CuSafe: Capturing Memory Corruption on NVIDIA GPUs

Anonymous Authors

Abstract

Modern GPU applications, particularly in machine learning and scientific computing, are increasingly affected by memory corruption due to their reliance on memory-unsafe languages like C/C++. However, existing solutions either depend on hardware/software that is not available on commodity GPUs, or incur prohibitive performance overheads, rendering them impractical for real-world deployment.

We present CuSafe, a novel GPU sanitizer that is readily deployable on commodity NVIDIA GPUs. CuSafe employs a hybrid metadata scheme combining pointer tagging with in-band buffer bounds to enable accurate and efficient memory safety validation. CuSafe also introduces mechanisms such as stack epoch tracking and virtual address randomization to mitigate metadata confusion caused by temporal corruption.

Our security evaluation on 33 programs demonstrates that CuSafe achieves the best coverages of both spatial and temporal bugs among existing solutions. Moreover, our performance benchmarks on 44 programs, including large-language models like LLaMA2-7B and LLaMA3-8B, show that CuSafe incurs an average slowdown of 13% and a negligible memory overhead of 0.3%.

1 Introduction

Graphics Processing Units (GPUs) were originally designed for graphics rendering, but the adoption of GPUs in general-purpose computing has transformed them into indispensable accelerators in various domains, such as scientific computing, machine learning, and computer vision [2, 7, 8, 48]. Furthermore, the fast-evolving ecosystem around CUDA and OpenACC [26] has made it easier than ever for developers to leverage the power of GPUs. However, modern GPU programming languages like CUDA are built upon *memory-unsafe* languages such as C/C++, which are notorious for causing memory corruption bugs. Therefore, the GPU ecosystem is now confronting the same pervasive challenges that have affected CPU applications. For example, a series of memory corruption bugs [9, 27, 28, 51] has been reported in the PyTorch framework [48]. Similar cases have also been reported in other popular frameworks such as TensorFlow [2] and PaddlePaddle [7]. These bugs not only affect the robustness of GPU applications, but can also be exploited by attackers to compromise their security.

Recent studies have revealed the potential damage from these memory bugs [24, 38, 47, 49]. For example, Guo et al. [24] have shown that these memory vulnerabilities can be exploited on GPUs, which may lead to attacks such as return-oriented programming (ROP). Similarly, attacks that leverage memory corruption can undermine the accuracy of deep learning models [47]. A more recent study [49] on CUDA security finds that basic security mechanisms on GPUs like stack canary are even less secure than their CPU counterpart. Together, these studies indicate that the nascent GPU ecosystem faces the imminent threat of widespread memory corruption.

To mitigate memory corruption issues on GPUs, researchers have proposed various memory safety solutions [14, 17, 33, 34, 53]. However, these solutions often fall short of comprehensive protection, with each approach facing unique drawbacks. For instance, LMI [34] and GPUShield [33] rely on customized hardware modifications that are not available on commodity GPUs, making them infeasible for real-world deployment. cuCatch [53] faces the same problem since it needs to modify the NVIDIA's proprietary toolchains which are not available to the public. Some approaches, on the other hand, provide high compatibility with existing programs but limited detectability. For instance, GMOD [14] and clArmor [17] take a canary-based approach which efficiently detect linear buffer overflow but miss non-linear ones.

In this paper, we introduce CuSafe, the first GPU sanitizer that is deployable on commodity NVIDIA GPUs, providing effective and efficient detection of memory corruption. Unlike previous solutions that are based on customized hardware or proprietary toolchains, CuSafe is designed to work with off-the-shelf NVIDIA GPUs and the open-source LLVM toolchain. Moreover, CuSafe employs a hybrid metadata scheme that combines pointer tagging and in-band buffer

bounds. This scheme enables fast metadata retrieval using the memory broadcast mechanism on GPUs, achieving efficient and accurate detection of memory corruption. To mitigate issues like metadata confusion caused by temporal corruption (i.e., misidentifying normal data as metadata due to memory reallocation), CuSafe incorporates mechanisms like stack epoch tracking and virtual address randomization to uniquely identify metadata for local and global memory, respectively. Lastly, CuSafe employs several optimizations like loop hoisting to remove redundant checks, further improving its performance.

To systematically examine CuSafe's performance overhead, we evaluated CuSafe with a benchmark suite of 44 GPU programs. The evaluated benchmark set includes well-known GPU benchmarks such as Rodinia [10], PolyBench [37], Tango [29], and large language models (LLMs), including LLaMA2-7B [54] and LLaMA3-8B [23]. The evaluation results show that CuSafe incurs an average performance overhead of only 13% and a negligible memory overhead of 0.3%. For LLMs, CuSafe only reduces throughput by 11% on average. Beyond performance, we conducted a thorough security evaluation of CuSafe. Specifically, we developed a set of 33 GPU programs containing various spatial and temporal memory bugs. Among the evaluated memory safety solutions, CuSafe is the only system that achieves full coverage for both spatial and temporal memory bugs on the evaluation suite. Thus, we conclude that CuSafe is the first practical GPU sanitizer that can be readily deployed on commodity GPUs, offering effective and efficient detection of memory corruption.

In summary, our contributions are as follows:

- Unlike solutions based on customized hardware components or proprietary toolchains, this paper proposes, for the first time, a practical sanitizer readily available on commodity GPUs, that provides strong detection capability.
- Technically, CuSafe incorporates various mechanisms to achieve accurate detection of memory corruption on GPU, such as pointer tagging and in-band buffer bounds. Furthermore, CuSafe includes several optimizations such as check hoisting to achieve efficient violation detection.
- We implemented a prototype of CuSafe using LLVM 21 toolchains [31] and evaluated its detection capability and performance overhead across various benchmarks. The evaluation shows CuSafe delivers accurate detection with an average performance penalty of 13%. Moreover, we release the source of CuSafe at [5] to promote future research.

2 Background & Prior Work

2.1 GPU Architecture

Fig. 1 illustrates the architecture of a typical GPU. Streaming Multiprocessors (SMs) are the basic compute units of a GPU. Each SM contains multiple computing cores to execute threads in parallel. A GPU can have tens to hundreds of SMs, enabling it to execute thousands of threads simultaneously.

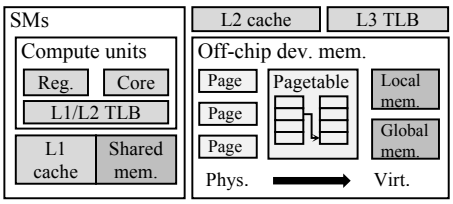

Figure 1: GPU memory architecture.

To achieve low-latency memory access, GPUs employ different types of memory to meet diverse latency and bandwidth requirements. As shown in Fig. 1, the GPU memory system comprises a hierarchy of shared, local, and global memory.

Shared memory is a small but fast on-chip memory region shared by all threads within the same SM. Unlike off-chip memory, shared memory can be accessed with low latency; it is 100× faster than off-chip memory [25]. This speed comes at the cost of limited capacity, for example, 64 KB per SM. Because of its low latency, the shared memory is often used to store data that requires frequent access to avoid accessing costly off-chip memory. Despite its name, shared memory is a scratchpad; some GPUs allow configuring the ratio between shared memory and L1 cache, but the total size is still limited.

Local memory is an area within off-chip memory that is private to each thread. It stores the thread's stack and is therefore referred to as stack memory. Similar to the stacks on CPUs, local memory is not persistent and is freed once all the threads within the same function (i.e., kernel) exit. The allocation of local memory is implicitly managed by NVIDIA's closed-source driver, meaning it cannot be directly controlled from a low-level perspective (e.g., changing its address mapping).

Global memory also resides in the same off-chip memory as local memory, but can be managed from the CPU side. Global memory is allocated through runtime APIs (e.g., `cudaMalloc`). Unlike local memory, which is freed once its threads exit, global memory is persistent and accessible by all threads across all GPU functions until it is explicitly freed (e.g., `cudaFree`). Additionally, NVIDIA partially open-sources the kernel driver [45]; its exposed APIs allow developers to customize the allocation of global memory. This capability enables CuSafe to implement an MMU-based pointer tagging scheme, which will be discussed in Sec. 4.1.

Cache hierarchy. As illustrated in Fig. 1, the GPU has a cache hierarchy to reduce the latency of off-chip memory. Each SM has a private L1 cache and a shared L2 cache. Furthermore, to reduce the latency of page table translation, each SM also has a private L1/L2 TLB and a shared L3 TLB. This hierarchical design reduces the latency of memory access from the numerous threads. However, due to its highly parallel nature, a GPU application must be carefully designed (e.g., avoiding

Table 1: Comparison between previous GPU memory safety solutions and CUSAFE.

Name	Base	Tag	Tag Size	Metadata	Spatial	Temporal	Deployability	Perf. Overhead[1]	Mem. Overhead
GPUShield [33]	HW	✘	/	Out-of-band	●	○	✘	<1%	None
IMT [52]	HW	✔	9/15	None[3]	◐	○	✘	None[2]	32 B-fragmentation
LMI [34]	HW	✔	5	None[3]	◐	◐	✘	<1%	2^n-fragmentation
GPUArmor [59]	HW	✔	7/16	Out-of-band	●	●	✘	<5%	16 B/buf
GMOD [14]	SW	✘	/	Canary	◐	○	✔	<5%	12 B/buf
clArmor [17]	SW	✘	/	Canary	◐	○	✔	<10%	4 B/buf
compute-sanitizer [42]	SW	✘	/	Out-of-band	◐	◐	✔	1,400%	Unknown
cuCatch [53]	SW	✔	4-8	Out-of-band	◐	◐	✘	<20%	160 M + 12.5%
CUSAFE	SW	✔	6*	In-band	●	●	✔	**13%**	**16.5 M + 8 B/buf**

[1] Due to the lack of released implementations, the overheads (except for compute-sanitizer and CUSAFE) are taken from published papers and are for reference only.
[2] IMT integrates its checks into ECC and introduces no additional overhead.
[3] IMT and LMI allocate memory in aligned granules and embed the corresponding tags in pointers, thereby requiring no extra metadata.
* CUSAFE might use up to 43 bits for metadata; 37 of them are for VA randomization (see Sec. 4.3) and 6 are for alignment tagging (see Sec. 4.2).

accesses to pointers that are widely dispersed across memory) to prevent TLB thrashing. In our observation, retrieving sanitizer metadata from an out-of-band shadow memory region as in cuCatch [53] and AddressSanitizer (ASAN) [50] is likely to cause this issue. To address this, CUSAFE embeds metadata in-band at the beginning of each buffer, a design choice well-suited to the GPU's cache architecture.

