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 bugs 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 uniquely achieves comprehensive coverages of both spatial and temporal bugs among existing GPU sanitizers. 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 [1, 4, 5, 40]. Furthermore, the fast-evolving ecosystem around CUDA and OpenACC [20] 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 [6, 22, 23, 43] has been reported in the PyTorch framework [40]. Similar cases have also been reported in other popular frameworks such as TensorFlow [1] and PaddlePaddle [4]. 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 [18, 31, 39]. For example, Guo et al. [18] 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 [39]. 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 [11, 12, 26, 27, 45]. However, these solutions often fall short of comprehensive protection, with each approach facing unique drawbacks. For instance, LMI [27] and GPUShield [26] rely on customized hardware modifications that are not available on commodity GPUs, making them infeasible for real-world deployment. cuCatch [45] faces the same problem since it needs to modify the NVIDIA's proprietary toolchains which are not available to the public. Beyond deployment hurdles, these approaches are also limited by their design choices. For instance, cuCatch cannot accurately detect local memory overflows as it lacks precise bounds information from the compiler's frontend. GMOD [11] and clArmor [12] take a canary-based approach that can only detect adjacent memory overflows.

In this paper, we introduce CuSafe, the first GPU sanitizer that is deployable on commodity NVIDIA GPUs, providing comprehensive 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 [13], PolyBench [30], Tango [24], and large language models (LLMs), including LLaMA2-7B [46] and LLaMA3-8B [17]. 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 comprehensive 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 comprehensive detection capability.
- Technically, CuSafe incorporates various mechanisms to achieve comprehensive 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 [25] 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 [3] to promote future research.

2 Background and 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.

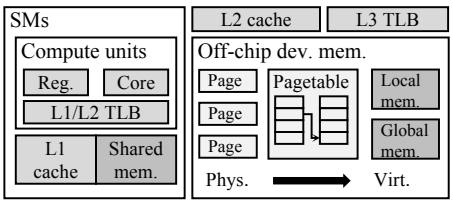

Figure 1: GPU memory architecture.

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 [19]. 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 [38]; 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. 2.2.

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 accesses to pointers that are widely dispersed across memory) to prevent TLB thrashing. In our observation, retrieving sanitizer metadata from a shadow memory region as in cuCatch [45] and AddressSanitizer (ASAN) [42] 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 Pointer Tagging

CUSAFE relies on pointer tagging to retrieve the metadata (see Sec. 4.1); we discuss its implementation on GPUs.

MMU-based global pointer tagging. Though major CPU vendors such as Intel have announced their pointer tagging features (e.g., Intel LAM [21]), no equivalent features exist on GPU. Fortunately, NVIDIA's open-source driver exposes interfaces for managing the allocation of global memory, which allow manipulation of virtual addresses for global buffers via the GPU MMU. We thereby leverage the MMU to customize the upper bits of the virtual address (VA) to encode metadata, such as tags.

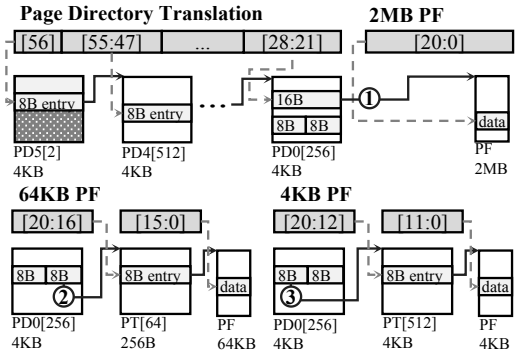

Figure 2: GPU MMU page table format.

For instance, Fig. 2 shows the page table format of Blackwell[1] GPUs [34]. As shown in the figure, a modern GPU supports multiple page sizes, including ① 4 KB, ② 64 KB and ③ 2 MB. 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).

