for retrieval time primarily for compulsory misses. A hash table with $O(N)$ time retrieval cost (*i.e.*, a linked list of allocation records) incurs negligible space overhead and 13% run time overhead, while an $O(logN)$ structure limits run time overhead to under 8%, on par with a speed-of-light solution.

While we base GPUArmor's design on cuCatch, we emphasize that our insights apply to other memory protection schemes with structural similarities. For example, using a variation of GPUArmor, we apply our approach to memory tagging, attaining 2% average run time overhead with insignificant memory bloat.

In this paper we make the following contributions:

- We analyze real-world GPU workloads, uncovering new insights into memory allocation properties.
- We propose and evaluate a hardware-software co-design for scalable and efficient memory safety for GPUs.
- We show that a small cache marginalizes a GPU memory safety scheme's metadata lookup cost, freeing designers to choose slow but space-efficient structures.
- We demonstrate that our approach reduces memory requirements of widely-used memory tagging algorithms.

2 Background and Related Work

We now overview memory safety vulnerabilities, describe the GPU programming model with its different memory spaces, and summarize prior memory safety schemes and their limitations. This paper uses the terms *object* and *allocation* interchangeably to refer to a memory allocation.

2.1 Memory Safety

Applications written in C, C++, and CUDA, are vulnerable to a variety of memory safety errors because they do not validate the bounds and lifetime of memory accesses. Memory safety errors can lead to control-flow hijacking, silent data corruption, difficult-to-diagnose crashes, and security exploitation [37] even in the presence of trusted execution environments (TEEs) [3].

Spatial Memory Safety. This class of memory safety errors occurs when a pointer is used to access an allocation beyond its intended bounds (i.e., base address and size), such as buffer over- or under-flows. If the overflow target is adjacent to the victim buffer, it is called linear overflow (e.g., using a large "size" argument in a `memcpy` call-site). On the other hand, if the overflow target is non-adjacent to the victim buffer, it is referred to as non-linear overflow (e.g., using an arbitrarily large array index, `a[index]`).

Temporal Memory Safety. This class of errors occurs when an application uses a pointer to access an allocation beyond its lifetime, such as use-after-free (UAF). If the application uses a dangling pointer to access a heap allocation after it is deleted, it is referred to as immediate UAF. If the dangling pointer is used after the deleted memory is reallocated, it is called delayed UAF or use-after-realloc.

2.2 GPU Memory Spaces

While our work applies broadly to GPUs, we focus on NVIDIA GPUs and the CUDA programming model due to their popularity and importance in safety-critical applications. A CUDA application consists of host-side functions, which run on the CPU, and device-side functions (called kernels), which run on the GPU with thousands of concurrent threads. Device-side GPU kernels can store user data in different memory spaces including: `local` memory, which is thread-private and cannot be shared, `shared` memory, which is only shared among threads running on the same streaming multiprocessors (SMs), and `global` memory (the heap), which is universally shared among all threads running on the GPU. None of our workloads uses `local` memory, reflecting the reality that `local` memory allocations are rare. We therefore focus on protecting `global` and `shared` memory spaces and argue for reverting to a software-only approach for `local` memory protection [39].

Table 1. Comparison with prior memory safety works on GPUs.

	Proposal	Technology	Spatial Prot. †	Temporal Protection	Protection Scalability	Metadata Type	Storage Costs	Run time Overheads ‡
Tripwires	Compute Sanitizer [24]	Binary	◒	**None**	Yes	Disjoint	2 words per object	High
	GMOD [7, 8]	Compiler	◒	**None**	Yes	Adjacent	8B canary per object	Low
	clARMOR [10, 11]	Compiler	◒	**None**	Yes	Adjacent	8B canary per object	Moderate
Memory Tagging	LAK [43]	Hardware	◒	Probabilistic	Yes	Co-joined	4 bits per 16B region	Moderate
	IMT [36]	Hardware	◒	Probabilistic	Yes	Co-joined	Embedded within ECC	None
	GPUArmor-Binary	Hardware	◒	Probabilistic	Yes	Co-joined	2 words per object	Low
Base and Bounds	cuCatch [39]	Compiler	●	Probabilistic	Yes	Co-joined	32 bits per 32B region	High
	GPUShield [18]	Compiler+HW	●	**None**	**No**	Co-joined	2 words per object	Low
	LMI [17]	Compiler+HW	●	Limited	Yes	Co-joined	2^n padding bytes per object	Low
	GPUArmor	Compiler+HW	●	Probabilistic	Yes	Co-joined	2 words per object	Moderate

† Spatial Safety Protection: ● - Complete (Adjacent and non-adjacent OOBs); ◒ - Adjacent OOBs only; ○ - No detection.
‡ Run time overheads based on reported average in original paper: High is > 10%, Low is ≤ 2%, Moderate is 2-10%

2.3 Memory Safety on GPUs

Existing GPU-based memory safety solutions can be grouped into three categories, as shown in Table 1.

Tripwires. This category of GPU solutions distinguishes allocated from unused memory regions by either tracking allocated memory ranges [24] or surrounding them with canaries [7, 8, 10, 11].

Existing tripwire-based solutions suffer from high run time overheads (as they are implemented in software) and are ineffective against non-adjacent out-of-bound (OOB) errors.

Memory Tagging. These schemes rely on tag mismatches to probabilistically detect the errors missed by tripwires. A recent implementation of memory tagging on GPUs is LAK [43], which uses a physical memory carve-out for tag storage (8-bit tags per 16B granularity). Carve-out based memory tagging implementations are storage inefficient. A 32KB object consumes a 2KB metadata storage for maintaining the same 8-bit tag value. Another GPU-based memory tagging proposal is IMT [36]. While IMT completely avoids the storage and run time overheads of performing the memory safety checks by embedding the memory tag bits in ECC, IMT has two main limitations. First, IMT fails to detect tag mismatches in the presence of data errors. While the probability of having a tag mismatch and a random error affecting the same memory word is rare, adversaries can leverage RowHammer [16] to induce errors for bypassing the tag check even in the presence of ECC [4]. Second, IMT is constrained by the underlying ECC properties such as granularity (e.g., 2B of ECC per 32B data access or 6.25% redundancy), which either results in memory fragmentation (due to padding allocations to 32B) or undetected sub-32B overflows.

Base and Bounds. The third category of GPU memory safety solutions provides the highest spatial safety coverage. One example is cuCatch, which is a software-only memory safety debugger that opportunistically creates fat pointers with base and bounds protection within ABI boundaries [39]. As a software-only solution, cuCatch suffers from non-negligible run time and memory overheads. Another example is GPUShield, which uses the upper pointer bits to index into a per-kernel bounds table for retrieving the bounds of the pointed-to object [18]. While GPUShield has low run time overheads and deterministic spatial safety guarantees, it has three main limitations. First, GPUShield can only protect a limited set of allocations (bounded by the number of currently unused upper pointer bits). This maximum object count (128 on 57-bit architectures) can be easily exceeded in case of (1) in-kernel `mallocs`, where each thread allocates its own object and (2) heterogeneous memory management (HMM) [13] allocations that are created on the host-side and implicitly migrated to the GPU. Second, the implicit migration poses another challenge for GPUShield, which only protects explicitly-passed allocations to device-side code via kernel arguments whereas HMM allocations can be passed to device-side code as a pointer within a `struct`, escaping GPUShield's instrumentation. Third, GPUShield creates an immutable metadata table for each kernel upon launch, which persists for the kernel's entire lifetime. Therefore, the metadata tables for active kernels do not reflect allocations or deallocations that occur during their execution, leaving applications vulnerable to false positives and temporal memory safety errors. A concurrent work, LMI [17], addresses the scalability issues of GPUShield by aligning objects to power of two addresses. However, LMI increases memory overheads due to the extra paddings bytes, still relies on upper pointer bits for spatial safety, and offers limited temporal safety protection. Thus, *there is a need for an efficient base and bounds scheme that can scale to an arbitrary number of allocations and handle data transfers without sacrificing temporal safety.*

3 GPU Workload Characterization

In order to better appreciate the GPUArmor system design, we first present several metrics of GPU workloads that pertain to memory safety. This section makes several observations regarding the metrics that motivate our eventual solution.

3.1 Workloads

We consider 28 CUDA kernels that industry product groups use to guide architecture design. The kernels come from real-world GPU applications, which we simulate using realistic inputs. Our workloads cover different workload segments, from scientific computing (`namd`, `amber18`,

Table 2. Workload information including instruction counts, number of allocations and memory footprint.

Workload Name	#Instr. (M)	#Alloc-ations	Footprint (MB)	Workload Name	#Instr. (M)	#Alloc-ations	Footprint (MB)
amber18_1	283	84	62	lammps_4	1723	49	1179
amber18_2	60	1081	3374	lammps_5	150	125	927
amber18_3	64	1081	3374	lammps_6	1180	128	1123
amber18_4	50	92	272	ldpc5Gdecode	56	4	2
AMG_1	112	2	473	namd_1	52	80	2048
AMG_2	77	1	236	namd_2	44	1184	291
AMG_3	1642	5	957	Optix1	71	14	1060
AMG_4	95	5	1181	Optix2	94	14	1060
fun3d	175	6	634	Optix3	4469	14	1060
Laghos_1	621	6	191	relion_1	821	22	1288
Laghos_2	406	6	191	RTM_1	1024	7	1675
lammps_1	659	43	530	RTM_2	340	9	789
lammps_2	4905	45	530	RTM_3	1632	13	1858
lammps_3	3166	49	1179	RTM_4	2347	12	1420

AMG, FUN3D, Laghos, lammps, Relion, and RTM), to commercial (5G decoding), and visualization (Optix [29]). Section 9 summarizes the profiling and simulation methodology. Table 2 shows each workload's total number of dynamic instructions, number of allocations, and memory footprint.