2.2 GPU MMU

CUSAFE relies on GPU MMU to manipulate the VA of memory objects and embeds metadata. We therefore introduce GPU MMU here.

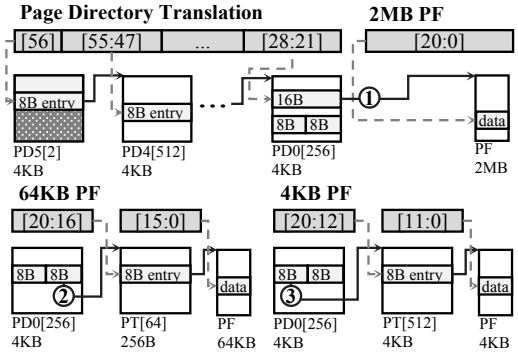

Figure 2: GPU MMU page table format.

Fig. 2 shows the page table format of Blackwell[1] GPUs [41]. As shown in the figure, a modern GPU supports multiple page sizes, including ① 2 MB, ② 64 KB and ③ 4 KB. Translation for all these page sizes follows the same traversal process, driven by bits 56 through 21 of the VA. The page directory 5 (PD5) has only two entries, which store the addresses of PD4 tables and are indexed by bit 56 of the VA. Page directories PD4 through PD1 each contains 512 entries, indexed by nine bits of the VA. PD0 differs, containing only 256 entries indexed by eight bits of the VA. Each entry in PD0 can either be viewed as one 16-byte entry or two 8-byte entries. In the former case, the 16-byte entry directly stores the address of a 2 MB page frame (① in Fig. 2). In the latter case, the lower half 8-byte entry points to a 64 KB page frame (② in Fig. 2) and the upper half 8-byte entry points to a 4 KB page frame (③ in Fig. 2).

Since the translation of VA to physical pages is solely determined by the contents and structure of these PDx tables in the MMU, CUSAFE leverages this mechanism to manipulate specific bits in a buffer's VA such that it still maps to the intended physical page. (see Sec. 4.1 for details).

[1]NVIDIA's GPUs have different page table designs across generations; we provide this example for illustration.

2.3 Memory Safety Solutions on GPUs

This section reviews memory safety solutions designed GPUs. Table 1 summarizes these approaches.

Hardware-based approaches. Many previous studies proposed new hardware solutions to implement memory access checking. One such approach is GPUShield [33], which introduces a new hardware unit that checks every memory access using an out-of-band table stored in the device memory. IMT [52] and LMI [34], on the other hand, choose to allocate memory in aligned granules and directly embed these sizes into the pointers, which removes the additional table but introduces additional fragmentation due to the alignment requirements. However, none of these approaches (GPUShield, IMT, or LMI) provides complete protection against memory corruption. GPUArmor [59], by contrast, delivers strong protection against both spatial and temporal corruption. It tags each pointer and uses the tag as an index to fetch buffer size from an out-of-band structure to check memory access. Though these hardware-based solutions offer efficient protection against memory corruption (<5% overhead), they require specialized hardware not available on commodity devices, which prevents them from being widely deployed. Consequently, we denote these approaches as undeployable in Table 1.

Software-based approaches. In addition to hardware-based

solutions, researchers also proposed software-based designs that can be directly deployed on existing hardware. For instance, GMOD [14] and clArmor [17] implemented canary-based protection on GPUs. With guard values (i.e., canaries) placed around each buffer, GMOD and clArmor effectively detect buffer overflows by checking if these guard values are altered after memory access. Though they show promising performance (<10%) and compatibility, they provide limited detection capability: any overflow that skips the guard values is missed. NVIDIA's compute-sanitizer, by contrast, tracks every allocation in shadow memory. Though this approach detects both spatial and temporal corruption, our evaluation shows that compute-sanitizer incurs an average slowdown of 15×, making it unsuitable for practical real-world deployment. NVIDIA also proposed cuCatch [53]. This solution embeds a tag into each pointer and queries a two-level structure for the valid bounds of the buffer. cuCatch demonstrates superior performance compared to compute-sanitizer (<20% overhead as shown in Table 1). However, cuCatch's tags could be polluted by overflows or even altered by attackers, as it embeds tags into pointers without checking if arithmetic overwrites them. Moreover, as cuCatch is based on proprietary toolchain, it is not released to the public, which prevents its deployment.

Impact of tag size. IMT, LMI, GPUArmor, cuCatch, and CuSafe all employ pointer tagging as a part of their mechanisms though with varying tag sizes. IMT and LMI solely use tagging to encode bounds information into pointers, where the tag size dictates the minimal allocation granularity. In such cases, a larger tag size typically leads to a lower memory overhead caused by fragmentation. For example, LMI's 5 bit tags can only encode allocation sizes from 2^8 to 2^{38} in a 2^n-aligned format whereas IMT's 7-bit or 16-bit tags enable encoding allocation sizes at a finer granularity of 32 B. GPUArmor, cuCatch, and CuSafe, on the other hand, additionally utilize pointer tags to detect temporal corruption. cuCatch uses up to 8 bits for tags, therefore, the probability of it encountering a tag collision is $\frac{1}{256}$. By contrast, CuSafe can allocate up to 37 bits (see Sec. 4.3) for detecting temporal corruption, with the remaining bits reserved for other purposes, thus drastically reducing the likelihood of tag collisions.

Summary. Table 1 compares CuSafe with current memory safety solutions for GPUs. In this table, we can see that existing GPU memory safety solutions either offer incomplete detection capability or can only be deployed with modified hardware/proprietary software. CuSafe, on the other hand, is a software-based solution that can be deployed on commercial NVIDIA GPUs. Furthermore, by adopting a hybrid design of pointer tagging and in-band metadata, CuSafe delivers effective detection for both spatial and temporal corruption.

3 Motivation

Characteristics of GPU workloads. GPU's high degree of parallelism directly leads to a need for high-throughput memory access. However, current solutions, such as cuCatch, leverage out-of-band shadow memory to store metadata, which generates a considerable memory footprint and puts significant pressure on the memory systems.

Deployment hurdles of prior work. A major problem for prior approaches on GPU is the difficulty of deployment. Currently, the only solution supported on commodity NVIDIA GPUs is the compute-sanitizer [42], which suffers from substantial performance overhead (15× as shown in Table 1). Recent studies like cuCatch and LMI [34, 53] depend on the proprietary toolchains or customized hardware, making them inaccessible to researchers and developers.

CuSafe. These observations motivate us to design a new memory safety solution for GPU – CuSafe. We implement CuSafe using off-the-shelf features (i.e., MMU) of NVIDIA GPUs so that CuSafe can be directly deployed on commodity GPUs. We also design a new hybrid metadata scheme that combines pointer-tagging and in-band metadata to accurately detect both spatial and temporal corruption. Moreover, CuSafe takes the characteristics of GPU workloads into account; its scheme leverages in-band metadata to minimize the pressure on the cache and memory system, achieving minimum performance overhead.

4 Design of CuSafe

4.1 CuSafe Overview

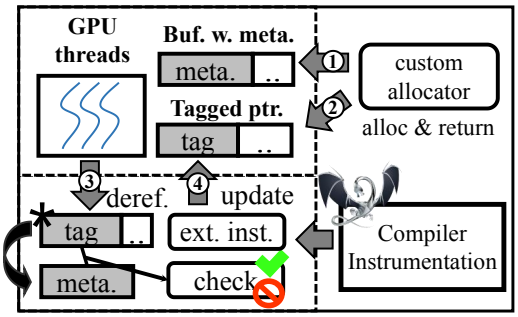

Figure 3: Overview of CuSafe.

To address the challenges faced by previous GPU sanitizers, CuSafe leverages a novel scheme to deliver accurate and efficient detection of GPU memory corruption on commodity GPUs. Essentially, CuSafe detects GPU memory corruption via compiler-inserted checks, which require no hardware modification or proprietary software, and therefore are deployable on commodity GPUs.

Workflow. As shown in Fig. 3, CuSafe consists of two components: a memory allocator (①②) and compiler-based instrumentation (③④). The memory allocator is responsible for allocating memory buffers with in-band metadata (①) and constructing tagged pointers for the GPU programs (②). To

utilize these two pieces of metadata for corruption detection, CUSAFE inserts instructions into the GPU programs via compiler-based instrumentation. Specifically, CUSAFE inserts checks before each pointer dereference to verify the validity of each memory access (③). Additionally, CUSAFE also inserts instructions to update the metadata when necessary (④). In this workflow, the key aspect is the design of CUSAFE's metadata, which determines the accuracy and efficiency of the detection. We now discuss this in detail.

CUSAFE's metadata. As described above, CUSAFE attaches metadata to two types of entities: *pointers* and *buffers*. For buffers, CUSAFE's memory allocator aggregates the buffers of the same 2^n-aligned size (e.g., 16 B) into the same VA range (e.g., `[0x1..000,0x1..fff]`), so that their VAs are automatically aligned and tagged with the same alignment tag. Additionally, CUSAFE's allocator prepends each buffer with its exact bound to enable more accurate bound checking. To retrieve this exact bound, CUSAFE utilizes the information encoded in the pointer's alignment tag to clear the lower n bits of the pointer, which automatically resets the pointer to the beginning of the buffer, where the exact bound resides.

Notably, this in-band metadata retrieval offers a key advantage for GPU applications: *when multiple GPU threads access the different parts of the same buffer (e.g., in a parallel for-loop), the GPU can broadcast the result of metadata retrieval to all threads, incurring only one extra memory transaction.* This avoids the significant performance overhead of the out-band scheme, which incurs multiple memory transactions for each thread, as they each access a different address to retrieve the metadata.

Table 2: Metadata of CUSAFE.