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

These PDx tables govern the translation result of a VA; therefore, by configuring the page table, we can embed metadata within the higher bits of the VA, effectively making these bits part of the pointer's tag. For example, CUSAFE enforces all objects of 256 B to be allocated in the VA region with bits [46:41]$= log(256) - log(16) + 1 = 6$, with 16 B being the minimal allocation size. This approach enables flexible tagging of global buffer pointers through judicious page table configuration.

Local/shared pointer tagging. For objects allocated in the local/shared memory, their virtual addresses cannot be arbitrarily manipulated for tagging, unlike those in global memory. This limitation arises because these addresses are exclusively managed by NVIDIA's closed-source runtime. Instead, we adopt a pseudo-pointer based approach. Fig. 3 presents a code snippet that illustrates the tagging process of local memory pointers. In this snippet, we allocate a local memory buffer of 24 bytes with 32-byte alignment (line 1). Subsequently, at line 2, a pseudo-pointer `%tagged` is constructed by embedding the necessary metadata (e.g., its 2^n aligned size) into the original pointer's value. As `%tagged` is not a directly dereferencable memory address, the tagging procedure must be reversed at line 4 to recover the original, valid pointer `%raw`. This raw pointer is then used to access the memory buffer at line 5. This technique allows local/shared memory buffers to be effectively tagged, albeit with a slight computational overhead due to the additional arithmetic operations.

```
1 %obj = alloca 24 align 32
2 %tagged = or %shifted, %tag
3 ...
4 %raw = and %shifted, %tagMask
5 %val = load %raw
```

Figure 3: Local/shared memory tagging.

Compatibility. A potential compatibility arises if a "pseudo-pointer" is passed to an external function, as the function would be unable to dereference it directly. While such scenarios are infrequent in typical CUDA applications (specifically, passing a local/shared pointer to an external function), CUSAFE mitigates this by automatically untagging such pointers before they are passed to external functions.

2.3 Memory Safety Solutions on GPUs

Memory safety is a long-standing problem. This section reviews various memory safety solutions designed for GPUs. Table 1 summarizes these approaches. Though some tools (e.g., GPUShield) might not be formally named "sanitizers", we categorize them as such for simplicity, given their common function of detecting memory corruption.

Redzone-based. Redzone is a widely used technique to detect memory corruption. A redzone-based sanitizer typically allocates a shadow memory region to track the status of the main

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

Name	Base	Mechanism	Spatial	Temporal	Deployability	Perf. Overhead	Mem. Overhead
GPUShield [26]	HW	Tagging	●	○	✘	1%	None
IMT [44]	HW	Tagging	◐	○	✘	4%	None
LMI [27]	HW	Aligning	◐	◐	✘	0.2%	2^n-fragmentation
GMOD [11]	SW	Redzone	◐	○	✔	2.9%	12 B/buf
clArmor [12]	SW	Redzone	◐	○	✔	9.6%	4 B/buf
compute-sanitizer [35]	SW	Redzone	◐	◐	✔	1,400%	Unknown
cuCatch [45]	SW	Tagging	◐	◐	✘	19%	160 M + 12.5%
CUSAFE	SW	Hybrid	●	●	✔	**13%**	**16.5 M + 8 B/buf**

memory and checks if the accessed memory has been flagged as "redzone" during runtime. An example of a redzone-based sanitizer is NVIDIA's compute-sanitizer [35]. It relies on binary rewriting to insert checks before each memory access, and thus introduces a considerable performance penalty due to the additional instructions inserted. As shown in Table 1, our evaluation shows that compute-sanitizer leads to an average slowdown of 15×. Furthermore, as this approach primarily places redzones between buffers, it cannot detect overflows that occur from one buffer into an adjacent, valid buffer (e.g., an overflowed access from buffer A into buffer B).

Other than compute-sanitizer, there are also GPU sanitizers like GMOD and clArmor [11, 12] that are canary-based. By inserting a guard value at the end of each memory allocation, these sanitizers detect memory corruption by checking if the guard value is modified. As Table 1 shows, though this simplified approach significantly reduces the performance overhead compared to compute-sanitizer, it also loses the capability to detect temporal memory corruption (e.g., use-after-free) and provides even worse detection accuracy; any overflow that skips the guard value is not detected.