3.2 Observations

Instruction behavior. Figure 2 shows the dynamic distribution of memory operations: global, shared, local, constant, texture, and generic with respect to all other operations. On average, global memory accesses account for 6% of executed instructions. Compared to typical CPU workloads, this percentage is small [34]. This observation is primarily due to the significant difference in registers available to GPU threads versus a typical CPU, where a GPU compiler can typically avoid spilling and can promote all local variables to registers. In addition, there's an inherent overhead in managing thread-level parallelism.

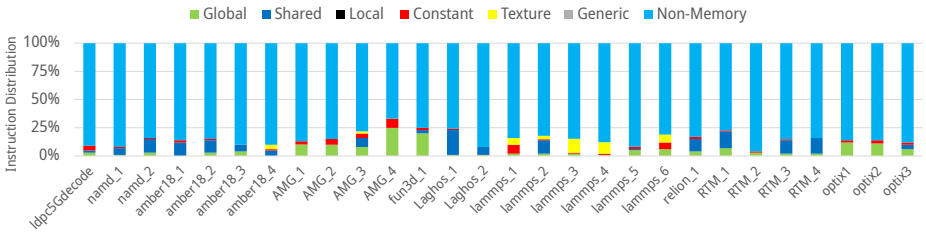


Fig. 2. Dynamic instruction distribution showing global, shared, local, constant, texture, generic, and non-memory instructions.

The CUDA Programming Guide recommends using shared memory for frequently touched data where possible, which offers lower latency accesses and a higher bandwidth connection than global memory does [25]. We see that the developers of our workloads offload, on average, half of all memory operations to shared memory. Some workloads, such as Laghos make significant use of shared memory, but on average shared memory accounts for roughly 6% of all dynamically executed instructions.

Finally, we note that constant, texture, and local memory usage is rare in modern workloads. For example, constant and texture memory accesses account for 2% and 1.5% of all memory operations

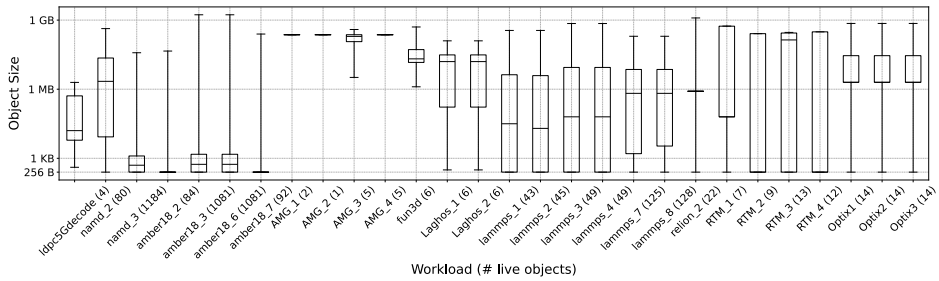


Fig. 3. Object size distribution for different workloads.

respectively. Further, none of our workloads has generic or local memory operations. Recall that local memory is used for thread-private data, such as for register spills and local variables, including stack-allocated arrays. Spilling is rare on GPUs because CUDA compilers aggressively inline functions (avoiding the spill/fill regions ABI calling conventions incur) and the compiler also takes advantage of the massive register files in GPUs. Local array allocations are of limited usefulness because stack allocations cannot be shared with other threads and because the CUDA programming model restricts per-thread stack size (16MB maximum).

Observation 1: Global and Shared memory accesses are less frequent in GPU workloads than in CPU workloads, and stack-allocated arrays are practically unused in the GPU paradigm.

Ramification: GPU global and shared memory protection schemes may be able to tolerate significant per-memory-operation overheads without commensurately affecting kernel-level overheads. Stack canaries or expensive software-only checking [39] may suffice for protecting stack-allocated arrays.

Hardware-level behavior. GPUs are designed to handle highly parallel workloads, which allows them to process many threads simultaneously. When a warp stalls because a memory request misses in the cache, the architecture can switch execution to another warp, effectively hiding memory access latencies.

Observation 2: GPUs are designed to tolerate long-latency memory operations.

Ramification: GPUs can tolerate the additional latency that memory checking incurs.

GPUs also have limited memory capacity, and Unified Virtual Memory oversubscription is inefficient. NVIDIA's H100 data center part has 80GB of memory, and an A5000 part has 32GB.

Observation 3: GPUs have limited memory capacity.

Ramification: Metadata structures that scale linearly with memory footprint waste a constraining resource (e.g., cuCatch's 32-bits per 32B [39] or LAK's 8-bits per 16B [43]).

Memory and allocation behavior. Allocation sizes impact a memory protection scheme's memory overhead. For instance, if a scheme adds 16B for every allocation to store the allocation's base and bounds, but an application's average allocation size is 16B, then the scheme doubles the application memory footprint. Figure 3 reports that the median allocation size of our workloads varies between 256B and 236MB. For efficiency, CUDA's memory allocator guarantees at least 256B alignment [25], which explains our lower bound. Around 25% of our workloads have a median object size less than 4KB. Further, the maximum object size in our workloads is 1.7GB.

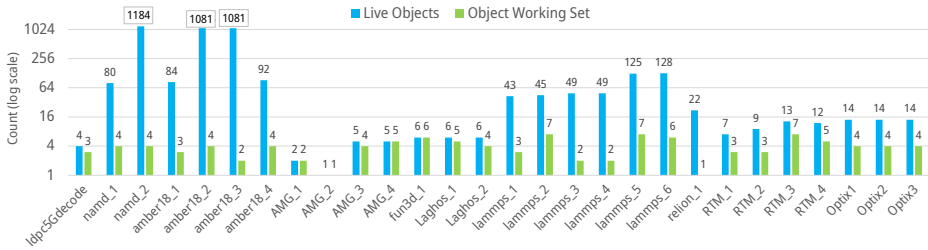


Fig. 4. Dynamic object count for different workloads.

Observation 4: GPU allocation sizes tend to be large.

Ramification: Small, constant-sized metadata per allocation contributes minor storage overhead.

The maximum number of potentially accessible allocations at any point in a kernel, which we call live allocations, determines the capacity requirements of a memory safety solution's metadata structure. Figure 4 shows that our workload set has three kernels with over one thousand simultaneously live allocations.

Observation 5: The maximum number of live allocations in GPU workloads can exceed small, fixed-size hash tables.

Ramification: A scalable GPU solution needs to effectively track the metadata of a large number of allocations. Solutions that rely on the upper address bits to index a metadata structure, such as GPUShield [18], cannot reliably account for all potentially reachable allocations for some workloads. With a 57-bit address space, only seven bits can be repurposed as an index, allowing for reliable tracking up to 128 live allocations. Even worse, if temporal safety is added to these schemes (e.g. cuCatch [39]), the available bits drop from seven to three.

On the other hand, while the total number of allocations can be large, the allocation working set tends to be small. We define the allocation working set as the maximal set of simultaneously accessed allocations at all points in a kernel. The working set size represents the number of entries a metadata cache needs to avoid capacity misses. Figure 4 shows that the allocation working set is less than eight for all of our applications.

Observation 6: Kernels have small allocation working sets.

Ramification: A small per-SM hardware structure with insignificant area cost can effectively cache allocation metadata.

The observations above motivate our GPUArmor design.

4 Threat Model

Due to its low run time and storage costs, we envision GPUArmor to be used during both: pre-deployment (for testing and verification of CUDA applications) and post-deployment (for catching bugs that escape testing) stages. To this end, we consider a threat model that is consistent with prior hardware-based memory safety work on GPUs [18, 36, 43].

Attacker Capabilities. We assume that the device-side GPU application suffers from one (or more) spatial or temporal memory safety errors that can be abused to illegally read from and write to arbitrary memory addresses. Attackers can only abuse these errors by providing inputs to the GPU application. Attackers cannot generate and execute their own device-side GPU kernels (or hijack program execution towards certain code gadgets) before bypassing GPUArmor protection first.

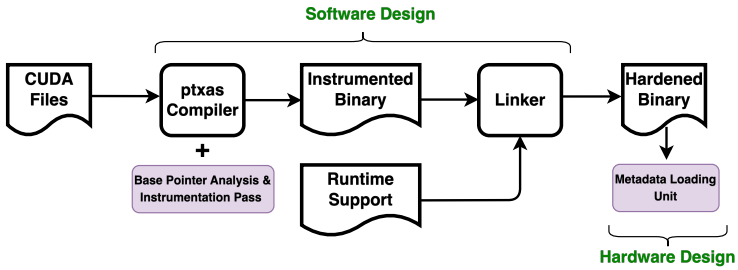


Fig. 5. GPUArmor system overview.

Hardening Assumptions. We assume default process isolation guarantees on the GPU and as such each process has its own GPU kernels and virtual address space. We consider the GPU hardware components to be reliable, and thus side- and covert-channel GPU attacks [9, 23, 33, 44] are out of scope. Finally, our current prototype focuses on protecting device-side code. Existing CPU tools [32] can be used to detect errors in host-side code.

5 GPUArmor System Overview

To address limitations of prior work, we propose GPUArmor, a hardware-software co-design for efficient and scalable memory safety on GPUs. We first describe the high-level idea of GPUArmor. We then describe its hardware and software details in Section 6 and Section 7, respectively.

5.1 Main Idea

Figure 5 provides an overview of our GPUArmor framework. Using CUDA source files as input, GPUArmor performs compile time analysis and instruction insertion (Section 7.2) for the device-side code. The resulting binary is linked against a runtime library that contains wrappers around the memory management APIs (Section 7.1). Finally, the hardened binary and runtime run on a GPU that supports the new instructions (Section 6.1) with memory safety optimizations (Section 6.2).