Entity	Properties	Metadata
Pointer	Validity (no overflow)	2^n-aligned size
	Type (global/local)	Pointer type bit
	Identity (bind ptr. to buf.)	Stack epoch (local) Randomized VA(global)
Buffer	Bound & liveness	In-band exact bounds

Aside from its friendliness to GPU applications, CUSAFE's metadata design also effectively captures both spatial and temporal properties of a memory buffer, and encodes them into the appropriate locations within the pointers and buffers. Table 2 summarizes these essential properties and how they are stored by CUSAFE. As shown in Table 2, CUSAFE embeds three pieces of metadata into the pointers: (1) the 2^n-aligned size, which is used to retrieve the exact bounds and to ensure the pointer is still valid, i.e., not tampered with by pointer arithmetics; (2) the local/global type bit, which is used to distinguish local pointers from global pointers; (3) stack epoch/randomized VA, which ensures the pointer is *not* pointing to a freed yet reallocated buffer. These three pieces of metadata comprehensively describe the properties of the pointer (i.e., validity, type, and identity). As for the buffer side, CUSAFE only embeds its exact bounds at its beginning. These bounds serve two purposes. First, they help validate whether the memory access is within the bounds of the buffer, which provides a more accurate check than the 2^n-aligned size. Second, they enable tracking the liveness property of a buffer. Upon deallocation, CUSAFE clears the exact bounds of the buffer to zero, which causes all subsequent accesses to the buffer to fail.

Pointer tagging. Given CUSAFE's heavy reliance on pointer tagging to associate metadata with pointers, We detail its implementation here. As described in Sec. 2.2, the translation of a VA is governed by a series of page directory tables (i.e., PDx). Therefore, by judiciously configuring the page table, we can manipulate specific bits within a VA while ensuring it still maps to the correct physical page. For example, to assign a specific tag value (e.g., bits [46:41][2] = 6) to all objects of 256 B, we can modify the corresponding PD3 entries to create the desired VAs.

For objects allocated in local/shared memory, their VAs are automatically managed by NVIDIA's closed-source runtime and cannot be manipulated through page table configurations. Consequently, We employ compiler instrumentation to generate "pseudo-pointers" with desired metadata bits. These pseudo-pointers are stripped of their tags before dereference to ensure correctness. A potential compatibility issue arises if a pseudo-pointer is passed to an uninstrumented external function, as that function would be unable to dereference it directly. While such scenarios are uncommon in typical CUDA applications (specifically, passing a local/shared pointer to an external function), CUSAFE mitigates this by untagging such pointers before they are passed to external functions. See Sec. 8 for a more detailed discussion.

The remainder of Sec. 4 is organized as follows. Sec. 4.2 and Sec. 4.3 detail how CUSAFE detects spatial and temporal corruption, respectively. Sec. 4.4 then discusses the optimizations adopted by CUSAFE to eliminate redundant checks and improve performance.

4.2 Spatial Corruption Detection

In this section, we discuss how CUSAFE encodes metadata to track the spatial information of the allocated buffers. We also discuss how CUSAFE prevents overflowed pointer arithmetics from corrupting the metadata.

Alignment tagging. As described in Sec. 4.1, CUSAFE tags alignment information into the pointer to retrieve metadata and validate pointer arithmetics. Illustrated in Fig. 4, CUSAFE stores the alignment information in the bits [46:41] for global pointers and bits [53:48] for local pointers. The difference in the tagged bits is primarily due to the limitation of the CUDA runtime, which fixes the bits [63:47] of global pointers to

[2] In this paper, the array notation [x:y] are inclusive on both ends.

be zero[3]. As a result, bits [63:47] cannot be used to store alignment information for global pointers. For local pointers, meanwhile, bits [46:0] are managed by the NVIDIA runtime, where we can only optionally zero the lower bits via `align n` attribute. Therefore, CUSAFE uses the bits [53:48] to store the alignment information of local pointers. Finally, to distinguish local pointers from global pointers, CUSAFE sets bit 47 of local pointers to one, leveraging the fact that this bit is fixed to zero for global pointers. We use the same length of 6 bits for both global and local pointers, encoding the buffer size in a logarithmic manner. For CUSAFE's pointers, alignment tags are assigned starting from 1, where a tag value of 1 signifies the minimal alignment unit of 16 bytes. For example, if the buffer size is 64 bytes, the alignment bits would be $log(64) - 3 = 3$. In this setup, the minimal buffer size is 16 bytes (tag = 1), and the maximum buffer size is 2^{66} bytes (tag = 63), which is more than sufficient for current GPU applications. Lastly, some additional bits (i.e., [63:54]) are reserved for pointers to uniquely identify the buffer they point to, avoiding metadata confusion caused by memory reallocation. (see Sec. 4.3).

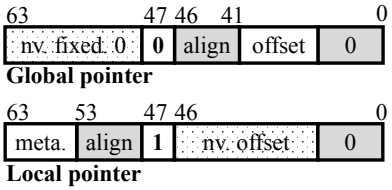

Figure 4: CUSAFE pointer tagging format.

In-band exact bounds. Some prior work like LMI directly uses the 2^n-aligned tagging for corruption detection. The main drawback of this approach is that it misses all out-of-bound (OOB) access that occur within the 2^n alignment but outside the precise bounds of the allocated buffer. Though such OOBs may seem minor, since they simply access padding data, recent studies show that feeding invalid data during ML training can cause performance degradation or even compromise the security properties of the trained model [24]. To address this issue, CUSAFE stores the exact size of the buffer in its padding area for accurate bounds checking. Fig. 5 demonstrates the allocation process of a buffer in CUSAFE. As shown in Fig. 5, when allocating a buffer, CUSAFE first prepends it with 8 bytes of metadata to store its *exact* size (e.g., for a 20-byte buffer in the figure). Then, CUSAFE aligns the VA of the buffer to the nearest 2^n boundary, to ensure that the metadata can be retrieved via the tagged pointer.

Specifically, retrieval is done by clearing the lower n bits of the pointer according to its 2^n-aligned tag. As Fig. 6 illustrates, regardless of the original pointer's specific location within the buffer, CUSAFE always uses the embedded `tag` to zero out the lower bits of the pointer (corresponding to the "offset" in

[3]VAs violating this limitation are treated as "invalid value" by CUDA runtime; this is a software constraint, as modern GPUs (e.g., RTX 5090) support 57-bit VA.

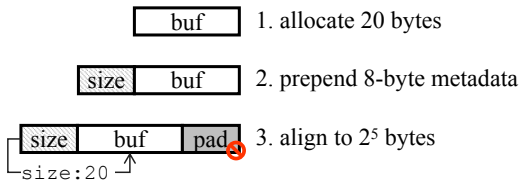

Figure 5: Metadata of exact size.

Fig. 6). Since the in-band metadata is allocated at the beginning of each buffer, this operation automatically redirects the pointer point to the in-band metadata. Then, CUSAFE uses the exact size stored in the metadata to accurately verify if the memory access is within the bounds.

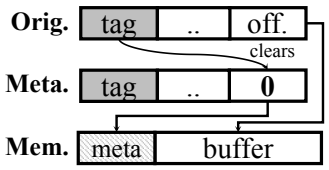

Figure 6: Retrieving exact size from pointer.

Advantage of in-band metadata. Compared to other schemes that store the metadata out-of-band (e.g., shadow memory), CUSAFE considers the high parallelism of the GPU. It leverages the GPU's *memory access broadcast* mechanism, which groups the accesses to the same address into a single memory transaction. This significantly reduces the metadata retrieval overhead. Fig. 7 highlights this advantage. As shown in the figure, when N threads access different parts of the same buffer (e.g., parallel for-loop), CUSAFE's in-band metadata scheme allows the GPU to broadcast its retrieval result to all threads, incurring only one additional memory transaction. By contrast, if the metadata is stored in shadow memory, each thread has to access the shadow memory individually with different offsets (e.g., `addr>>3+off` as in ASAN), which cannot be broadcast due to the differences in the accessed addresses, leading to N additional memory transactions. This is particularly important for GPU applications, which can easily execute thousands of threads in parallel. The extra memory transactions might overwhelm the memory bandwidth of the GPU, leading to significant performance degradation.

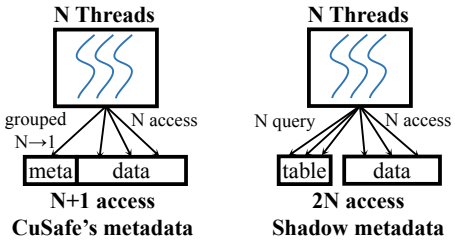

Figure 7: Comparison of metadata scheme.

Pointer arithmetic validation. Careful readers might notice

that, if the pointer goes beyond the 2^n aligned bounds of a buffer, the retrieval process described earlier will fetch the wrong metadata, leading to incorrect results. This is due to a key assumption of CuSafe: the pointer arithmetic does not modify the metadata stored in the pointers. However, this assumption often does not hold in real-world applications. A considerable number of bugs [9, 27, 28, 51] in frameworks like PyTorch [48] and TensorFlow [2] originate from the miscalculation of pointers (e.g., integer overflow) and can inadvertently corrupt the tagging bits, disabling CuSafe.

Previous solutions address this issue differently. Some, like LMI, modify the GPU's hardware to enforce pointer integrity, but this approach is not feasible on commodity GPUs. Others, such as cuCatch, offer incomplete protection by inserting padding between buffers. cuCatch's solution, however, is limited to off-by-one overflows and cannot handle more general pointer miscalculations. To address this issue, CuSafe instead introduces an additional validation step for *pointer arithmetics* to ensure that the embedded tagging information remains invariant. As depicted in Fig. 8, a pointer's "modifiable part" is defined by its tag n, which governs a 2^{n+3} address range (e.g., CuSafe allocates at a minimum size of 16 bytes where $n = 1$). Therefore, following any pointer arithmetic operation, the resulting pointer is expected to retain identical higher bits compared to the original pointer. To enforce this, CuSafe performs a bitwise `xor` between the original pointer and the resulting pointer. If the `xor` result is non-zero in the higher bits, CuSafe determines that the resulting pointer has exceeded its valid range, and invalidates it.

