Let-Me-In: (Still) Employing In-pointer Bounds Metadata for Fine-grained GPU Memory Safety

Jaewon Lee[†], Euijun Chung[†], Saurabh Singh[†], Seonjin Na[†], Yonghae Kim[‡], Jaekyu Lee[§], and Hyesoon Kim[†]
Georgia Institute of Technology[†], Arm[‡], Intel[§]
{jaewon.lee, euijun, saurabh.s, seonjin.na}@gatech.edu, yonghae.kim@intel.com, jaekyu.lee@arm.com,
hyesoon@cc.gatech.edu

***Abstract*—The importance of ensuring the robustness of GPU systems has grown significantly, especially as GPUs have become vital in critical decision-making systems such as autonomous driving and medical diagnostics. However, GPU programming languages, primarily based on C/C++, inherit memory vulnerabilities that threaten the robustness of GPU applications. The heterogeneous GPU memory hierarchy makes it more difficult to find effective universal solutions. While several studies have proposed advanced GPU memory safety mechanisms, they still grapple with significant challenges, including substantial metadata storage and access overhead, elevated hardware implementation costs, and limited security coverage, particularly regarding fine-grained memory safety.**

We address this issue with Let-Me-In (LMI), a fine-grained memory safety mechanism specifically designed for GPUs. LMI features an efficient hardware bounds-checking mechanism that ensures negligible impact on performance and hardware costs, even in scenarios where thousands of concurrent threads perform memory operations across buffers in heap and local memory. This is achieved by aligning memory allocation to powers of two and performing static analysis to identify and mark pointer arithmetic instructions. This approach also enables storing metadata inside the unused upper bits of pointers, which are shrinking due to the expansion of the virtual memory address space. The unique characteristics of GPU programs make this approach feasible, unlike in CPU programs, where the inherent complexity of programs poses challenges. Our evaluation shows that LMI incurs only negligible hardware and performance overhead, making it a practical and efficient solution for enhancing GPU memory safety.

I. Introduction

Graphics Processing Units (GPUs) have revolutionized the computing landscape, enabling the processing of large datasets at unprecedented speeds by leveraging their massively parallel architecture. With the recent surge in deep learning applications, GPUs are increasingly being employed in life-critical decision-making processes, such as autonomous driving [25] and medical diagnostics [18]. GPU applications are commonly developed using GPU programming languages such as CUDA [45], HIP [1], OpenCL [30], and OpenACC [49], which adhere to the C/C++ standard to maintain a familiar development environment. However, these languages inherit one of C/C++'s most critical issues: memory vulnerabilities. Previous studies [21], [51] have shown that exploiting such vulnerabilities can compromise GPU applications, potentially disrupting the functionality of deep learning systems running on cloud servers.

Memory safety has been a critical issue in the CPU domain for over half a century [6]. These vulnerabilities can enable attacks that escalate privileges to superuser levels, expose private information, or even hijack entire systems [9]. To address these threats, extensive research has focused on diverse solutions, including memory-safe programming languages [28], [37], analysis tools [43], [54], and compiler-based memory safety mechanisms [4], [13], [39].

Applying CPU-based memory safety solutions to GPUs, however, is challenging due to the unique GPU architecture. First, unlike GPU global memory, heap and local memory are individually allocated and accessed by thousands of concurrently running threads, which imposes significant overhead for buffer management. Second, GPUs have a distinct memory hierarchy—including global, heap, shared, and local memory—each requiring specialized allocation mechanisms.

Recent research has introduced various GPU memory safety mechanisms, but these solutions often provide limited protection or incur substantial performance and hardware overhead. For instance, GPUShield [35] utilizes unused upper bits in pointers to store tags for buffers passed through kernel arguments. However, adopting 5-level memory paging [24] reduces the availability of these unused bits, and GPUShield's treatment of heap and local memory (stack) as a single entity leaves them vulnerable to heap or stack overflow attacks. IMT [57] employs ECC values as memory tags to avoid additional memory overhead, but this feature is unavailable on most consumer-level GPUs, which typically lack default ECC support. Similarly, cuCatch [58] reduces bounds metadata access overhead through compiler optimizations but still imposes a 20% performance overhead, posing a significant obstacle to practical adoption in real-world applications.