When an allocation is created, GPUArmor stores the memory safety metadata in a disjoint in-memory structure. To verify the memory safety of an arbitrary pointer at run time, GPUArmor retrieves the memory safety metadata from memory and performs spatial and temporal safety checks. To better understand how GPUArmor works, we consider the metadata life cycle from creation to access and finally deletion, as shown in Figure 1.

5.2 Allocation Life-cycle

Metadata Creation. At program initialization, GPUArmor reserves, but does not allocate, dedicated address space for the metadata structure. The metadata address space is read-only on the device. In this paper, we experiment with three metadata structures with very different asymptotic behaviors: a linked list, a balanced binary search tree, and a hypothetical (and unrealizable) speed-of-light (SoL) table. The linked list and binary tree store a per-allocation 32-byte node, divided into a 16-byte metadata entry and two 8-byte pointers for next and previous (or child) nodes. The SoL table is an array of 16-byte entries, each of which contains the base and bounds for an allocation.Inspired by cuCatch [39] and GPUShield [18], GPUArmor uses a pointer's value (rather than the pointer's location in memory [22, 26]) as a key to search the metadata structure. This design choice avoids storage overheads and the issue of identifying pointer copy operations (to update per-pointer metadata), which is generally intractable in unsafe languages.

When an application allocates memory (Lines 12 and 13 in Figure 1), GPUArmor stores a 16-byte metadata entry in the metadata structure. The metadata entry consists of an 8-byte *tagged* base address and an 8-byte allocation size. To detect temporal safety violations, GPUArmor tags pointers during the allocation process. That is, in GPUArmor every pointer is a tuple of a randomly generated t-bit tag and v-bit virtual address. To store the tuple in a 64-bit pointer, which is important for preserving CUDA's existing ABI, the tag size, t, depends on the number of available upper pointer bits. It ranges from seven on modern x86_64 architectures with a 57-bit linear virtual address [14] to 16 bits on traditional 48-bit architectures. The remaining v bits are used for storing the virtual base address of the allocation. In addition to tagging the pointer for an allocation, GPUArmor stores the t-bit tag in the upper bits of the corresponding metadata entry's base address.

Metadata Retrieval. During program execution, when a pointer is used to access memory (Line 8 in Figure 1), GPUArmor validates the safety of the access. To do so, GPUArmor first retrieves the corresponding allocation metadata from the metadata structure using the compiler-identified root pointer (detailed in Section 7). This step might involve multiple memory accesses depending on the metadata structure's layout. A linked list (or binary tree) metadata layout incurs n (or $\log n$) 16-byte loads to traverse the linked list (or binary search tree), where n is the position of the allocation metadata entry in the metadata structure. Each load is followed by a comparison to check if the address falls between the allocation base and size of a given node. We elaborate on how this metadata retrieval process is optimized in Section 6.2. For the SoL table, we use an oracle to find a reference to the metadata entry's location in a table, then a single 16-byte load retrieves the corresponding allocation's base, tag, and size information, achieving $O(1)$ performance.

Metadata Usage for Address Checking. Once the allocation metadata (base address, size, and tag) is retrieved, we check the memory safety of the corresponding memory access by checking whether the input address lies within the base and size of the allocation (i.e., spatial safety check) and comparing the metadata tag versus the tag from the upper bits of the input address (i.e., temporal safety check). In other words, GPUArmor effectively constructs a fat pointer that holds all the memory safety information without changing the allocation layout or restricting the number of protected allocations to the unused upper pointer bits.

Metadata Deletion. When an allocation is deleted (Lines 17 and 18 in Figure 1), GPUArmor invalidates the metadata entry associated with this allocation. This way GPUArmor captures dangling pointer accesses to this region.

6 Hardware Design

This section describes the GPUArmor instruction set extensions and microarchitecture design.

Table 3. GPUArmor ISA Extensions.

Opcode	Inputs	Outputs	Functionality
LOADMETA	Ra: 64-bit compiler-identified root pointer	Rd: 64-bit pointer to metadata	Traverses the metadata structure using the root pointer and returns the address of the corresponding allocation metadata. Populates the MLB.
MEMCHECK.G	Ra: 64-bit global address to-be-checked Rb: 64-bit pointer to metadata	Rd: 64-bit address with tag cleared	Fetches base, tag, and bounds information. Performs global spatial and temporal safety checks. Clears the upper address tag bits.
MEMCHECK.S	Ra: 32-bit shared address to-be-checked Rb: 32-bit shared allocation base address Rc: 32-bit shared allocation size	-	Performs spatial memory safety checks on shared memory address.

6.1 Instruction Set Extensions

To reduce the performance overheads of software-based memory safety, GPUArmor adds three new instructions to the GPU ISA as shown in Table 3. We do not use special instructions for populating the metadata, opting instead to use software-based routines within our runtime wrappers because populating the metadata is an uncommon operation that is executed less frequently (i.e., upon (de)allocation) than operating on the metadata.

Rd = LOADMETA [Ra]. This instruction takes as input a 64-bit compiler-identified root pointer (Ra) and returns a 64-bit pointer to the metadata location, (Rd) as output. Upon executing LOADMETA, the hardware traverses the metadata structure to find the metadata location associated with the allocation pointed-to by Ra. It then stores the allocation metadata, `mdata` (i.e., 64-bit base address with tag and 64-bit size) and its 64-bit memory location into a hardware structure dubbed the Metadata Lookaside Buffer or MLB (Section 6.2). LOADMETA does not return the 128-bit metadata entry as output for two reasons: (1) to reduce register pressure and (2) to avoid storing a stale tag which might prevent identification of use-after-free violations in long-running kernels.

To avoid accidentally fetching the `mdata` of an unrelated allocation, this instruction verifies that Ra's tag matches the `mdata.tag`. The instruction returns zero otherwise. Similarly, the instruction returns zero if the `mdata` does not exist without raising an exception to avoid false positives which may happen if a corrupted pointer is computed but never accessed. We note that shared memory base and bounds information is statically determined by the compiler and requires no LOADMETA.

Rd = MEMCHECK.G Ra, Rb. This instruction takes as input a 64-bit memory address (Ra) and 64-bit metadata address (Rb), performs bounds and tag checking, and returns an untagged 64-bit memory address (Rd) to be safely consumed by the follow-up associated memory instruction. Upon executing MEMCHECK.G, the hardware uses Rb to fetch the allocation metadata (`mdata.base`, `mdata.size` and `mdata.tag`) either from the MLB if available or from the metadata location in memory (stored in Rb). It then performs bounds check by computing the difference between Ra and `mdata.base` and comparing it to `mdata.size`. Next, it compares the upper 7 (or 16 on 48-bit systems) bits of Ra with the fetched `mdata.tag` to make sure there is no tag mismatch. An exception is raised if (1) the memory access is not within legitimate bounds or (2) there is a tag mismatch.

MEMCHECK.S Ra, Rb, Rc. This instruction takes as input a 32-bit shared memory address (Ra), a 32-bit shared memory allocation base address (Rb), and a 32-bit shared memory allocation size, and performs bounds checking. Since temporal safety is not a concern for shared memory, which cannot be freed during kernel execution, there is no tag for shared memory pointers. Thus, this instruction returns nothing. Upon executing MEMCHECK.S, the hardware performs bounds check by computing the difference between the Rb and Ra and comparing it to Rc. An exception is raised if the memory access is not within legitimate bounds.

6.2 Microarchitecture Design

When retrieving the memory safety metadata, GPUArmor can incur multiple serial memory accesses, depending on the metadata structure layout (e.g., binary tree or linked list). We implement these memory accesses using a Metadata Loading Unit (MLU) in addition to hardware logic for performing the safety checks.

Metadata Loading Unit. The MLU is responsible for retrieving the safety metadata (base address, size, and tag) of the allocation pointed-to by a given memory address. In a baseline system, the Load-Store Unit sends requests to the memory input-output (MIO) unit which coalesces memory accesses across a warp. The coalesced address is sent to the L1 cache, and upon eventual response the LSU writes data to the corresponding registers. With GPUArmor, the MIO coalesces the LOADMETA addresses as usual and sends it to the MLU which in turn communicates with the L1 cache.

Conceptually, the MLU sits between the MIO and the L1 cache acting as a pass-through for conventional load/store instructions. The MLU traverses the metadata structure to find the location of the metadata entry (`mdata.location`) associated with the coalesced memory address and returns it to the LSU which writes it to registers. The metadata traversal logic is similar to a page table walk and occurs in the virtual address space. The MLU holds the base virtual address for the metadata structure (e.g., head of the linked list) and performs the traversal as described in Section 5.2. The MEMCHECK instruction uses the `mdata.location` to fetch the safety metadata from the L1 cache (or MLB as described below).

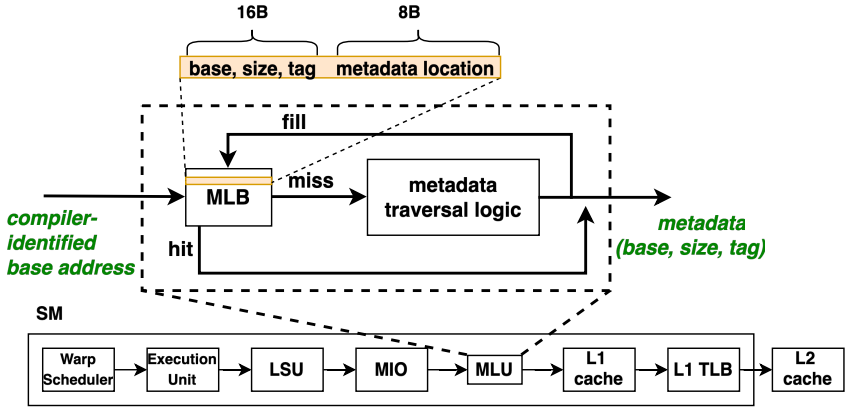


Fig. 6. The Metadata Loading Unit (MLU) block diagram.