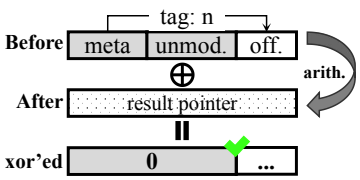

Figure 8: Validation of pointer arithmetics.

Note that we cannot directly terminate the program in such cases. This is because certain valid code constructs may produce invalid pointers without ever dereferencing them (e.g., in the final iteration of an array loop). Therefore, instead of reporting an error immediately, CuSafe sets the highest bit of the resulting pointer to one. See Sec. 8 for detailed discussion.

Summary. The detection process of spatial memory violations involves a two-step process. First, CuSafe checks whether the highest bit of the pointer is set. A set bit indicates that the pointer is invalid due to overflowed pointer arithmetics. If the pointer is valid, CuSafe then clears the lower bits of the pointer based on its 2^n-aligned tag. This retrieves the in-band exact bounds, and CuSafe checks whether the memory access is within these bounds. A memory access is only permitted to proceed if both of the checks pass. Otherwise, CuSafe raises an error and terminates the program.

4.3 Temporal Corruption Detection

This section details how CuSafe detects temporal corruption, such as use-after-free (UAF) vulnerabilities. We first discuss how CuSafe invalidates a buffer's metadata upon its deallocation, and then introduce the countermeasures we employ to prevent metadata confusion due to memory reallocation.

Metadata invalidation. Instead of deploying a new mechanism to detect temporal corruption, CuSafe invalidates a buffer's spatial metadata upon its deallocation to prevent temporal corruption like UAF. As shown in Fig. 9, CuSafe achieves this by setting the in-band exact bounds of a freed buffer to zero. Consequently, any subsequent memory access to the deallocated buffer will be captured by CuSafe since its valid size is now zero. Note that CuSafe implements invalidation differently for global and local buffers. For global buffers allocated with `cudaMalloc` and freed by `cudaFree`, CuSafe instruments the `cudaFree` function to clear the in-band metadata. It also checks the metadata before deallocation to prevent double-free errors. Local buffers, however, have *no* explicit deallocation sites. Instead, CuSafe invalidates them upon function exits.

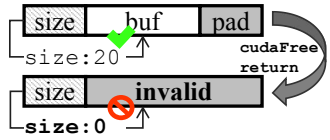

Figure 9: Metadata invalidation.

While the invalidation mechanism described above prevents most temporal corruption, it is vulnerable to metadata confusion. For example, Fig. 10 illustrates how a local buffer's metadata can be confused with subsequent local variable. As the figure shows, a pointer, `ptr`, initially points to a valid buffer. Upon function exit, CuSafe clears `ptr`'s metadata to zero, which should prevent its subsequent use. However, because stack memory is constantly reused, `ptr`'s metadata can be overwritten by the local variable `var` of another function. This causes CuSafe to use incorrect metadata to validate `ptr`, leading to incorrect results. A similar issue occurs with global buffers due to VA reuse, where data can be misidentified as metadata. To address this issue, we design *stack epoch tracking* for local memory and *VA randomization* for global memory.

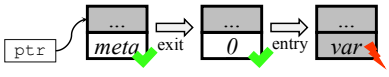

Figure 10: Metadata confusion.

Stack epoch tracking. To prevent metadata confusion for local buffers, CuSafe embeds a *stack epoch* into each pointer using the technique described in Sec. 2.2. Each epoch consists of two parts: stack depth (5-bit) and generation (5-bit). The

stack depth represents the stack depth of the current thread, while the generation represents the number of function calls at current stack depth. We additionally allocate a global buffer to keep track of the current epoch of each thread. CuSafe permits a local pointer to be dereferenced if ①: its stack depth is less than or equal to the current thread's stack depth (i.e., the pointer is from an "older" stack frame), and ② its generation equals that of its stack depth (i.e., this "older" stack frame is still alive, not replaced by a "new generation")

Fig. 11 demonstrates how CuSafe uses stack epoch to validate local pointers upon dereference. In the code snippet of Fig. 11, a CuSafe-instrumented function begins by obtaining and updating the depth `_d` and the generation `_g` of the current thread (line 2) and assigning these values (`f_d` and `f_g`) to each local buffer (line 4). Line 5 shows that `func` erroneously holds a dangling pointer, `ptr`, returned by the function `local`. However, before `ptr` is dereferenced, CuSafe finds that `ptr`'s stack depth `ptr_d` is greater than the current functions's stack depth `f_d` (line 6-7). This indicates that `ptr` has outlived its allocation function, `local`. CuSafe then raises an error and prevents the dereference.

Though the stack depth and generation values wrap around after reaching 32, this is considered an acceptable limitation. For stack depth, our experiments show that, even for a basic CUDA function, the maximum stack depth it can reach is 17, which is far less than 32.[4] As for generation, it indeed introduces false negatives under a highly specific scenario: an obsolete pointer is dereferenced after *exactly* 32 function calls at the same stack depth. This scenario is considered rare and is highly unlikely to occur in practical applications.

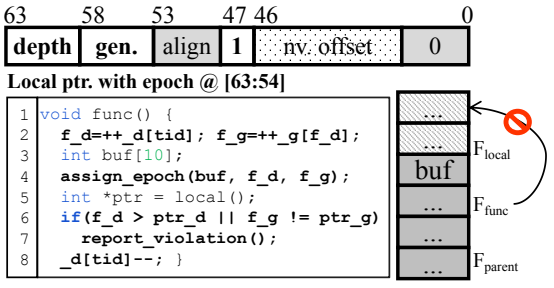

Figure 11: Stack epoch tracking.

Stack epochs for parallel threads. Due to the parallel nature of GPU applications, we create global arrays to store stack epochs, with each thread assigned one unique element. Specifically, we create two global arrays (i.e., depth and generation) with sizes equal to the number of concurrent threads on the GPU, calculated as threads/SM × #SMs. For existing NVIDIA GPUs, the maximum value of threads/SM is 2048, and the maximum number of SMs is 132 [40]. We therefore conservatively allocate $2048 \times 256 = 524,288$ elements for each array, which is more than sufficient for existing GPUs. Each element in the depth array occupies one byte. For the generation array, each element is allocated 32 bytes, providing one byte for the generation counter corresponding to each of the 32 possible stack depth values (0-31). This results in a total memory overhead of approximately 16.5 MiB, which is negligible for modern GPUs.

VA randomization. Similar to local memory, global memory also faces the problem of metadata confusion. This occurs when the VA of a freed buffer is reclaimed and allocated to a new buffer. Consequently, dereferencing a dangling pointer to the original freed buffer would inadvertently access the new buffer's data, leading to a false negative. To prevent this, CuSafe deploys a robust VA randomization mechanism.

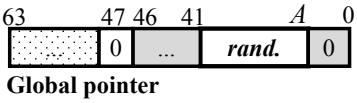

Figure 12: VA randomization.

As shown in Fig. 12, a simplified review of the global pointers from Sec. 4.2, CuSafe randomizes bits [40:*A*] of global pointers to prevent a VA from being reused. Bit *A* is calculated based on the alignment of the allocated buffer. For example, if the buffer size is 16 B, its lower 4 bits are fixed to zero. Therefore, *A* is equal to four, and CuSafe can randomize 37 bits (i.e., [40:4]) of the VA to mitigate metadata confusion. This mechanism creates an expansive random space (i.e., 2^{41-A}), rendering the probability of a VA collision negligible. For example, even in an extreme setting where $A = 33$ (i.e., each allocation is 8 GiB), there are still $2^8 = 256$ possible VAs. A GPU with 80 GiB memory (e.g., H100) can only make ten such allocations before running out of memory. For more common allocation sizes in practice, such as 1 MiB (i.e., $A = 20$), the size of random space is enormous, 2^{21}. We can therefore safely assume that the VA of a freed buffer will not be reused in the same execution.

In contrast, cuCatch [53], generate a fixed random space with size $2^8 = 256$ VAs, irrespective of the allocation size. This makes it considerably more vulnerable to metadata confusion than CuSafe because GPU applications might concurrently hold over 256 live objects [53]. Furthermore, CuSafe's current design is constrained by the available bits in the VA; only bits [46:0] are available for CuSafe's control, as bits [63:47] are fixed to zero by NVIDIA. If this restriction were lifted, CuSafe could significantly extend the random space by randomizing more bits in the VA.

Summary. The detection of temporal violations is integrated into the general process for detecting spatial memory violations. Specifically, CuSafe invalidates a freed memory buffer by clearing its in-band exact bounds to zero. As a result, any subsequent access to the freed buffer will be detected by CuSafe. To prevent metadata confusion caused by mem-

[4] If this number exceeds 32 in future GPUs, CuSafe wraps depth to 0 after reaching 32, falling back to a probabilistic defense; P(collision)=1/32.

ory reallocation, CuSafe additionally checks the stack epoch of local pointers and randomizes VA bits of global pointers.

4.4 Optimization

In this section, we discuss how CuSafe reduces redundant checks to improve its performance. Fig. 13 illustrates the three types of optimizations that CuSafe adopts to reduce these redundant checks.

```
1 __global__ void ker() {
2   __shared__ int A[10];
3   int tid = threadIdx.x;
4   _check(A, tid);   // Cdom
5   if (tid % 32 > 28) {
6     _check(A, tid); // Csub
7   }
8 }
```

(a) Recurring check.

```
1 __global__ void ker(int* A, int N){
2   int l = tIdx.x, s = bDim.x;
3   for (int i=l;i<N;i+=s) {
4     _check(A, i+2);   // Cmax
5   // _check(A, i+1);
6     _check(A, i+0);   // Cmin
7   }
8 }
```

(b) Neighboring check.

```
1 __global__ void ker(int* A, int N){
2   int l=tIdx.x, s=bDim.x;
3   _check(A, 0);     // Cmin
4   _check(A, N - 1); // Cmax
5   for (int i=l;i<N;i+=s) {
6     _check(A, i);
7   }
8 }
```

(c) Loop-inductive check.

Figure 13: Optimizations adopted by CuSafe.