Tagging-based. Tagging-based sanitizers attach a tag to each pointer. When memory access occurs, the sanitizer retrieves the metadata associated with the tag and checks if the access is valid. For instance, NVIDIA's cuCatch [45] embeds a tag into each pointer and queries a two-level structure for the valid bounds of the buffer. Built upon the compiler's backend, cuCatch shows superior performance (19% overhead as shown in Table 1) compared to approaches that rely on binary rewriting (e.g., compute-sanitizer). However, relying on the backend restricts it from obtaining accurate bounds, as these are only available on the frontend; this leads to inaccurate detection. Moreover, cuCatch's tags could potentially be tampered with as cuCatch directly encodes the tag into the pointer yet does not check if a pointer arithmetic overwrites it. GPUShield [26] adopts a similar approach to cuCatch and uses a customized hardware component to perform the bounds check. Though the customized hardware reduces the runtime overhead, it also prevents GPUShield from being deployed on commercial GPUs, where the hardware is not modifiable. IMT [44] is another hardware-based solution that uses ECC metadata as tags. Similar to GPUShield, IMT also requires hardware modification and therefore faces similar deployment challenges as GPUShield. Therefore, we mark all the hardware-based solutions in Table 1 as undeployable. Moreover, IMT does not specifically target memory safety but instead provides a general approach for pointer tagging.

Aligning-based. Aligning-based sanitizers take a different approach compared to the above ones. Instead of performing bound checking at memory access, they check pointers' bounds during arithmetic instructions. Such approaches enforce the alignment of the allocated memory to 2^n and encode the alignment n into the pointers. During the pointer arithmetic, the sanitizer checks if the resulting pointer is still within the 2^n bounds and terminate the program if it is not. LMI [27] is a recent work that adopts this approach on GPUs. LMI encodes the alignment information into the pointers and modifies the ALUs to restrict pointer arithmetic. However, like other pointer aligning schemes, LMI misses overflow that occurs within the 2^n alignment and introduces substantial memory waste due to fragmentation. Furthermore, LMI has only limited support for temporal memory corruption (e.g., use-after-free); it marks a pointer as invalid upon the free-site and thus cannot detect corruption if the pointer is copied prior to the free operation. Lastly, similar to GPUShield, LMI also requires modification of the GPU hardware, which limits its deployment on commercial GPUs. Consequently, we mark its detectability of both types of memory corruption as partial, and mark it as undeployable, in Table 1.

Summary. Table 1 CUSAFE with current memory safety solutions on 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 comprehensive detection for both spatial and temporal memory corruption.

3 Motivation

Characteristics of GPU workloads. Since GPUs are designed for highly parallel workloads, it is common for GPU

applications to execute thousands of threads concurrently. This high degree of parallelism directly leads to a need for high-throughput memory access. However, current GPU sanitizers, such as cuCatch, leverage 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 GPU sanitizers is the difficulty of deployment. Currently, the only sanitizer supported on commodity NVIDIA GPUs is the compute-sanitizer [35], which, as shown in Table 1, has limited detection capability and substantial performance overhead (15× on average). Recent studies such as cuCatch and LMI [27, 45] either depend on the proprietary NVIDIA toolchains or customized hardware, making them inaccessible to academic researchers and community developers.

Insufficient detection capability. Existing work such as cuCatch often achieves only partial detection of memory corruption. As shown in Table 1, cuCatch is implemented at the compiler backend and cannot obtain accurate information about allocation sizes from the frontend, leading to incomplete detection. Moreover, cuCatch uses a heuristic scheme with limited entropy ($2^8 = 256$) to detect temporal corruption, which misses corruption when the number of allocated objects exceeds 256. Similar issues also exist with LMI, which cannot handle temporal corruption with copied pointers.