Metadata Lookaside Buffer. To accelerate metadata retrieval, the MLU contains a Metadata Lookaside Buffer (MLB) which holds the metadata for recently accessed allocations and their locations (Figure 6). Each MLB entry consists of the 16B `mdata` (base, size, and tag) for an allocation in addition to the 8B virtual address of the metadata entry `mdata.location`. For LOADMETA instructions, an MLB lookup finds the MLB entry whose range (i.e., [base, base+size]) covers the lookup address, stored in `Ra`. On a hit, the `mdata.location` of the matching entry is returned to the LSU. On a miss, the MLU uses the metadata traversal logic to identify the `mdata` memory address and sends a request to the cache hierarchy. For MEMCHECK instructions, if an MLB lookup finds the MLB entry whose `mdata.location` field matches the `Rb` value, the data is returned to the LSU. On a miss, the MLU uses the pointer to metadata operand (`Rb`) to load the memory safety metadata from the cache hierarchy. We use a small per-SM 8-entry MLB to reduce power and meet timing requirements. MLB entries are invalidated on deallocations.

Memory Safety Check Logic. We implement the logic for performing the spatial and temporal memory safety checks described by the MEMCHECK instruction as an extension to the SM functional unit. The operands of the MEMCHECK instruction are read and operated on as summarized in Section 6.1. If a memory safety error (out-of-bounds access or tag mismatch) is detected, a device-side exception is raised, which can then be captured by the host-side application code. Users can choose to implement an exception handler that either terminates the application or reports the error and resumes execution.

Binary Compatibility. To maintain compatibility with code that is non-GPUArmor aware (e.g., accesses to local memory regions in third-party libraries), we leave the `0x0` value unused while assigning the random tags. We only perform the metadata retrieval and safety checks for memory addresses with non-zero tags. Afterwards, our GPU hardware masks off the tag bits before sending

the data request to the memory hierarchy similarly to existing CPU features, such as ARM's Top Byte Ignore (TBI), Intel's Linear Address Masking (LAM), and AMD's Upper Address Ignore (UAI).

7 Software Design

Here, we describe our changes to the software runtime and compiler in order to enable GPUArmor.

7.1 Dynamic Memory Management.

We implement the GPUArmor runtime as wrappers around CUDA's global memory management APIs (e.g., `cudaMalloc` and `cudaFree`). When a program allocates memory, we capture the size (`s`) and base address (`p`) of the allocation, and also generate a 7- (or 16)-bit tag. We store the size, base address and tag as an entry in the metadata structure, as discussed in Section 5.2. Finally, we add the tag to the upper bits of `p` and return the tagged pointer `p` to the program. When a program deletes an allocation, we leave the corresponding entry in the metadata structure as is (base address and size) and set the tag field to a special reserved tag (e.g., `0x7F`) to indicate that this memory region has been previously allocated but is currently freed. In this way, dangling pointer accesses to this area (also known as immediate use-after-free) will fail the memory safety check due to a tag mismatch between the reserved tag and the dangling pointer's upper bits.[2] Hence, GPUArmor requires no changes to the memory allocator internals nor does it pose any implementation restrictions. Since the number of allocations and frees is limited, the runtime overhead of metadata management is negligible compared to kernel execution.

Heterogeneous Memory Management (HMM). Our current prototype wraps CUDA device-memory APIs and does not cover HMM allocations [13], which are created on the host via standard `malloc` or `new` and implicitly migrated to the GPU. HMM allocations require specialized handling for two reasons. First, the number of host-side allocations visible to the GPU can be orders of magnitude larger than CUDA allocations, invalidating the small working-set assumption that underpins the MLB design. A potential remedy is to maintain separate metadata trees for CUDA and host-originated allocations, searching the latter only on a miss in the former. Second, pointer tagging for HMM allocations requires CPU-side support (e.g., ARM TBI or Intel LAM) to avoid faults when the host dereferences a tagged pointer; without such support, GPUArmor can still perform bounds checking but loses temporal safety coverage. Full HMM integration is left for future work.

7.2 Compiler Support

We implement an instrumentation pass in the GPU-compute compiler back-end to insert the new memory safety instructions: LOADMETA and MEMCHECK described in Section 6.1. The compiler inserts a MEMCHECK instruction before every global or shared memory access (e.g., the global memory read in Line 8 of Figure 1) and a LOADMETA instruction per object being accessed, instead of one per memory access. We define a *root pointer* as the initial pointer from which all other pointers to the object are derived after pointer arithmetic. We then limit LOADMETA insertion to only fetch metadata once for each root pointer. This approach has two advantages: first, we limit the number of memory accesses to fetch metadata; second, we improve the detection of out-of-bounds memory accesses when pointer arithmetic causes an overflow into an adjacent allocation.

Root Pointer Analysis. To identify root pointers, we adopt the intra-procedural compiler analysis called the *root pointer analysis* from cuCatch [39]. Our implementation of root pointer analysis performs a recursive search through reaching definitions to find all potential root pointers of

[2]If memory (de)allocations exceed a threshold, deleted metadata entries can be reclaimed, though this was not observed in the evaluated workloads.

an address being accessed. During this backward search, if we find pointer arithmetic, we keep searching until we find a definition that may be a root pointer. For each such reaching root pointer definition, we insert a corresponding LOADMETA instruction, as demonstrated in Figure 1.

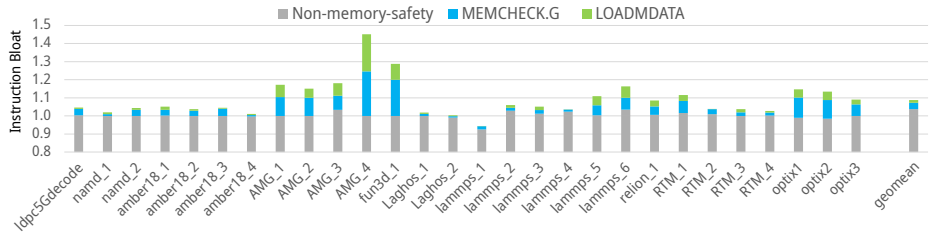


Fig. 7. Dynamic instruction bloat of our instrumentation.

For the memory access instruction, `R0 = LDG[tmp2]` in `BB3`, our analysis scans backwards through reaching definitions of `tmp2` to identify kernel parameters, `buf1` and `buf2` as the reaching root pointers of `tmp2`. Next, we insert LOADMETA instructions for both `buf1` and `buf2` in `BB0` to fetch the pointers to the metadata for `buf1` and `buf2` respectively and propagate the metadata pointers using `MOV` instructions in `BB1` and `BB2`. Finally, we insert a MEMCHECK instruction before the `LDG` instruction to perform the memory safety checks using the metadata pointer `md_a`. Figure 7 shows the total dynamic instruction bloat due to adding our new instructions. On average, our instrumentation increases the dynamic instruction count by 8.7%, comprising 1.4% for LOADMETA instructions and 3.6% for MEMCHECK instructions. Interestingly, for some workloads (e.g., `lammps_1`), the instrumented binary executes fewer instructions than the baseline. This result stems from the instrumentation altering the compiler's instruction selection and scheduling heuristics, inadvertently optimizing the baseline code path. We further analyze this sensitivity to compiler changes in Section 10.1.

Note that due to the limitations of static intra-procedural compiler analysis, the root pointer identified by our analysis may or may not be the actual base address of the object, and instead may represent some other intermediate address within the object. Crucially, the CUDA compiler aggressively inlines device functions, reducing the need for inter-procedural analysis.

Handling Different GPU Memory Spaces. While our compiler is able to instrument memory accesses targeting different memory spaces: global, shared, and local, we note that only global memory accesses require a corresponding LOADMETA instruction to fetch the object base and size. This is because the base and bounds information of local and shared memory allocations are known at compile time and are propagated accordingly to the MEMCHECK instructions using a similar compiler analysis. For generic memory accesses, we instrument them with LOADMETA and MEMCHECK instructions similar to global memory accesses. Then, we rely on the hardware to identify the target memory space of a memory instruction address at runtime. Based on the identified memory space, the hardware performs the corresponding safety check: a global memory safety check (similar to MEMCHECK.G) if the address targets global memory, or a coarse-grained safety check if it targets shared or local memory, as the exact object bounds for shared/local allocations are not available to the hardware in this case.

Finally, we note that our run-time overheads represent an upper bound that may further be reduced with compile-time optimizations proposed by prior work, including bounds check coalescing [39] and statically identifying memory-safe accesses [18].

Binary-Only Protection Mode. In the absence of compiler support (e.g., libraries with no source code availability), GPUArmor runtime and hardware support can still be used for providing reduced memory safety guarantees. We do so by using a mode called GPUArmor-Binary, in which we trigger the functionality of LOADMETA and MEMCHECK with every memory access instruction. In this design, the MLU is no longer in pass-through mode, instead each memory instruction fetches the allocation metadata, namely allocation base, size, and tag, and performs the memory safety checks. This approach increases the request rate of memory instructions by fetching the data associated with the memory instruction and also fetching the metadata. We faithfully model this and observe that high hit-rates in the MLB ensure the request rate is similar to baseline. With this metadata, we can deterministically detect spatial (and temporal) safety violations from one allocation to a non-allocated (or recently deleted) allocation and probabilistically detect violations from one allocation to other live allocations in third-party uninstrumented libraries that lack compiler support. While GPUArmor-Binary provides similar error detection guarantees as commercialized memory tagging [1, 27], GPUArmor-Binary is more memory efficient. A 32KB allocation consumes a 1KB metadata storage with ARM's MTE (4-bit tags per 16B granularity) while consuming 16B with GPUArmor, which is ≈ 64× lower memory cost. Memory savings are higher for large allocations.