Recurring checks. As shown in Fig. 13(a), a common optimizable code pattern involves the same pointer being dereferenced multiple times. In such cases, a naive instrumentation would insert the same checks repeatedly, causing unnecessary overhead. Our optimization iterates over all checks with the same address and retains only the dominating ones. The execution of a dominating check C_{dom} implies the execution of its subordinate check C_{sub}, allowing the subordinate checks to be safely optimized.

Neighboring checks. Fig. 13(b) shows a case where multiple checks are performed on the same base address within the same basic block. In this scenario, CuSafe identifies these checks and retains only those corresponding to the maximum offset C_{max} and the minimum offset C_{min}. Since these checks share the same base address, the successful validation of both C_{max} and C_{min} logically guarantees the validity of all intermediate checks. Therefore, CuSafe can safely eliminate these intermediate checks to reduce the overhead.

Loop-inductive checks. The final optimization CuSafe employs is the hoisting of loop-inductive checks. As shown in Fig. 13(c), a check inside a loop would normally be executed on every iteration with a varying offset. However, in many common cases (e.g., array iteration), the offset used in these checks is a *loop-inductive variable*. This type of variable changes a fixed value on each iteration, allowing its value to be pre-determined if the number of iterations is known. For example, the index `i` in Fig. 13(c) is a loop-inductive variable. It initializes to zero and increments by a constant value `s` on each iteration. Therefore, CuSafe can hoist the check to the loop's prologue by only checking its initial value and maximum value. Note that a *loop-invariant* check, where the offset remains constant throughout the loop, is a special case of a loop-inductive check. Therefore, loop-invariant checks are optimized in the same manner described above.

5 Implementation

CuSafe is implemented as a transform pass of LLVM 21 and a dynamic library to hook CUDA APIs (e.g., `cudaMalloc`). CuSafe consists of 2,964 lines of C/C++ code. This section details our modifications to the compilation/execution pipeline of CUDA programs.

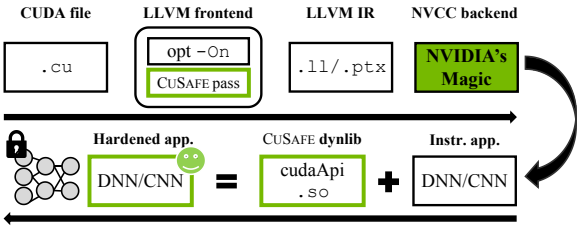

Figure 14: CuSafe-enhanced CUDA pipeline.

Fig. 14 shows CuSafe's integration into the standard CUDA program compilation and execution pipeline. First, CuSafe introduces a new pass within the LLVM compilation pipeline, which processes the Intermediate Representation (IR) to insert necessary checks. Next, NVIDIA's proprietary `nvcc` compiles this instrumented IR into executables. At last, CuSafe replaces memory management APIs (e.g., `cudaMalloc`) through dynamic library preloading (i.e., utilizing `LD_PRELOAD`), thereby ensuring that allocated memory is embedded with the needed metadata.

6 Evaluation

To evaluate CuSafe, we first benchmark its ability to detect different types of GPU memory corruption in Sec. 6.1. We then assess the performance of CuSafe under various GPU workloads in Sec. 6.2.

Evaluation setup. Our evaluation used the following software setup: Linux 6.12.21, NVIDIA driver 570.144 and CUDA

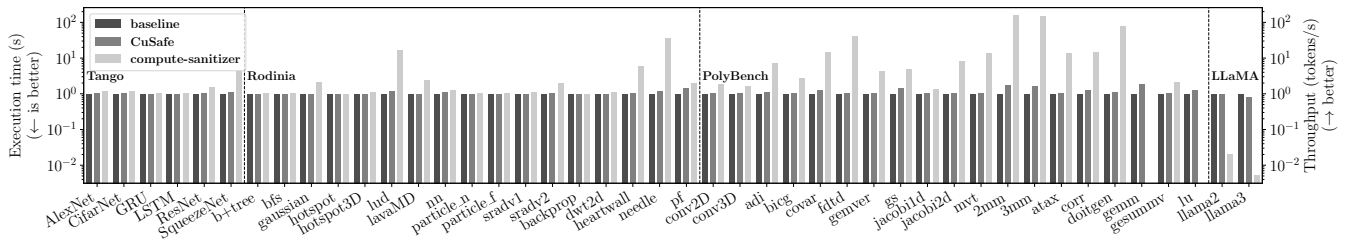

Figure 15: Execution slowdown of CUSAFE on different benchmarks. Note that for LLM benchmarks, we measure the throughput (tokens/s), which is higher is better; we measure other cases' execution time (seconds), which is lower is better. PolyBench's `lu` and `gemm` cannot finish within 60 minutes under compute-sanitizer, we omit them in the figure.

12.8. The hardware setup consists of an AMD Ryzen 9950X and an NVIDIA RTX 5090 with 32 GiB device memory.

6.1 Security Evaluation

Table 3: Security evaluation of CUSAFE.

Type		#	compute sanitizer	GPU Shield[1]	cuCatch[1]	LMI[1]	CuSafe
Spatial	**Linear**	9	3	3	6	3	**9**
	Non-linear	6	0	5	4	6	**6**
Coverage		/	20%	53.3%	66.7%	60%	**100%**
Temporal	**UAF**	12	4	N.A.[2]	9	6	**12**
	IF	2	2	N.A.	2	0	**2**
	DF	4	4	N.A.	4	4	**4**
Coverage		/	55.6%	/	83.3%	66.7%	**100%**

UAF: Use-After-Free, IF: Invalid Free, DF: Double Free.

[1] Due to the lack of public implementations of GPUShield, cuCatch and LMI, we estimated their coverage based on the description in their papers.

[2] GPUShield does not detect temporal memory corruption.

We evaluated CUSAFE using a benchmark similar to those described in the papers of cuCatch and LMI, as neither project has released their security benchmarks. Our benchmarks, which we constructed and extended based on their descriptions, comprise a total of 33 programs: 15 contain spatial memory corruption and 18 contain temporal memory corruption. For spatial vulnerabilities, our benchmarks include both linear and non-linear overflows. Additionally, for local memory overflows, we evaluate both in-frame and cross-frame overflows. As for temporal vulnerabilities, we include cases with both immediate and delayed access. This diverse set of testcases provides a comprehensive evaluation of the detectability of GPU memory safty solutions. Our benchmarks have been released at [5] to support future research. Table 3 summarizes the evaluation results.

Spatial corruption. From the Fig. 3, we can observe that compute-sanitizer detects only three of the 15 spatial memory corruption programs. It misses all non-linear memory overflows and has limited coverage for the linear ones. LMI and GPUShield demonstrate similar coverage as both of them rely on a mechanism that encodes the 2^n-aligned size into the pointer. This approach makes them unable to detect memory overflows that occur within the 2^n range. In contrast, CUSAFE overcomes this limitation by embedding the exact size of a buffer into the metadata, allowing it to accurately detect spatial overflow. While cuCatch also captures the exact bounds of memory buffers, it fails to infer the exact bounds of local/shared memory buffers. This is because the bound information is not available in the compiler's backend, where cuCatch's prototype is implemented. Implemented on the LLVM frontend, CUSAFE can easily obtain the exact bounds of memory buffers for all memory types, thereby avoiding the limitations of cuCatch. Thus, CUSAFE achieves the best coverage for spatial corruption among all evaluated solutions.

Temporal corruption. As for temporal memory corruption, compute-sanitizer detects 10 out of 18 cases; it misses all delayed UAF bugs and an immediate local UAF (i.e., dereferencing a pointer which points to a local variable in a recent-returned function). GPUShield does not offer protection against temporal memory corruption and was excluded from the evaluation. cuCatch detects delayed UAFs probabilistically by assigning one of 256 tags to each allocation. However, when many buffers (>256) are allocated between the use-site and free-site, tag collisions become likely. In our evaluation, cuCatch failed to detect three delayed UAFs due to tag collision. LMI, on the other hand, provides deterministic detection of temporal memory corruption via metadata invalidation. Yet, since its metadata is embedded solely in the pointer, the invalidation of the original pointer does not propagate to the pointers that were copied before the free-site. As a result, LMI misses all the UAFs where copied pointers are involved. In contrast, CUSAFE uses VA-randomization to detect delayed UAFs, which provides stronger heuristic protection (i.e., 2^{41-A} as described in Sec. 4.3). This makes the tag collision seen in cuCatch highly unlikely. Moreover, unlike LMI, CUSAFE embeds the metadata in both the pointer and the buffer, ensuring the invalidation is visible to all use-site, including those of copied pointers. Therefore, CUSAFE achieves the best coverage in detecting temporal corruption.

6.2 Performance Evaluation

We evaluated CUSAFE's performance across a diverse set of 44 testcases, summarized in Table 4. Our evaluation incorpo-

rates both established GPU benchmark suites, such as Rodinia, Tango, and PolyBench-GPU [10, 29, 37], and two popular large language models (LLMs) to assess performance on real-world workloads: LLaMA2 (7B) and LLaMA3 (8B) [23, 54]. All reported overheads are the average of five iterations.

Table 4: Benchmarks used to evaluate CuSafe.