CuSafe. These observations motivate us to design a new GPU sanitizer – 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

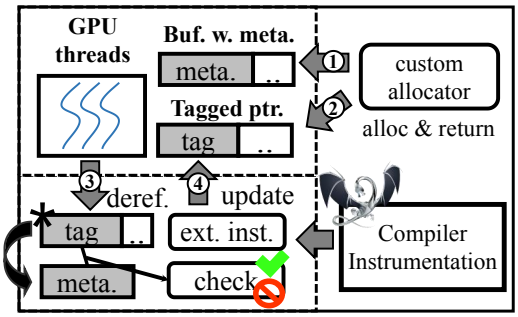

Figure 4: Overview of CuSafe.

As described in Sec. 3, existing GPU sanitizers face the problems of high overhead, deployment challenges, and insufficient coverage. CuSafe instead 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. 4, 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 shadow memory scheme, which incurs multiple memory transactions for each thread, as they each access a different shadow 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 comprehensively 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.

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. 5, 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 be zero[2]. 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 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).

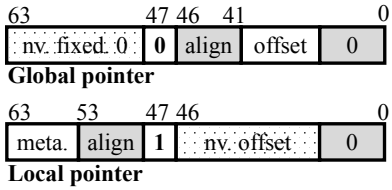

Figure 5: CUSAFE pointer tagging format.

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 [8, 29, 32, 41]. To address this issue, CUSAFE stores the exact size of the buffer in its padding area for accurate bounds checking. Fig. 6 demonstrates the allocation process of a buffer in CUSAFE. As shown in Fig. 6, 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.

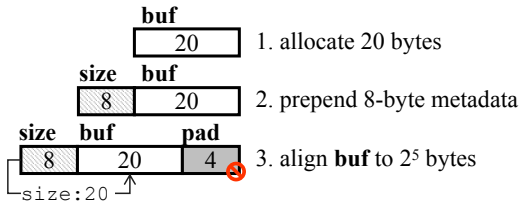

Figure 6: Metadata of exact size.

Specifically, retrieval is done by clearing the lower n bits of the pointer according to its 2^n-aligned tag. As Fig. 7 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 Fig. 7). 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

[2]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.

the exact size stored in the metadata to accurately verify if the memory access is within the bounds.

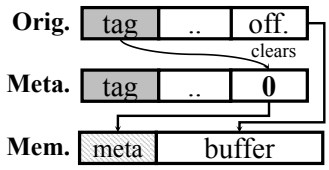

Figure 7: Retrieving exact size from pointer.

Advantage of in-band metadata. Compared to other schemes that store the metadata separately (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. 8 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.

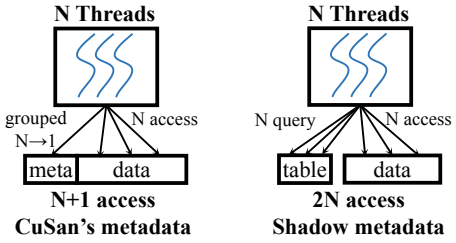

Figure 8: Comparison of metadata scheme.

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 [6, 22, 23, 43] in frameworks like PyTorch [40] and TensorFlow [1] originate from the miscalculation of pointers (e.g., integer overflow) and can inadvertently corrupt the tagging bits, disabling the sanitizer.

Previous solutions address this issue in different ways. 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. 9, 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.

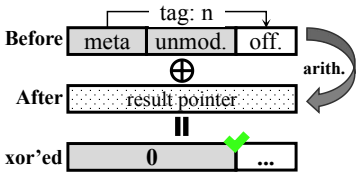

Figure 9: Validation of pointer arithmetics.

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. This delays the error until a dereference occurs, thus avoiding false positives.

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 then 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 corruptions like UAF. As shown in Fig. 10, 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.