Table 4. Memory Safety Error Detection Rates for GPU-based Hardware-assisted Solutions.

	GPUArmor	GPUShield [18]	**GPUArmor-Binary**	LAK [43] & IMT [36]
Adjacent OOB	100%	100%	100%	100%
Non-adj. Intra-scope OOB	100%	**99.2%**	**99.2%**	**99.2%**
Non-adj. Inter-scope OOB	99.2%	99.2%	99.2%	99.2%
Sub-object OOB	**0%**	**0%**	**0%**	**0%**
Immediate Use-after-free	100%	**0%**	100%	100%
Delayed Use-after-free	99.2%	**0%**	99.2%	99.2%

8 Security Results

Table 4 compares the memory safety error detection rates of GPUArmor against GPU-based hardware-assisted solutions. Red entries are particularly concerning. For consistency, we assume all schemes have the same number of unused upper pointer bits, $t = 7$ on 57-bit architectures.

GPUArmor Coverage. As stated in Section 7.2, GPUArmor's compiler analysis can slice through pointer arithmetic operations (which might potentially go out of bounds) and resolve the base address of the pointed-to object (or at least an intermediate non-base address that does not match the input pointer). Thus, it can achieve 100% detection rates for non-adjacent intra-scope[3] OOB errors. Otherwise, GPUArmor relies on memory tagging and hence provides the same probabilistic guarantees as GPUArmor-Binary (99.2% detection rate) for catching (1) inter-scope OOB errors that exceed the intra-procedural capabilities of our compiler analysis and (2) delayed UAF errors whose dangling pointers access the deleted memory after being assigned to a new object.[4]

To measure the compiler analysis coverage, Figure 8 shows the percentage of MEMCHECK.G instructions whose LOADMETA is using an input operand (i.e., root) that is (1) a true object base

[3]Scope here refers to the view of our intra-procedural root pointer analysis.

[4]Other temporal safety solutions, such as quarantining [32], garbage collection [2], or pointer nullification [40] can be used to deterministically detect delayed UAF errors. While integrating GPUArmor with any of the above schemes is feasible, comparing these methodologies for addressing temporal memory safety threats on GPUs is beyond our scope.

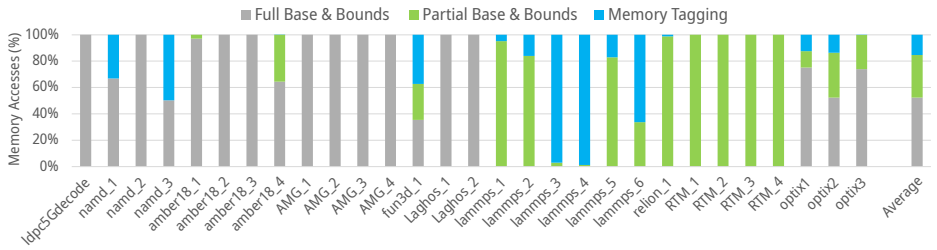


Fig. 8. Root pointer analysis coverage for GPUArmor.

address (full base-and-bounds detection), (2) a non-base address that does not match the to-be-checked memory address (partial base-and-bounds detection), or (3) a non-base address that matches the to-be-checked memory address (memory tagging detection). The difference between the first two categories is that in the first (full) category the compiler correctly identifies the true base address of the pointed-to object whereas in the second (partial) category the compiler fails to identify the object base address yet it manages to slice through pointer arithmetic instructions. For the third category, the compiler analysis uses the given memory address itself to load the metadata, which is similar to the GPUArmor-Binary functionality. On average, our analysis provides 100% full base and bounds detection rate for 55.5% of the MEMCHECK.Gs, partial detection ($99.2\% < detection < 100\%$) for 36% MEMCHECK.Gs, and memory tagging detection (99.2%) for the remaining 8.5% MEMCHECK.Gs.

GPUArmor-Binary Coverage. Like existing memory tagging schemes on GPUs [36, 43] and CPUs [1, 27], GPUArmor-Binary provides probabilistic memory safety guarantees that depend on the number of unique valid tags, with a detection rate of $100\% - \frac{100\%}{\#\text{Tags}}$. As we assume 57-bit architectures (with the `0x00` and `0x7F` tags being reserved for binary compatibility and decorating deleted memory, respectively), GPUArmor has an average detection rate of $100\% - \frac{100\%}{126} = 99.2\%$. The only exceptions are the (1) adjacent OOB errors, which can be 100% detected if different tags are assigned to adjacent objects in memory (e.g., The Android Scudo allocator [19]) and (2) immediate UAF errors, whose dangling pointers access the deleted memory before being assigned to a new object as deleted memory receives a unique tag value of `0x7F`. Unlike carve-out based memory tagging schemes [1, 43], GPUArmor-Binary can scale its probabilistic guarantees (if more upper address bits are unused) without increasing the storage costs.

Sub-object OOB Errors. As shown in Table 4, existing GPU schemes are incapable of detecting sub-object OOB errors that occur between two fields within the same struct. These errors account for around 1% of observed errors on CPUs [15]. Addressing them requires source code instrumentation to either (1) narrow the object bounds whenever an inner buffer field is accessed [22] or (2) promote the inner buffer fields within an object into standalone objects [38]. Integrating these options with GPUArmor is left for future work.

Concurrent Use-after-free. Finally, a temporal safety threat that impacts memory safety techniques that load safety metadata into registers (e.g., GPUArmor and cuCatch [39]) is the concurrent UAF, where an object is deleted by one thread while being concurrently used by another thread, leaving the `mdata.tag` bits stale in the second thread's registers. As this error is relevant for GPU code with thousands of threads/kernels executing in parallel, our prototype always fetches the `mdata.tag` from the metadata structure entry as part of executing the MEMCHECK.G instruction to avoid using a stale `mdata.tag` for temporal safety checks.

9 Experimental Methodology

Compilation. We build our workloads using the `nvcc` compiler with the default optimization level (-O3). For evaluating GPUArmor, we modify the backend `ptxas` compiler to perform the root pointer analysis and emit unique opcodes for the LOADMETA and MEMCHECK.G instructions in the compiled workloads while maintaining the correct register dependency with the corresponding memory instructions. We refer to binaries with metadata instructions as hardened binaries. For evaluating GPUArmor-Binary, we use vanilla binaries as the hardware-based memory tagging functionality does not require program modifications. In both configurations, we use the CUDA runtime library with wrappers around memory management APIs for populating the GPUArmor metadata at runtime, as described in Section 7.1.

Tracing and Simulation. We collect workload traces for the vanilla and hardened binaries on an NVIDIA GA100 GPU and simulate a GA100 using the NVIDIA Architectural Simulator (NVAS) [42]. The minimum dynamic instruction count per trace is 40 million. We implement GPUArmor, as described in Section 6 and Section 7, respectively. The LOADMETA latency depends on the number of memory accesses issued by the MLU to fetch the metadata followed by a single-cycle latency to perform the tag check. The MEMCHECK latency is bounded by a single memory load to read the metadata from its pre-identified location followed by a three-cycle latency to perform the safety check. Note that the load and safety check are all performed within the specialized MLU without involving the CUDA cores. We do not include the performance overheads of the memory management wrappers as they are typically negligible compared to the execution time of the memory management APIs themselves. For all configurations, we report performance relative to the baseline vanilla binaries.

10 Evaluation

In this section, we answer the following research questions:

(1) What are the performance overheads of GPUArmor with different metadata structure layouts?
(2) How do the hardware optimizations, namely the MLB, improve GPUArmor performance?
(3) How does GPUArmor compare to prior GPU proposals?
(4) What are the benefits and overheads of GPUArmor in the absence of recompilation?
(5) What are the area and energy costs of GPUArmor?

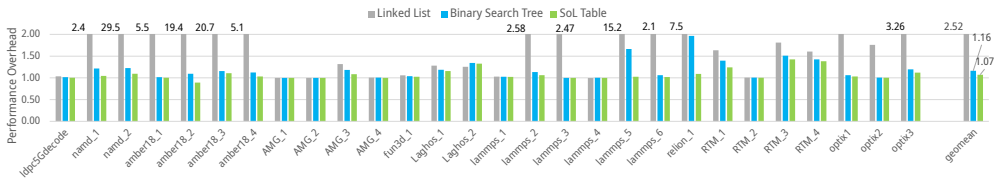


Fig. 9. Run time overheads of GPUArmor with different metadata structures.

10.1 GPUArmor Performance Results

Sensitivity to Metadata Structure. Figure 9 illustrates the performance sensitivity of GPUArmor to different metadata structures: linked list, binary search tree, and the SoL Table. We observe that workloads with high object counts (e.g., *namd*, *amber*18, *lammps*) incur high performance overheads when using a linked list. On average, a linked list data structure incurs an overhead of 2.5× (up to 29×) due to its O(N) traversal complexity in terms of memory loads. On the other hand, a more efficient data structure like a binary search tree reduces the performance overheads to an

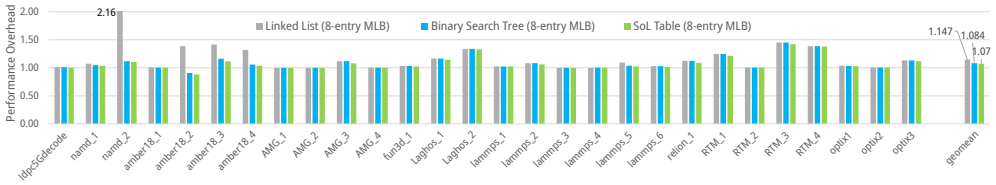


Fig. 10. Run time overheads of GPUArmor with different metadata structures (with an 8-entry MLB).

average 16% (up to 96%) because of its O(log N) traversal complexity in terms of memory loads. Large allocation count workloads like *lammps* and *relion* still observe high performance overheads due to binary tree traversal latency. Comparatively, the SoL Table incurs a 7% performance overhead due to its O(1) traversal complexity. Since the metadata is frequently accessed, we observe high metadata hit-rates (>95%) in the L1 cache, suggesting that overheads are predominantly from L1 cache access latency.