Suite	#	Testcases
Rodinia	17	b+tree, bfs, backprop, lavaMD, bfs, gaussian pathfinder, srad, particlefilter lud, nn, particle_naive, particle_float sradv1, sradv2,hotspot, hotspot3d dwt2d, heartwall, needle, pathfinder
PolyBench	19	conv2d, conv3d, adi, bicg, covar, gramschmidt, jacobi1d, jacobi2d fdtd, gemver, mvt, 2mm, 3mm atax, corr, doitgen, gemm, gesummv, lu
Tango	6	AlexNet, CifarNet, GRU, LSTM, ResNet, SqueezeNet
LLM	2	LLaMA2, LLaMA3

6.2.1 Runtime Overhead

Execution slowdown. To evaluate the execution slowdown of CuSafe, we measured `cudaEventElapsedTime` and excluded routines like host-device `memcpy` and initialization. Fig. 15 presents the performance overhead of CuSafe and compute-sanitizer. On average, CuSafe introduces a 13% overhead to the execution time and an 11% drop on the throughput of LLMs. In contrast, the average overhead of compute-sanitizer is substantially higher; it imposes a 15× overhead and reduces LLM throughput by 98%. For worst case scenarios, compute-sanitizer imposes a 153× overhead on the execution time and a 98.5% reduction on LLM throughput. CuSafe, on the other hand, incurs a maximum overhead of 83% on the execution time (observed on `gemm`) and an 18% drop on LLaMA3's throughput. This maximum overhead of 83% is observed in the `gemm` testcase from PolyBench, which includes a naive implementation of matrix multiplication with sparse memory access. We attribute this overhead to its sparse memory access pattern, which has a negative impact on the cache efficiency. On the same `gemm` testcase, compute-sanitizer incurs an overhead of 153×, which partially supports our analysis. We believe this situation rarely happens in real-world applications with highly optimized CUDA kernels (e.g., LLaMAs)

Other related works, while valuable, are not directly comparable for practical reasons. cuCatch, for instance, reports a 19% average and 3.1× maximum overhead; as cuCatch's evaluation suites are not open-source, we cite its published results[5]. In another domain, hardware-assisted schemes like LMI show promising potential with a minimal 0.2% overhead. However, their effectiveness relies on specialized hardware that is unavailable in current commercial GPUs, limiting their immediate applicability. Our work, in contrast, focuses on a practical solution designed for direct use and evaluation on today's hardware.

Memory overhead. Beyond execution time, memory consumption is another critical metric, with details summarized in Table 5. We omit compute-sanitizer as it is a closed-source tool with no details on its design. LMI's memory overhead is a byproduct of its requirement that all memory be 2^n-aligned, meaning the specific overhead is payload-dependent. cuCatch and CuSafe's memory overhead, on the other hand, consists of a fixed part and a scalable part. cuCatch introduces a fixed overhead of 160 MiB from its initial metadata structure. In addition, it imposes a scalable overhead of 12.5%, requiring 32 bits of metadata for every 32 bytes of allocated memory. CuSafe allocates a fixed memory for the stack epoch, which is only 16.5 MiB (see Sec. 4.3). As for scalable part, CuSafe only attaches an 8-byte in-band size for each allocation.(see Sec. 4.2). Though CuSafe enforces 2^n-alignment on allocated buffers, this alignment is applied only to their VAs. Unlike LMI, CuSafe does not allocate additional physical memory to fill the padding region; the physical memory is mapped only to the actually used portion. Consequently, this alignment introduces no extra memory consumption.

Table 5: Memory overhead analysis.

Tool	Memory overhead
LMI	2^n-aligned fragmentation
cuCatch	Fixed part: 160 MiB; Scale part: 12.5%
CuSafe	Fixed part: 16.5 MiB; Scale part: 8 bytes/alloc

Apart from the above analysis, we also measured the memory overhead of CuSafe and other solutions. Table 6 summarizes the results. Since the implementations of cuCatch and LMI are not available, we first recorded the size of each allocation in the test program, and then calculated the overhead according to the allocation profile and the designs described in their papers. We omit compute-sanitizer in Table 6, as there is neither its design details nor an accurate way to track its memory usage (it does not use `cudaMalloc`). Nonetheless, we estimated its memory usage using `nvidia-smi`, observing an overhead ranging from 200 MiB to 600 MiB, depending on the specific test program.

Table 6: Memory overhead evaluation.

	cuCatch	LMI	CuSafe
Max Relative	160 MB (5121.5x) `jacobi1d`	32 MB (2x) `needle`	16.5 MB (528x) `jacobi1d`
Max Absolute	3.79 GB (13%) `llama2`	6.89 GB (23.7%) `llama2`	16.51 MB (0%) `llama3`
Average	0.73 GB (15%)	1.10 GB (23%)	0.5 MB (0.01%)

Table 6 reports the maximum and average memory overhead . From this table, LMI demonstrates the best performance in terms of relative overhead, which is 2×, while cuCatch's and CuSafe's are 5,121.5× and 528×, respec-

[5] We contacted the authors of cuCatch for this information, but have not received a response.

tively. This superior relative performance by LMI, however, is attributed to its 2^n-aligned memory policy, which inherently bounds its worst-case relative overhead at 2×. The `jacobi1d` is also a special case; it allocates a very small amount of memory (32 KiB), allowing the fixed overheads of cuCatch (160 MiB) and CuSafe (16.5 MiB) to dominate its relative overhead. Regarding absolute overhead, both cuCatch and LMI incur significant memory overheads of 3.79 GiB and 6.89 GiB on the LLM benchmark, whereas CuSafe only requires 16.51 MiB of additional memory. Concerning average overhead, LMI performs the worst, incurring an average overhead of 23% and 1.10 GiB of additional memory. cuCatch has a moderate average overhead of 15% with 0.73 GiB. CuSafe, on the other hand, has a negligible average overhead of only 0.3% and requires merely 16.5 MiB of additional memory.

For LLMs, the memory overhead from cuCatch and LMI might trigger out-of-memory errors, causing functional LLM applications to crash unexpectedly. In contrast, CuSafe adds negligible memory overhead, making such errors highly unlikely. This property is crucial for LLMs, where memory usage is substantial and sensitive to even modest increases.

6.2.2 Occupancy and Divergence

Occupancy and divergence are two important concepts in GPU programming. Occupancy refers to the number of active warps per SM, which depends on the hardware resources used by each thread and reflects kernel parallelism. Low occupancy indicates that the GPU is underutilized, potentially leading to performance degradation. We used `nsight` [44] to measure the occupancy of CUDA kernels before and after CuSafe's instrumentation. Across the 113 CUDA kernels analyzed (one GPU application may contain multiple kernels), only 13 showed an occupancy drop of more than 10% after instrumentation, six of which involve matrix multiplication (e.g., `gemm`). The largest drop, 50%, was observed in the `lavaMD` testcase from Rodinia. Notably, occupancy changes did not directly correlate with CuSafe's runtime overhead: `gemm` had the highest overhead of 83% but only a 24% occupancy drop, while `lavaMD` had a negligible overhead (<1%) but a 50% drop. This indicates that CuSafe's overhead primarily comes from the extra time spent on instrumented instructions, rather than from reduced kernel occupancy.

Divergence refers to the situation where threads in a warp take different execution paths on the same branch, causing the divergent instructions to be executed serially, leading to slowdown. Though CuSafe inserts additional conditional branches, these branches are solely for checking the validity of the memory access. In other words, divergence in these checks indicates memory corruption, and therefore the program should be terminated. Therefore, such divergence is not expected in normal execution. To confirm this, we measured the divergence rate before and after CuSafe's instrumentation, and observed no measurable increase.

6.2.3 Optimized Checks

In this section, we evaluate the effectiveness of CuSafe's optimizations by measuring both the number of checks removed and the resulting performance improvement. As shown in Fig. 16, applying the three optimization rules described in Sec. 4.4 eliminates, on average, near 20% of checks across the 44 GPU programs used in our performance evaluation. These results demonstrate the effectiveness of the proposed rules in identifying and removing redundant checks.

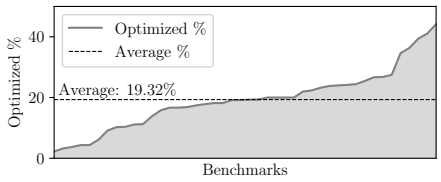

Figure 16: Percentage of checks optimized by CuSafe.

Performance gain. We also measured the performance improvement from CuSafe's optimizations. On average, these optimizations reduce the execution time by 3.5% and improve LLM's throughput by nearly 2%. The most significant performance gain is observed in the `lud`, where CuSafe's optimizations reduces the execution time by over 60%. We attribute this to the large number of shared memory accesses in `lud`; removing the redundant checks allows the compiler to better optimize the shared memory access pattern, leading to substantial performance improvement.

7 Related Work

This section delves into memory safety solutions for CPUs, Many of which significantly influenced CuSafe's design. This discussion complements our earlier review of GPU-based approaches in Sec. 2.3.

Extensive efforts have been devoted to developing efficient and effective CPU memory safety solutions. A popular approach stores pointer bounds in the metadata and check their validity upon dereference. Nagarakatte et al. [39] proposed SoftBound, proving that storing metadata in a disjoint (i.e., out-of-band) structure provides complete spatial memory safety. During the same period, BaggyBounds [3] was also proposed. BaggyBounds enforces all allocation to be 2^n-aligned, which helps reduce metadata size and enables more efficient out-of-band metadata structure. The same 2^n-alignment idea is also adopted by CuSafe and LMI. At that time, memory safety solutions still primarily relied on out-of-band structures to store metadata. LowFat [15, 16] changed this situation. By re-arranging the VA space, LowFat directly embeds the metadata into the pointer itself and therefore requires no out-of-band metadata structure. This approach also inspired CuSafe to leverage GPU MMU to achieve low-cost

memory tagging. Other studies, like DeltaPointer [30], embed a special value in a pointer's higher bits, mimicking pointer arithmetic on that value to achieve auto-invalidation. In addition to relying on software for memory safety, many studies [11, 22, 35, 57] also explored emerging hardware features like MTE [6]. MTSan [11] utilizes MTE to detect memory errors in closed-source software, while PACSan [35] employs a similar hardware feature named PAC [1] to tag pointers. StickyTag [22] improves MTE's performance by making MTE tags permanently attached to buffers, which therefore eliminate the need for memory retagging. BOGO [57] leverages MPX [46] to track memory bounds.

Another common approach is to record the status of memory bytes (e.g., allocated or freed) in an out-of-band shadow memory. One of the most widely used sanitizers ASan [50], adopts this approach. Due to its reliance on out-of-band shadow memory, ASan incurs a significant overhead of nearly 100%. To reduce this overhead, various improvements have been proposed. For example, ASan-- [58] reduces ASan's overhead by removing redundant checks. Gorter et al. [21] repurposed the floating-point unit to accelerate memory bound checking. RSan [19] introduces an in-band redzone scheme. By storing the buffer size in the unused redzone area, RSan validates a memory range with a single check, showing significantly better performance than ASan.

There is also a line of studies solely focusing on capturing temporal corruption [18, 20, 32, 55]. DangSan and DangNull [32, 55] track all pointers' locations and invalidate them once their target object is freed. DangZero [20] leverages MMU to achieve efficient pointer invalidation, while PTAuth [18] utilizes PAC [1] to validate dereferenced pointers and thus detects UAF errors.