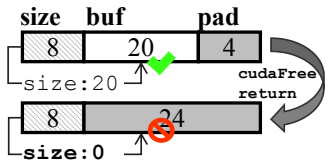

Figure 10: Metadata invalidation.

While the invalidation mechanism described above prevents most temporal corruption, it is vulnerable to metadata confusion. For example, Fig. 11 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.

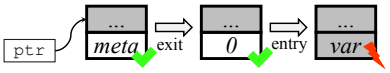

Figure 11: Metadata confusion.

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. 12 demonstrates how CUSAFE uses stack epoch to validate local pointers upon dereference. In the code snippet of Fig. 12, 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. 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.

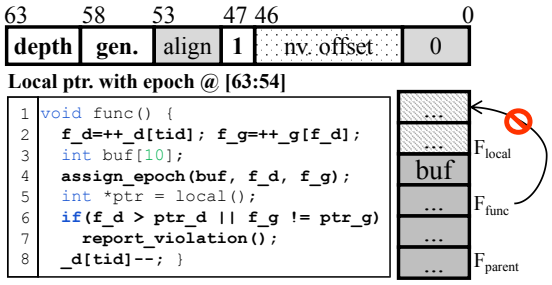

Figure 12: Stack epoch tracking.

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 own 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 $\times$ #SMs. For existing NVIDIA GPUs, the maximum value of threads/SM is 2048, and the maximum number of SMs is 132 [33]. 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.

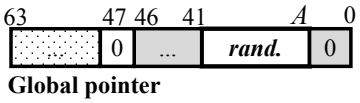

Figure 13: VA randomization.

As shown in Fig. 13, 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 the bits [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 [45], 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 [45]. 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 memory 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. 14 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 14: Optimizations adopted by CUSAFE.

Recurring checks. As shown in Fig. 14(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. 14(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. 14(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. 14(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.

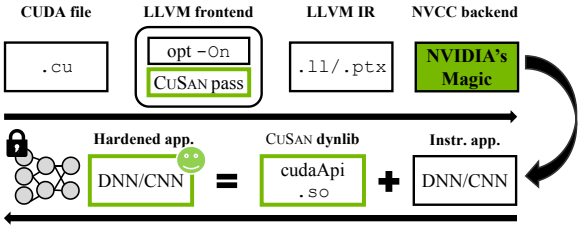

Figure 15: CuSafe-enhanced CUDA pipeline.

Fig. 15 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 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

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 adjacent and non-adjacent 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 sanitizers. Our benchmarks have been released at [3] to support future research. Table 3 summarizes the evaluation results of existing GPU sanitizers.

Table 3: Security evaluation of CuSafe.

Type		#	compute sanitizer	GPU Shield[1]	cuCatch[1]	LMI[1]	CuSafe
Spatial	Global	4	2	2	4	2	**4**
	Local	8	0	6	5	6	**8**
	Shared	3	1	0	1	1	**3**
Coverage		/	20%	53.3%	66.7%	60%	**100%**
Temporal	UAF	8	1	N.A.[2]	5	6	**8**
	UAS	4	3	N.A.	4	0	**4**
	IF	2	2	N.A.	2	2	**2**
	DF	4	4	N.A.	4	4	**4**
Coverage		/	55.6%	/	83.3%	66.7%	**100%**

UAF: Use-After-Free, UAS: Use-After-Scope, 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.

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-adjacent memory overflows and has limited coverage for the adjacent 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[3]. 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 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 sanitizers.

Temporal corruption. As for temporal memory corruption, compute-sanitizer detects 10 out of 18 cases; it misses all delayed UAF bugs and one immediate UAS (i.e., dereferencing a local pointer immediately after the function returns). 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

[3] Unlike LMI, which relies on 2^n-aligned size for detection, GPUShield adopts it to improve the performance.