Sensitivity of Metadata Structure to MLB. Figure 10 illustrates the performance overhead of the three different data structures in the presence of an 8-entry MLB. Recall, that the MLB enables the LOADMETA instruction direct access to the metadata without traversing the metadata structure. For the majority of our workloads, we observe that an 8-entry MLB hides performance overheads due to metadata traversal. Some workloads like *namd* and *amber*18 incur non-negligible performance overheads with a linked list data structure because of compulsory accesses to the meta data structure (they have over 1000 objects and only 50M instruction run lengths). These results reveal that an adequately sized MLB is high performing and provides independence from the underlying metadata structure (simple or complex). In rare cases, we observe performance improvements over the baseline (e.g., `amber18_2` in Figure 10). Our analysis reveals that this is due to the additional metadata load instructions altering the cache locality and memory access scheduling. Specifically, the metadata accesses can prefetch data or change the eviction order in the L1 cache, coincidentally favoring the workload's memory access pattern. Similar anomalous behavior has been reported in prior GPU memory safety works [39].

Sensitivity to MLB Size. As seen above, an MLB can eliminate performance overheads if it is adequately sized to hold the metadata for the object working set (i.e., the number of objects actively being referenced). Thus, we now study sensitivity of our workloads to number of MLB entries (see Figure 11). The figure shows the minimum, maximum, and geomean performance overheads across all workloads for different MLB sizes. The results show that GPUArmor overheads saturate for an 8-entry MLB and beyond. This correlates with object working set sizes reported in Figure 4.

Sensitivity to Compiler Changes. The insertion of new instructions into the backend compiler can interfere with optimization passes, potentially leading to performance regressions. To quantify this, we use a baseline hardened binary where memory safety instructions are treated as NOPs. This represents an ideal compiler scenario with no unintended side effects. Figure 12 shows that under this model, GPUArmor's overhead drops from 8% to 2.3%, and SoL's from 7% to 1.4%. This indicates that lost compiler optimizations are the primary source of overhead, which could be recovered with further engineering.

GPUArmor Overhead Breakdown. GPUArmor mitigates performance overheads using an efficient Binary Search Tree (BST) metadata structure and a Metadata Lookaside Buffer (MLB). While the BST alone reduces overheads to 16% (vs. linked lists), the MLB further lowers this to 8%. As shown in Figure 13, approximately 6% of this overhead stems from instruction bloat due to the aforementioned compiler optimization issues, while only 2% is attributed to actual metadata access and safety checks. With an ideal compiler, GPUArmor's overheads would be approximately 2%.

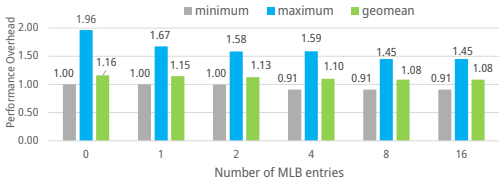


Fig. 11. Sensitivity analysis of the MLB.

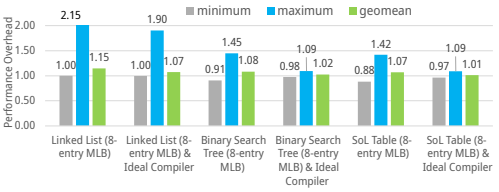


Fig. 12. Sensitivity to compiler changes.

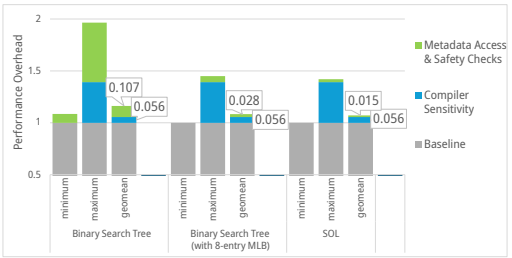


Fig. 13. GPUArmor overhead breakdown.

Shared Memory Protection Overheads. Protecting shared memory does not incur additional memory accesses, but compute logic to verify bounds. We modeled shared memory protection by increasing the latency of shared memory accesses to account for the bounds check. Across all our workloads, we observe negligible performance overhead (<0.1%) from the shared memory protection.

Stress-test Workload. To evaluate GPUArmor under worst-case conditions, we developed a microbenchmark designed to stress the metadata retrieval and caching mechanisms. The workload allocates a large number of live objects (1,024) and executes a kernel where each thread within a warp accesses a unique object chosen with a large stride to ensure sparsity in memory. This setup forces uncoalesced memory accesses across 32 distinct objects (the warp size), thereby maximizing the pressure on the MLB and metadata structure traversal. Despite these adversarial conditions, GPUArmor with a binary search tree incurs a 189% slowdown. However, with an 8-entry MLB, this overhead drops significantly to 65%, which is comparable to the outliers in our primary evaluation. An ideal SoL configuration achieves a 30% slowdown. Increasing the MLB size to 32 entries further bridges the gap, reducing the slowdown to 39%. Notably, 21% of the observed overhead is attributed to compiler-induced instruction bloat (measured using the specialized NOP baseline). These results demonstrate that GPUArmor maintains acceptable performance overheads even under artificially induced worst-case scenarios.

10.2 Comparison to Related Work on GPUs

We compare GPUArmor with the recent GPU proposals. For quantitative comparisons, we focus on software-only cuCatch [39] which has identical spatial and temporal safety guarantees but incurs 12.5% metadata storage overhead as compared to the .0005% overheads incurred by GPUArmor. We do not consider clArmor [10] and GMOD [8] since they have lower temporal and spatial safety guarantees and higher runtime overheads on average compared to GPUArmor. We also do not consider GPUShield [18] since it does not (a) provide temporal safety guarantees and (b) cannot

scale to workloads with a large number of live allocations. As clarified in Section 2.3, we consider LMI [17] to be concurrent to this work, and therefore we do not directly compare against it.

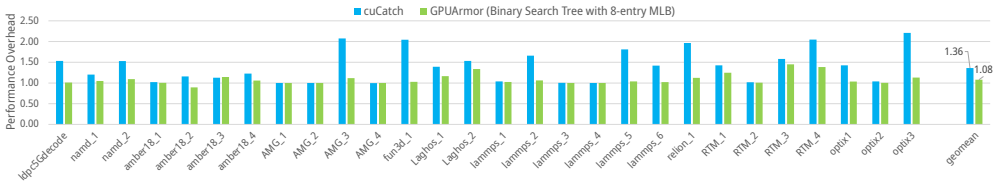


Fig. 14. Performance comparison between cuCatch [39] and GPUArmor.

To ensure a unified platform for performance comparison against cuCatch, we use the cuCatch compiler to compile, run, and collect traces of the cuCatch hardened binaries on our GPU simulator. Figure 14 illustrates that cuCatch incurs up to 2.2X performance overhead, 36% on average. On the other hand, GPUArmor with a binary search tree and 8-entry MLB incurs only a 8% performance overhead. The performance benefits over cuCatch are due to (a) reduction in instruction bloat and register pressure by avoiding executing the metadata traversal entirely in the compiler and (b) reduction in memory bandwidth and latency by leveraging the MLB. For example, *AMG*_3, GPUArmor reduces the instruction bloat by 2X and register pressure by 1.7X. The reduction in code bloat and register pressure enables GPUArmor to maintain similar SM occupancy as the baseline. This is a general trend across all workloads where cuCatch has high overheads. Thus, our hardware-software co-designed GPUArmor outperforms the software-only cuCatch with negligible performance and metadata storage overhead.

10.3 GPUArmor-Binary Results

In the absence of compiler support, GPUArmor provides memory tagging-like error detection coverage, as discussed in Section 7.2. We refer to this mode as GPUArmor-Binary in which we operate on vanilla binaries without recompilation. Figure 15 compares the run time overheads of running our workloads with GPUArmor-Binary (with an 8-entry MLB) against an ARM's MTE GPU-based implementation (dubbed Lock-and-Key or LAK [43]), which uses a physical memory carve-out for tag storage (8-bit tags per 16B memory region). GPUArmor-Binary incurs lower runtime overheads than LAK on average (2.2% versus 10%) with a maximum of 18.1% versus 42.1%. LAK incurs higher overheads because of additional memory bandwidth required to fetch the metadata (20% in our evaluations). Compared to LAK, GPUArmor-Binary requires significantly lower metadata storage overheads for its binary tree (less than 0.001% of total memory footprint) compared to LAK's tag array (with 3.25% memory storage overheads).

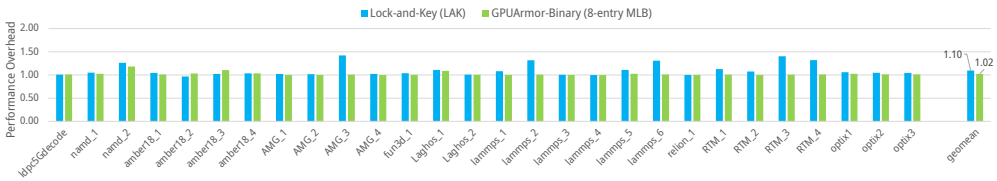


Fig. 15. Run time overheads of GPUArmor-Binary (with Binary Tree metadata structure and an 8-entry MLB) versus a memory tagging implementation on GPU (LAK [43]).