Comparison with RSan. Careful readers might notice that CuSafe shares similar conceptual elements (e.g., in-band metadata) with CPU sanitizers such as RSan [19]. Parts of CuSafe's design are inspired by RSan. However, CuSafe and RSan aim to solve different challenges despite the similarity in their methods. Specifically, CuSafe leverages in-band metadata to handle highly-paralleled memory access in GPU environments, while RSan uses it to reduce redundant metadata retrieval. Moreover, CuSafe mitigates tag alteration through pointer arithmetic validation, whereas RSan lacks a similar mechanism. While this mechanism introduces compatibility issues for programs with benign overflows (see Sec. 8), it also protects against inadvertent alterations to CuSafe's metadata, a common problem in GPU programs (e.g., integer overflows [28]). RSan omits this mechanism, likely due to compatibility concerns, and mature CPU-side tools such as UBSan [36] already detect such errors at compile time. CuSafe provides fundamental temporal protection for both global and local pointers since previous literature [24] reveals that NVIDIA stores the return address on stack, thereby enabling attacks like return-oriented programming. RSan, on the other hand, focuses on the spatial errors. This is probably because there are already extensive literatures aim to detect temporal errors [20, 32, 55], and standard CPU-side compilers are mature enough to detect many temporal issues with local pointers.

8 Discussion

Intra-object overflow. Currently, CuSafe only tracks bounds at object-level (e.g., a `struct`). As a result, it cannot detect overflows that occur between two fields within the same object. Theoretically, CuSafe can be extended to capture these errors by treating each field as an individual object. However, this might significantly increase the number of tracked objects and incur substantial performance overhead.

Benign overflow. In Sec. 4.2, we discussed how CuSafe prevents arithmetic overflows from altering metadata. Specifically, CuSafe invalidates a pointer by setting its highest bit to one when detecting such altering. This mechanism raises false alarms when a program temporarily creates an overflowed pointer and later restores it to a valid one. We clarify that CuSafe tolerates such benign overflows within a 2^n-byte range. Fig. 17 shows such an example. In this example, though the pointer `p` overflows (line 4), it is not invalidated as the overflow falls within the 1 KB boundary and does not modify CuSafe's metadata. Therefore, it is regarded as a benign overflow, and the recovered pointer at line 5 is not affected by the invalidation mechanism.

```
1 //600B; 1KB-aligned;
2 char p[600];
3 //+700 not invalidated(<1024)
4 p += 700;
5 p -= 700; //recovered
```

Figure 17: CuSafe's tolerance to benign overflow.

That said, if a program increment `p` by 1024 and decrement it back to its original value, CuSafe will mark `p` as invalid by setting its highest bit to one, despite the pointer still being valid. This is because such an increment exceeds `p`'s 2^n-aligned boundary and might alter the pointer's tag bits. Though CuSafe indeed raises false alarms in this case, we believe such scenarios are rare and does not impair CuSafe's overall deployability.

Uninstrumented libraries. Since CuSafe is designed as an LLVM pass in the compiler's frontend, it cannot instrument closed-source modules. In the current CUDA ecosystem, a significant portion of such modules comprises NVIDIA's proprietary libraries, such as `cuDNN` and `cuBLAS` [12]. Nevertheless, CuSafe can still interact with these uninstrumented libraries safely by stripping CuSafe's metadata before invoking them. As mentioned in Sec. 4.1, only local/shared pointers require stripping. Global pointers in CuSafe are tagged by

the MMU and can be safely dereferenced by an uninstrumented library. Because global-pointer tagging is enabled via `LD_PRELOAD`, even if a library is uninstrumented, its global allocations are still intercepted and annotated with the required metadata. By contrast, for local/shared pointers returned by these libraries, CuSafe can avoid false positives by checking tag bits: CuSafe assigns tags starting at one, while untagged pointers have their tag bits set to zero. Moreover, these libraries are highly optimized and extensively tested, thus reducing the likelihood of memory bugs. In addition, open-source alternatives exist, such as `MAGMA` and `ArrayFire` [13, 56].

Unsupported memory. Though CuSafe supports most of the memory types in the NVIDIA GPUs (i.e., local, global, and shared memory), it does not support two rarely used memory APIs, dynamic shared memory and in-kernel `malloc`. Dynamic shared memory allows the host to dynamically allocate a part of the shared memory before kernel launch. Since its allocation is exclusively managed by the NVIDIA's proprietary runtime via a unique kernel launch syntax (i.e., $\texttt{<<<dim}_{\texttt{grid}}\texttt{,dim}_{\texttt{blk}}\texttt{,size}_{\texttt{shared}}\texttt{>>>}$), CuSafe cannot intercept the allocation to associate metadata. In-kernel `malloc` enables CUDA kernels to allocate global memory directly. Due to similar challenges in intercepting such allocations, CuSafe does not support this feature. Fortunately, these two features are rarely used in practice. Shared memory is very limited in size (for example, 64 KiB) where static allocation is sufficient. As for in-kernel `malloc`, it incurs a significant performance penalty [43] compared to its host-side counterpart (i.e., `cudaMalloc`), and is therefore seldom used.

Other platforms. Though CuSafe is designed for NVIDIA GPUs, the underlying mechanism of CuSafe does not depend on features specific to NVIDIA GPUs. Modern GPUs from other vendors (e.g., AMD, Intel and Apple) also support MMUs, making them viable platforms for implementing CuSafe. However, the openness of GPU APIs varies across vendors. For example, AMD provides an open-source CUDA-like API named HIP [4], which could facilitate a swift port of CuSafe to AMD GPUs. In contrast, Intel and Apple are more restrictive regarding their GPU APIs, particularly low-level memory management APIs that CuSafe requires. This presents challenges for porting CuSafe to those platforms. Nonetheless, since low-level memory control is essential for high-performance computing, we expect that other vendors will eventually expose such APIs.

Other compiler frameworks. Currently, CuSafe's prototype is only implemented within the LLVM framework. Consequently, it does not currently support features like JIT compilation. However, since CuSafe's required features widely exist in modern GPUs, we believe that it would take only moderate engineering effort to implement CuSafe

9 Conclusion

CuSafe is the first practical GPU memory sanitizer that effectively detects both spatial and temporal errors on commodity NVIDIA GPUs. It leverages only off-the-shelf hardware and open toolchains. By combining pointer tagging with in-band metadata, it ensures precise and low-overhead memory safety. Evaluated on diverse benchmarks and LLMs, CuSafe achieves full detection coverage with 13% runtime overhead and minimal memory use.

Ethical Considerations

We carefully considered the ethical implications of this work throughout the research process. This paper introduces a software-based memory safety solution for commodity NVIDIA GPUs, aiming to improve system reliability and security without introducing new risks. We carefully performed fair and reproducible comparisons between CuSafe and available competing systems. Specifically, we used identical software/hardware configurations, maintaining consistent baselines, and reporting results based on the average of multiple runs. We list the primary stakeholders, the impacts of our research, our mitigation against the potential risks, and why we have decided to publish the paper as follows.

Stakeholders. We identify three primary stakeholders (GPU manufacturers, system developers and general users) and discuss CuSafe's potential impacts below.

GPU manufacturers. Our approach primarily targets NVIDIA GPUs. That said, CuSafe introduces no new attack vectors for existing CUDA programs or NVIDIA GPUs, posing no additional risk to the CUDA ecosystem or to NVIDIA's products. By contrast, CuSafe detects memory bugs in CUDA programs, strengthening the overall reliability of the platform.

System developers. With strong memory bug detection capabilities, system developers can use CuSafe to identify memory bugs at early stages of development, building more reliable CUDA programs.

General users. As for general users, the adoption of CuSafe significantly reduces the risk of exploitation, protecting them from malicious adversaries.

Mitigation. Because our experiments involve memory vulnerabilities, to avoid unintended consequences we conducted our security evaluation locally with our own synthetic benchmarks derived from known bugs rather than real-world programs. Furthermore, our research focuses on detecting memory bugs rather than exploiting them. As a result, our experiments did not introduce any new attack vectors or enable exploitation.

Decision to publish the paper. Given the emerging threat of GPU memory vulnerabilities, we believe that our research on CuSafe is necessary and has the potential to foster a healthier computing ecosystem. Therefore, the publication of this paper is justified. CuSafe offers strong detectability for

GPU memory bugs, and its moderate overhead suggests its suitability for wider deployment.

Open Science

We confirm that the submitted paper follows the open science policy of USENIX Security' 26. The source code of CuSafe (LLVM pass, dynamic linking library, and all benchmarks) is publicly available at [5]. We will continue to maintain the source code for future research after the paper is accepted.

References

[1] Arm pointer authentication code, 2025. URL https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Learn%20the%20Architecture/Providing%20protection%20for%20complex%20software.pdf.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

[3] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, volume 10, page 96, 2009.

[4] AMD. Hip documentation — hip 6.4.43484 documentation. 2025. URL https://rocm.docs.amd.com/projects/HIP/en/latest/.

[5] Anonymous. CuSafe website. 2025. URL https://doi.org/10.6084/m9.figshare.30821396.

[6] Arm. MTE manual. 2025. URL https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[7] Ran Bi, Tongtong Xu, Mingxue Xu, and Enhong Chen. Paddlepaddle: A production-oriented deep learning platform facilitating the competency of enterprises. In *2022 IEEE 24th Int Conf on High Performance Computing Communications; 8th Int Conf on Data Science Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud Big Data Systems Application (HPCC/DSS/SmartCity/DependSys)*, pages 92–99, 2022. doi: 10.1109/HPCC-DSS-SmartCity-DependSys57074.2022.00046.

[8] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[9] ChaitanyaRS06. Tensor.long() produces inconsistent results for torch.inf between cpu and gpu · issue 154724 · pytorch/pytorch. 2025. URL https://github.com/pytorch/pytorch/issues/154724.

[10] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009. doi: 10.1109/IISWC.2009.5306797.