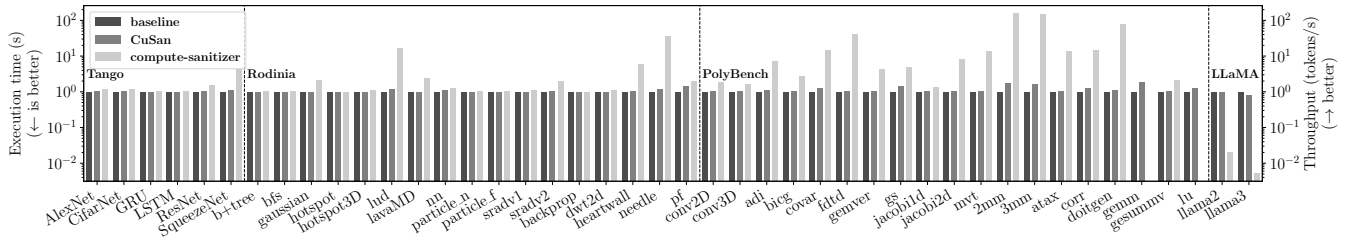

Figure 16: Execution slowdown of CuSafe on different benchmarks. Note that for LLM benchmarks, we measure the throughput (tokens/s), which is higher is better; we measure other cases' execution time (seconds), which is lower is better. PolyBench's `lu` and `gemm` cannot finish within 60 minutes under compute-sanitizer, we omit them in the figure.

(>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 two UAFs and all UASs 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 incorporates both established GPU benchmark suites, such as Rodinia, Tango, and PolyBench-GPU [13, 24, 30], and two popular large language models (LLMs) to assess performance on real-world workloads: LLaMA2 (7B) and LLaMA3 (8B) [17, 46]. 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. Fig. 16 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. The software sanitizer 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[4]. 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 for GPU sanitizers, 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

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

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 of different GPU sanitizers.

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 sanitizers. 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 of GPU sanitizers.

	cuCatch	LMI	CUSAFE
Max. Rel. Overhead	5,121.5× `jacobi1d`	2× `needle`	528× `jacobi1d`
Max. Abs. Overhead	3.79 GiB `llama2`	6.89 GiB `llama2`	16.51 MiB `llama3`
Average Overhead	0.73 GiB 15%	1.10 GiB 23%	16.5 MiB 0.3%

Table 6 reports the maximum and average memory overhead for each sanitizer. 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×, respectively. 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` [37] 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. 17, 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.

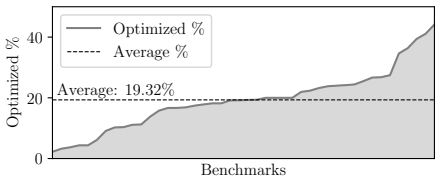

Figure 17: Percentage of checks optimized by CuSafe.

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

In this section, we discuss sanitizers for CPU programs. We already reviewed highly relevant GPU sanitizers in Sec. 2.3. Extensive efforts have been devoted to developing efficient and effective sanitizers for CPU memory safety. One of the most widely used sanitizers is ASan [42]. ASan allocates *shadow memory* to track the memory status and instruments the program to prevent invalid memory access. A major drawback of this approach is a performance overhead of approximately 100% [42]. To reduce this overhead, various improvements have been proposed. For example, ASan-- [48] reduces ASan's overhead by removing redundant checks. Specifically, ASan-- leverages static analysis to identify and remove the redundant checks of ASan, achieving a performance improvement of 30% to 60%. RSan [14] introduces an efficient redzone scheme that can check ranges of memory to improve performance. Since ASan's overhead primarily stems from instrumentation, researchers have also explored sanitizer designs that rely on hardware features [7, 15, 16, 28]. MTSan [7] utilizes emerging memory tagging features in Arm to track memory status for closed-source software. PACSan [28] employs a similar hardware feature named pointer authentication in Arm to tag pointers. Beyond memory tagging, hardware acceleration has also been applied to common sanitizer operations. For instance, Gorter et al. [16] proposed FloatZone, which repurposes the floating-point unit to accelerate bound checking. DangZero [15] achieves efficient UAF detection by delegating the memory management to the sanitizer.