Table 5. Characterization of additional workloads from ML, Graph, Lonestar, Polybench, and Rodinia suites using default input sizes unless otherwise specified in the workload name.

Workload Name	#Instr. (M)	#Alloc-ations	Footprint (MB)	Workload Name	#Instr. (M)	#Alloc-ations	Footprint (MB)
ML				**Graph**			
ncf_bs_1M	6707	3012	33734	wb_edu	205	8	3131.12
ncf_bs_64K	497	3008	31430	orkut	420	8	5568.95
GEMM	1451	149	4945	indochina	695	8	9289.04
Lonestar				italy	25	8	691.16
bfs-atomic	1304	37	41.11	hollywood	206	8	2749.04
bfs-road	10608	35	41.11	**Rodinia**			
mst-2d	423	48	76.97	kmeans-819200	65	5	215.63
bfs-wlc	569	47	82.51	kmeans-kddcup	39	5	130.04
mst-rmat	15	48	0.63	leukocyte	1666	11	6.66
mst-road	652	48	87.33	cfd-euler3d	21649	21	12.23
sp-rand	1	49	0.01	pathfinder	550	3	1529.69
sssp-wlc	5317	48	86.59	streamcluster	8642	5	2154.34
Polybench				bfs-graph1MW_6	38	7	37.19
fdtd-2d	5435	4	48	backprop-512K	12	6	72
gramschmidt	1725	3	48	lud-1024	53	1	4
lu	1822	1	16	heartwall	4670	626	41.35
jacobi-2d-imper	34	2	8	gaussian-s256	19	3	0.5
covariance	786	3	32.01	b+tree	8	9	35.38
correlation	791	4	32.02	dwt-rgb-1024	4	9	46
convolution-3d	29	2	128	nw-16K-10	848	2	2048.25
adi	5	3	12	srad_v1	321	12	7.02
jacobi-1d-imper	47	2	0.03	gaussian-s64	0.3	3	0.03

Results Across Broader Set of Workloads. To provide a comprehensive evaluation, we extended our analysis to include a broader set of workloads from MLPerf, Graph, Lonestar, Polybench, and Rodinia suites. Unlike the production workloads used in our primary evaluation, these benchmarks represent more traditional, older GPU applications (some over a decade old) often used in prior academic studies. Table 5 summarizes their characteristics in terms of instruction counts, allocations, and memory footprints. Figures 16, 17, and 18 further illustrate their diversity in instruction distribution, object sizes, and dynamic object counts, respectively. Notably, these additional workloads largely corroborate the observations made in Section 3. For instance, consistent with Observation 5, workloads like `ncf` and `heartwall` exhibit a large number of live allocations (over 3000 and 600, respectively), yet their efficient execution with a small MLB confirms Observation 6 regarding small allocation working sets.

We evaluate GPUArmor-Binary across this additional set of workloads as it requires no recompilation. Figure 19 shows the performance overheads of GPUArmor-Binary with an 8-entry and 16-entry MLB, with the number of live allocations for each workload shown in parentheses. Our evaluations show that an 8-entry MLB incurs performance overheads of 6% on average. Workloads like *adi* and *jacobi* incur higher performance overheads due to reduction in effective L1 cache bandwidth to perform the memory safety checks. While graph workloads such as `sp` and `sssp` show sensitivity to MLB capacity, suggesting slightly larger working sets, an increased MLB size of 16 effectively captures this locality, resulting in lower performance overheads (less than 4%). Overall, the low overheads across this diverse set further validate Observations 1 and 2, demonstrating that GPU applications can tolerate the latency of our memory safety checks.

10.4 Area and Energy Overheads

We now estimate the area and energy overheads of the 16-entry MLB, where each entry stores 24 B (along with the logic for metadata traversal within the MLU) totaling roughly 384 B per SM. We use CACTI [21] to estimate the MLB area and access energy, and model comparator and priority

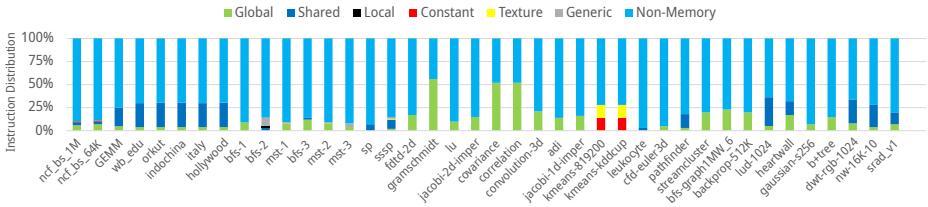


Fig. 16. Dynamic instruction distribution for additional workloads.

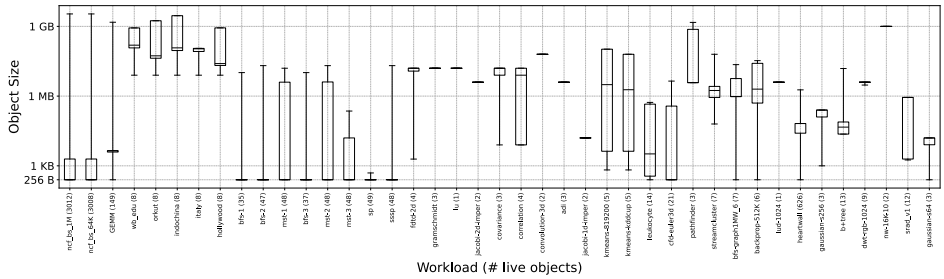


Fig. 17. Object size distribution for additional workloads.

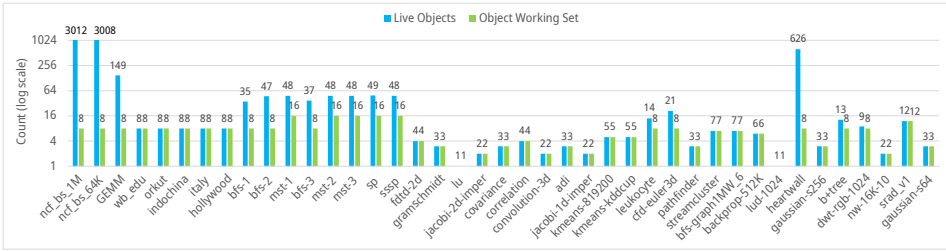


Fig. 18. Dynamic object count for additional workloads.

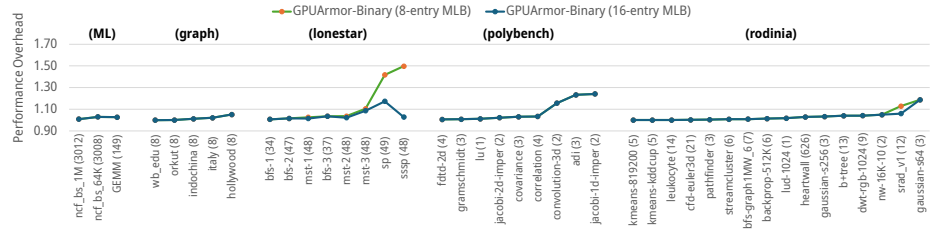


Fig. 19. Run time overheads of GPUArmor-Binary on conventional workloads with an 8- and 16-entry MLB.

logic using standard-cell energy estimates. We find that the MLB accounts for approximately 0.2% of the area and ~5% of the per-access energy of a 128 KB cache, a typical L1 cache configuration in modern GPUs. Moreover, since the MLB is consulted only for global memory instructions, which as Figure 2 shows constitute ~6% of all executed instructions, its impact on total GPU power is negligible (<0.0006%). As such, the MLB incurs minimal overheads.

11 Conclusion

Recent work shows that GPU applications are vulnerable to memory safety errors. State-of-the-art GPU solutions either trade scalability or memory bloat for performance. In this paper, we challenge traditional wisdom by proposing a scalable, memory efficient, and performant solution for GPUs called GPUArmor. GPUArmor combines a simple compiler analysis with hardware optimizations. GPUArmor's key design choices are driven by insights from real-world GPU workloads. For example, our characterization study showed that although GPU kernels have a large number of live allocations, they tend to have smaller "allocation working set". Hence, GPUArmor uses a small 8-entry MLB to cache metadata with negligible storage. Further, we observed that GPU allocation sizes tend to be large, so GPUArmor stores per-allocation metadata rather than per-address to save GPU storage (0.0005%). In summary, we show that GPUArmor achieves base and bounds error detection coverage with a simple binary tree data structure with 8% average slowdowns. If recompilation is not possible, GPUArmor achieves memory tagging error detection guarantees with the same hardware and storage cost while maintaining negligible slowdowns (2.2% versus 10%) compared to traditional lock and key implementations.

References

[1] ARM. 2019. Memory Tagging Extension: Enhancing memory safety through architecture. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/enhancing-memory-safety.