[11] Xingman Chen, Yinghao Shi, Zheyu Jiang, Yuan Li, Ruoyu Wang, Haixin Duan, Haoyu Wang, and Chao Zhang. MTSan: A feasible and practical memory sanitizer for fuzzing COTS binaries. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 841–858, Anaheim, CA, August 2023. USENIX Association. ISBN 978-1-939133-37-3. URL https://www.usenix.org/conference/usenixsecurity23/presentation/chen-xingman.

[12] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning, 2014. URL https://arxiv.org/abs/1410.0759.

[13] Andrzej Chrzeszczyk and Jacob Anders. Matrix computations on the gpu cublas, cusolver and magma by example. 2025. URL https://d29g4g2dyqv443.cloudfront.net/sites/default/files/akamai/cuda/files/Misc/mygpu.pdf.

[14] Bang Di, Jianhua Sun, Dong Li, Hao Chen, and Zhe Quan. Gmod: A dynamic gpu memory overflow detector. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pages 1–13, 2018.

[15] Gregory J Duck and Roland HC Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 132–142, 2016.

[16] Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. Stack bounds protection with low fat pointers. In *NDSS*, volume 17, pages 1–15, 2017.

[17] Christopher Erb and Joseph L. Greathouse. clarmor: A dynamic buffer overflow detector for opencl kernels. In *Proceedings of the International Workshop on OpenCL*, IWOCL '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450364393. doi: 10.1145/3204919.3204934. URL https://doi.org/10.1145/3204919.3204934.

[18] Reza Mirzazade farkhani, Mansour Ahmadi, and Long Lu. PTAuth: Temporal memory safety via robust points-to authentication. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1037–1054. USENIX Association, August 2021. ISBN 978-1-939133-24-3. URL https://www.usenix.org/conference/usenixsecurity21/presentation/mirzazade.

[19] Floris Gorter and Cristiano Giuffrida. Rangesanitizer: Detecting memory errors with efficient range checks. 2025. URL https://www.usenix.org/conference/usenixsecurity25/presentation/gorter.

[20] Floris Gorter, Enrico Barberis, Raphael Isemann, Erik van der Kouwe, Cristiano Giuffrida, and Herbert Bos. FloatZone: Accelerating memory error detection using the floating point unit. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 805–822, Anaheim, CA, August 2023. USENIX Association. ISBN 978-1-939133-37-3. URL https://www.usenix.org/conference/usenixsecurity23/presentation/gorter.

[21] Floris Gorter, Enrico Barberis, Raphael Isemann, Erik van der Kouwe, Cristiano Giuffrida, and Herbert Bos. FloatZone: Accelerating memory error detection using the floating point unit. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 805–822, Anaheim, CA, August 2023. USENIX Association. ISBN 978-1-939133-37-3. URL https://www.usenix.org/conference/usenixsecurity23/presentation/gorter.

[22] Floris Gorter, Taddeus Kroes, Herbert Bos, and Cristiano Giuffrida. Sticky tags: Efficient and deterministic spatial memory error mitigation using persistent memory tags. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4239–4257, 2024. doi: 10.1109/SP54263.2024.00263.

[23] Aaron Grattafiori et al. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

[24] Yanan Guo, Zhenkai Zhang, and Jun Yang. GPU memory exploitation for fun and profit. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4033–4050, Philadelphia, PA, August 2024. USENIX Association. ISBN 978-1-939133-44-1. URL https://www.usenix.org/conference/usenixsecurity24/presentation/guo-yanan.

[25] Mark Harris. Using shared memory in cuda c/c++ | nvidia technical blog. 2025. URL https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/.

[26] J. A. Herdman, W. P. Gaudin, O. Perks, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis. Achieving portability and performance through openacc. In *2014 First Workshop on Accelerator Programming using Directives*, pages 19–26, 2014. doi: 10.1109/WACCPD.2014.10.

[27] jwnhy. [cuda] illegal memory access with 'convtranspose2d' · issue 144611 · pytorch/pytorch. 2025. URL https://github.com/pytorch/pytorch/issues/144611.

[28] jwnhy. [cuda] illegal memory access with 'adaptiveavgpool2d' · issue 145349 · pytorch/pytorch. 2025. URL https://github.com/pytorch/pytorch/issues/145349.

[29] Aajna Karki, Chethan Palangotu Keshava, Spoorthi Mysore Shivakumar, Joshua Skow, Goutam Madhukeshwar Hegde, and Hyeran Jeon. Tango: A deep neural network benchmark suite for various accelerators. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 137–138, 2019. doi: 10.1109/ISPASS.2019.00021.

[30] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta pointers: buffer overflow checks without the checks. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355841. doi: 10.1145/3190508.3190553. URL https://doi-org.lib.ezproxy.hkust.edu.hk/10.1145/3190508.3190553.

[31] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. page 75–88, San Jose, CA, USA, Mar 2004.

[32] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.

[33] Jaewon Lee, Yonghae Kim, Jiashen Cao, Euna Kim, Jaekyu Lee, and Hyesoon Kim. Securing gpu via region-based bounds checking. In *Proceedings of the 49th*

Annual International Symposium on Computer Architecture, ISCA '22, page 27–41, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450386104. doi: 10.1145/3470496.3527420. URL https://doi.org/10.1145/3470496.3527420.

[34] Jaewon Lee, Euijun Chung, Saurabh Singh, Seonjin Na, Yonghae Kim, Jaekyu Lee, and Hyesoon Kim. Let-me-in:(still) employing in-pointer bounds metadata for fine-grained gpu memory safety. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1648–1661. IEEE, 2025.

[35] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. Pacsan: Enforcing memory safety based on arm pa, 2022. URL https://arxiv.org/abs/2202.03950.

[36] llvm. Undefinedbehaviorsanitizer. 2025. URL https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[37] Suejb Memeti, Lu Li, Sabri Pllana, Joanna Kolodziej, and Christoph Kessler. Benchmarking opencl, openacc, openmp, and cuda: programming productivity, performance, and energy consumption, 2017. URL https://arxiv.org/abs/1704.05316.

[38] Andrea Miele. Buffer overflow vulnerabilities in cuda: a preliminary analysis, 2015. URL https://arxiv.org/abs/1506.08546.

[39] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 245–258, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583921. doi: 10.1145/1542476.1542504. URL https://doi.org/10.1145/1542476.1542504.

[40] NVIDIA. Dgx b200: The foundation for your ai factory | nvidia. 2025. URL https://www.nvidia.com/en-us/data-center/dgx-b200/.

[41] NVIDIA. The engine behind ai factories | nvidia blackwell architecture. 2025. URL https://www.nvidia.com/en-us/data-center/technologies/blackwell-architecture/.

[42] NVIDIA. Compute sanitizer, 2025. URL https://developer.nvidia.com/compute-sanitizer.

[43] NVIDIA. kernel malloc() efficiency really bad - cuda / cuda programming and performance - nvidia developer forums. 2025. URL https://forums.developer.nvidia.com/t/kernel-malloc-efficiency-really-bad/20575.

[44] NVIDIA. Nvidia nsight systems | nvidia. 2025. URL https://developer.nvidia.cn/nsight-systems.

[45] NVIDIA. Nvidia/open-gpu-kernel-modules: Nvidia linux open gpu kernel module source. 2025. URL https://github.com/NVIDIA/open-gpu-kernel-modules.

[46] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2018.

[47] Sang-Ok Park, Ohmin Kwon, Yonggon Kim, Sang Kil Cha, and Hyunsoo Yoon. Mind control attack: Undermining deep learning with gpu memory exploitation. *Computers & Security*, 102:102115, 2021. ISSN 0167-4048. doi: https://doi.org/10.1016/j.cose.2020.102115. URL https://www.sciencedirect.com/science/article/pii/S0167404820303886.

[48] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[49] Jonas Roels, Adriaan Jacobs, and Stijn Volckaert. Cuda, woulda, shoulda: Returning exploits in a sass-y world. In *Proceedings of the 18th European Workshop on Systems Security*, EuroSec'25, page 40–48, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400715631. doi: 10.1145/3722041.3723099. URL https://doi.org/10.1145/3722041.3723099.

[50] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 28, USA, 2012. USENIX Association.

[51] SilentTester73. Crash in 'sparsebincount' due to overflow · issue 94118 · tensorflow/tensorflow. 2025. URL https://github.com/tensorflow/tensorflow/issues/94118.

[52] Michael B. Sullivan, Mohamed Tarek Ibn Ziad, Aamer Jaleel, and Stephen W. Keckler. Implicit memory tagging: No-overhead memory safety using alias-free tagged ecc. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700958. doi: 10.1145/3579371.3589102. URL https://doi.org/10.1145/3579371.3589102.

[53] Mohamed Tarek Ibn Ziad, Sana Damani, Aamer Jaleel, Stephen W Keckler, and Mark Stephenson. Cucatch: A debugging tool for efficiently catching memory safety violations in cuda applications. *Proceedings of the ACM on Programming Languages*, 7(PLDI):124–147, 2023.

[54] Hugo Touvron et al. Llama 2: Open foundation and fine-tuned chat models, 2023. URL https://arxiv.org/abs/2307.09288.

[55] Erik Van Der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 405–419, 2017.

[56] Pavan Yalamanchili, Umar Arshad, Zakiuddin Mohammed, Pradeep Garigipati, Peter Entschev, Brian Kloppenborg, James Malcolm, and John Melonakos. ArrayFire - A high performance software library for parallel computing with an easy-to-use API, 2015. URL https://github.com/arrayfire/arrayfire.

[57] Tong Zhang, Dongyoon Lee, and Changhee Jung. Bogo: Buy spatial memory safety, get temporal memory safety (almost) free. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 631–644, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362405. doi: 10.1145/3297858.3304017. URL https://doi.org/10.1145/3297858.3304017.

[58] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu. Debloating address sanitizer. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4345–4363, Boston, MA, August 2022. USENIX Association. ISBN 978-1-939133-31-1. URL https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-yuchen.

[59] Mohamed Tarek Ibn Ziad, Sana Damani, Mark Stephenson, Stephen W. Keckler, and Aamer Jaleel. Gpuarmor: A hardware-software co-design for efficient and scalable memory safety on gpus, 2025. URL https://arxiv.org/abs/2502.17780.