8 Discussion

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 [36] compared to its host-side counterpart (i.e., `cudaMalloc`), and is therefore seldom used.

Closed-source modules. Since CuSafe is designed as a 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` [9]. However, these libraries are highly optimized and extensively tested, making the likelihood of memory bugs low. In addition, open-source alternatives exist, such as `MAGMA` and `ArrayFire` [10, 47].

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 [2], 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.

9 Conclusion

CuSafe is the first practical GPU memory sanitizer that comprehensively 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

This paper introduces a software-based memory safety solution for commodity NVIDIA GPUs, aiming to improve system reliability and security without causing harm. All experiments were conducted locally using synthetic data, eliminating any risks to personal privacy or public services. Our approach involves no attack vectors against existing CUDA programs or NVIDIA GPUs, ensuring no risk to the CUDA ecosystem or NVIDIA's interests. We carefully conducted fair and reproducible comparisons between CuSafe and competing systems by using identical software and hardware configurations, maintaining equalized baselines, and reporting results based on the average of multiple runs.

These practices align with ethical principles of beneficence, respect for persons, and justice, ensuring that our research advances knowledge while safeguarding privacy, fairness, and the integrity of existing systems.

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 an anonymous site [3]. We will continue to maintain the source code for future research after the paper is accepted.

References

[1] 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.

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

[3] Anonymous. Cusan website, 2025. URL https://sites.google.com/view/safe-gpu/.

[4] 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.

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

[6] 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.

[7] 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.

[8] Yanzuo Chen, Zhibo Liu, Yuanyuan Yuan, Sihang Hu, Tianxiang Li, and Shuai Wang. Compiled models, built-in exploits: Uncovering pervasive bit-flip attack surfaces in DNN executables. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025. URL https://www.ndss-symposium.org/ndss-paper/compiled-models-built-in-exploits-uncovering-pervasive-bit-flip-attack-surfaces-in-dnn-executables/.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] Github. gpu-rodinia, 2024. URL https://github.com/yuhc/gpu-rodinia.

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

[15] 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.

[16] 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.

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

[18] 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.

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

[20] 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.

[21] Intel. Enable intel lam in linux, 2025. URL https://lpc.events/event/11/contributions/1010/attachments/875/1679/LAM-LPC-2021.pdf.

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

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

[24] 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.

[25] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization (CGO)*, page 75–88, San Jose, CA, USA, Mar 2004.

[26] 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.

[27] 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.

[28] 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.

[29] Yuzhe Ma, Kwang-Sung Jun, Lihong Li, and Xiaojin Zhu. Data poisoning attacks in contextual bandits. In *Decision and Game Theory for Security: 9th International Conference, GameSec 2018, Seattle, WA, USA, October 29–31, 2018, Proceedings*, page 186–204, Berlin, Heidelberg, 2018. Springer-Verlag. ISBN 978-3-030-01553-4. doi: 10.1007/978-3-030-01554-1_11. URL https://doi.org/10.1007/978-3-030-01554-1_11.

[30] 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.

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

[32] Sedir Mohammed, Lukas Budach, Moritz Feuerpfeil, Nina Ihde, Andrea Nathansen, Nele Noack, Hendrik Patzlaff, Felix Naumann, and Hazar Harmouch. The effects of data quality on machine learning performance on tabular data. *Information Systems*, 132:102549, July 2025. ISSN 0306-4379. doi: 10.1016/j.is.2025.102549. URL http://dx.doi.org/10.1016/j.is.2025.102549.

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

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

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

[36] 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.

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

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

[39] 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.

[40] 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.

[41] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. Bit-flip attack: Crushing neural network withprogressive bit search. *CoRR*, abs/1903.12269, 2019. URL http://arxiv.org/abs/1903.12269.

[42] 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.

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

[44] 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.

[45] 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.

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

[47] 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.

[48] 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.