[2] Hans-Juergen Boehm and Mark Weiser. 1988. Garbage Collection in an Uncooperative Environment. *Journal of Software: Practice and Experience* 18, 9 (1988). https://doi.org/10.1002/spe.4380180902

[3] Tobias Cloosters, Michael Rodler, and Lucas Davi. 2020. TeeRex: Discovery and Exploitation of Memory Corruption Vulnerabilities in SGX Enclaves. In *SEC '20: Proceedings of the 29th USENIX Security Symposium*. 841–858. https://www.usenix.org/conference/usenixsecurity20/presentation/cloosters

[4] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. 2019. Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks. In *S&P '19: Proceedings of the 2019 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA, 55–71. https://doi.org/10.1109/SP.2019.00089

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2022. *Introduction to Algorithms, Fourth Edition*. MIT Press. https://books.google.com/books?id=HOJyzgEACAAJ

[6] Bang Di, Jianhua Sun, and Hao Chen. 2016. A study of overflow vulnerabilities on GPUs. In *NPC '16: Proceedings of the 13th International Conference on Network and Parallel Computing*. Xi'an, China, 103–115. https://doi.org/10.1007/978-3-319-47099-3_9

[7] Bang Di, Jianhua Sun, Hao Chen, and Dong Li. 2021. Efficient buffer overflow detection on GPU. *IEEE Transactions on Parallel Distributed Systems* 32, 5 (2021), 1161–1177. https://doi.org/10.1109/TPDS.2020.3042965

[8] Bang Di, Jianhua Sun, Dong Li, Hao Chen, and Zhe Quan. 2018. GMOD: a dynamic GPU memory overflow detector. In *PACT '18: Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. Limassol, Cyprus, 20:1–20:13. https://doi.org/10.1145/3243176.3243194

[9] Sankha Baran Dutta, Hoda Naghibijouybari, Arjun Gupta, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. 2023. Spy in the GPU-Box: Covert and Side Channel Attacks on Multi-GPU Systems. In *ISCA-50: Proceedings of the 50th Annual International Symposium on Computer Architecture*. Orlando, FL, USA, Article 45, 13 pages. https://doi.org/10.1145/3579371.3589080

[10] Christopher Erb, Mike Collins, and Joseph L. Greathouse. 2017. Dynamic buffer overflow detection for GPGPUs. In *CGO '17: Proceedings of the 2017 IEEE/ACM International Symposium on Code Generation and Optimization*. Austin, TX, USA, 61–73.

[11] Christopher Erb and Joseph L. Greathouse. 2018. clARMOR: A dynamic buffer overflow detector for OpenCL kernels. In *IWOCL '18: Proceedings of the International Workshop on OpenCL*. Oxford, United Kingdom, Article 15, 2 pages. https://doi.org/10.1145/3204919.3204934

[12] Yanan Guo, Zhenkai Zhang, and Jun Yang. 2024. GPU Memory Exploitation for Fun and Profit. In *SEC '24: Proceedings of the 33rd USENIX Security Symposium*. USENIX Association, Philadelphia, PA, 4033–4050. https://www.usenix.org/conference/usenixsecurity24/presentation/guo-yanan

[13] John Hubbard, Gonzalo Brito, Chirayu Garg, Nikolay Sakharnykh, and Fred Oh. 2023. Simplifying GPU Application Development with Heterogeneous Memory Management. https://developer.nvidia.com/blog/simplifying-gpu-application-development-with-heterogeneous-memory-management/.

[14] Intel Corp. 2017. 5-Level Paging and 5-Level EPT. https://software.intel.com/content/www/us/en/develop/download/5-level-paging-and-5-level-ept-white-paper.html. Document Number: 335252-002, Revision 1.1.

[15] Nicolas Joly, Saif ElSherei, and Saar Amar. 2020. Security Analysis of CHERI ISA. https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/Security%20analysis%20of%20CHERI%20ISA.pdf.

[16] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *ISCA '14: Proceeding of the 41st International Symposium on Computer Architecuture* (Minneapolis, Minnesota, USA). 361–372.

[17] Jaewon Lee, Euijun Chung, Saurabh Singh, Seonjin Na, Yonghae Kim, Jaekyu Lee, and Hyesoon Kim. 2025. Let-Me-In: (Still) Employing In-pointer Bounds metadata for Fine-grained GPU Memory Safety. In *HPCA '25: Proceedings of the 2025 IEEE International Symposium on High-Performance Computer Architecture*. Las Vegas, NV, USA, 1648–1661. https://doi.org/10.1109/HPCA61900.2025.00122

[18] Jaewon Lee, Yonghae Kim, Jiashen Cao, Euna Kim, Jaekyu Lee, and Hyesoon Kim. 2022. Securing GPU via region-based bounds checking. In *ISCA '22: Proceedings of the 49th Annual International Symposium on Computer Architecture*. New York, NY, USA, 27–41. https://doi.org/10.1145/3470496.3527420

[19] LLVM Project. 2023. Scudo Hardened Allocator. https://llvm.org/docs/ScudoHardenedAllocator.html.

[20] Andrea Miele. 2016. Buffer overflow vulnerabilities in CUDA: a preliminary analysis. *Journal of Computer Virology and Hacking Techniques* 12, 2 (2016), 113–120. https://doi.org/10.1007/s11416-015-0251-1

[21] N. Muralimanohar, A. Shafiee, and V. Srinivas. 2024. CACTI: An analytical cache and memory access time, cycle time, area, and power model. https://github.com/HewlettPackard/cacti.

[22] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for C. In *PLDI '09: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Dublin, Ireland, 245–258. https://doi.org/10.1145/1542476.1542504

[23] Ajay Nayak, Pratheek B., Vinod Ganapathy, and Arkaprava Basu. 2021. (Mis)Managed: A Novel TLB-Based Covert Channel on GPUs. In *ASIA CCS '21: Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. Virtual Event, 872–885.

[24] NVIDIA. 2024. Compute Sanitizer v2024.2.0. https://docs.nvidia.com/cuda/sanitizer-docs/pdf/ComputeSanitizer.pdf.

[25] NVIDIA. 2024. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/contents.html.

[26] 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 on Measurement and Analysis of Computing Systems* 2, 2 (2018), 28.

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

[28] 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), 102–115. https://doi.org/10.1016/j.cose.2020.102115

[29] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: A general purpose ray tracing engine. *ACM Transactions on Graphics* 29, 4, Article 66 (July 2010), 13 pages. https://doi.org/10.1145/1778765.1778803

[30] Jonas Roels, Adriaan Jacobs, and Stijn Volckaert. 2025. CUDA, Woulda, Shoulda: Returning Exploits in a SASS-y World. In *EuroSec '25: Proceedings of the 18th European Workshop on Systems Security*. Rotterdam, Netherlands, 40–48. https://doi.org/10.1145/3722041.3723099

[31] R. Sedgewick. 2003. *Algorithms in Java*. Addison-Wesley. https://books.google.com/books?id=TKpQAAAAMAAJ

[32] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In *ATC '12: Proceedings of the 2012 USENIX Annual Technical Conference*. Boston, MA, USA.

[33] Mert Side, Fan Yao, and Zhenkai Zhang. 2022. LockedDown: Exploiting Contention on Host-GPU PCIe Bus for Fun and Profit. In *Euro S&P-7: Proceedings of the 2022 IEEE 7th European Symposium on Security and Privacy*. Genoa, Italy, 270–285. https://doi.org/10.1109/EuroSP53844.2022.00025

[34] Sarabjeet Singh and Manu Awasthi. 2019. Memory Centric Characterization and Analysis of SPEC CPU2017 Suite. In *ICPE'19: Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering* (Mumbai, India). Association for Computing Machinery, New York, NY, USA, 285–292. https://doi.org/10.1145/3297663.3310311

[35] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for security. In *S&P '19: Proceedings of the IEEE Symposium on Security and Privacy*. San Francisco, CA, USA, 1275–1295. https://doi.org/10.1109/SP.2019.00010

[36] Michael B. Sullivan, Mohamed Tarek Ibn Ziad, Aamer Jaleel, and Stephen W. Keckler. 2023. Implicit Memory Tagging: No-Overhead Memory Safety Using Alias-Free Tagged ECC. In *ISCA-50: Proceedings of the 50th Annual International*

Symposium on Computer Architecture. Orlando, FL, USA.

[37] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal war in memory. In *S&P '13: Proceedings of the 2013 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA, 48–62. https://doi.org/10.1109/SP.2013.13

[38] 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 *ISCA-48: Proceedings of the 48th Annual International Symposium on Computer Architecture*. Worldwide Event, 916–929. https://doi.org/10.1109/ISCA52012.2021.00076

[39] Mohamed Tarek Ibn Ziad, Sana Damani, Aamer Jaleel, Stephen W. Keckler, and Mark Stephenson. 2023. cuCatch: A Debugging Tool for Efficiently Catching Memory Safety Violations in CUDA Applications. *Proceedings of the ACM on Programming Languages, Issue. PLDI* 7, 111 (June 2023), 24 pages. https://doi.org/10.1145/3591225

[40] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. 2017. DangSan: Scalable Use-after-free Detection. In *EuroSys '17: Proceedings of the Twelfth European Conference on Computer Systems*. Belgrade, Serbia, 405–419.

[41] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. 2012. Memory errors: The past, the present, and the future. In *Research in Attacks, Intrusions, and Defenses*. Springer, Berlin, Heidelberg, 86–106.

[42] Oreste Villa, Daniel Lustig, Zi Yan, Evgeny Bolotin, Yaosheng Fu, Niladrish Chatterjee, Nan Jiang, and David Nellans. 2021. Need for Speed: Experiences Building a Trustworthy System-Level GPU Simulator. In *HPCA '21: Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture*. Seoul, South Korea, 868–880. https://doi.org/10.1109/HPCA51647.2021.00077

[43] Chaochao Zhang and Rui Hou. 2022. LAK: A Low-Overhead Lock-and-Key Based Schema for GPU Memory Safety. In *ICCD-40: Proceedings of the 2022 IEEE 40th International Conference on Computer Design*. Olympic Valley, CA, USA, 705–713. https://doi.org/10.1109/ICCD56317.2022.00108

[44] Zhenkai Zhang, Tyler Allen, Fan Yao, Xing Gao, and Rong Ge. 2023. TunneLs for Bootlegging: Fully Reverse-Engineering GPU TLBs for Challenging Isolation Guarantees of NVIDIA MIG. In *CCS '23: Proceedings of the 2023 ACM Conference on Computer and Communications Security*. Copenhagen, Denmark, 15 pages. https://doi.org/10.1145/3576915.3616672