To address these challenges, we propose LMI, a pragmatic hardware bounds-checking mechanism that embeds in-pointer bounds metadata by leveraging 2^n-aligned pointers. Whereas pointer alignment is a popular technique [5], [22], [36] for efficient memory safety mechanisms, its adoption in real systems has been limited due to substantial performance degradation, hardware overhead, and memory overhead caused by fragmentation. In GPU programming models, however, the impact of pointer alignment becomes negligible because 1) GPU programming predominantly relies on index-based memory accesses, rather than complex pointer chasing, to harness massive thread-level parallelism, and 2) GPU memory buffers are naturally aligned to 2^n sizes, enabling efficient warp-level execution. Building on the Baggy Bounds Checking method [5], LMI significantly enhances performance and coverage by integrating tailored hardware and software support. LMI provides a hardware bounds-checking unit tailored for 2^n-aligned pointers working with hint bits in instructions. Compiler support is also essential to 1) inject code to allocate aligned memory in the local stack memory and 2) annotate instructions that necessitate bounds checks at runtime. The runtime memory allocator is also redesigned to oversee aligned buffer allocation and deallocation.

The contributions of our paper are as follows:

- LMI efficiently supports fine-grained GPU memory safety by leveraging in-pointer size information. This is enabled by power-of-two-aligned memory allocation.
- LMI examines the aligned-pointer memory safety mechanism and demonstrates that the memory fragmentation problem is negligible in GPUs due to the unique characteristics of GPU programming models.
- To eliminate the need to store bounds information in memory, LMI validates pointers during every pointer operation. Following a Correct-by-Construction approach guarantees pointer validity throughout the entire lifecycle, from pointer generation to pointer destruction.

II. Background & Related Work

A. Heterogeneous GPU Memory System

GPUs are designed as highly parallel processing architectures to handle computationally intensive tasks efficiently. To achieve optimal performance, they utilize a multi-level memory hierarchy comprising global memory, shared memory, and local memory. (Registers, constant memory, texture memory, and surface memory are excluded from this discussion as they are irrelevant attack targets.) This hierarchical structure allows GPUs to access frequently used data with minimal latency while optimizing memory usage.

Global memory, the largest and primary memory resource, is shared across threads and kernels. Global memory has high access latency, making it less suitable for frequent access. Shared memory, accessible to threads in the same thread block, offers latency comparable to L1 cache, enabling rapid data retrieval and intra-block communication. Local memory, used for thread-specific variables and buffers, resides in DRAM alongside global memory but is separated at the thread level.

Since global memory can be accessed by any thread in the GPU, buffers in global memory are particularly vulnerable to exploitation, making global memory the primary focus of previous memory safety mechanisms [35], [57], [58]. In contrast, heap and local memory (stack) are allocated separately for each thread, meaning a single `malloc()` or `alloc()` call in the kernel can simultaneously create buffers for all active threads. Interestingly, while threads share the same local memory address space for their variables and buffers, GPU address translation maps these addresses to distinct physical memory locations, ensuring isolation between thread-specific memory regions. However, recent research [21] demonstrates that this address translation mechanism can be bypassed, enabling unauthorized access to local memory regions by other threads. This underscores the urgent need for robust local memory protection mechanisms.

B. Memory Safety

Memory vulnerabilities pose a significant threat to system security, potentially compromising system integrity. Exploits can circumvent a program's intended control flow, enabling remote arbitrary code execution [9], [52] or privilege escalation. These vulnerabilities often arise from improper management of memory operations, including allocation, access, and deallocation. We categorize prior approaches to memory safety based on their methods and the specific stages of the pointer life cycle they address, as summarized in Table I.

TABLE I: Pointer life cycle and activated mechanisms.

Pointer Life Cycle	**Method/Technique**
Pointer Generation	All
Pointer Update	Pointer Aligning [5], Pointer Tracking [55]
Pointer Dereferencing	Pointer Tagging [32], [48], [58], [64], Memory Tagging [7], [57], Tripwires [53], [56]
Pointer Destruction	Canary [14], [17]

Table II provides a comparison of various software- and hardware-based memory safety solutions. Software-based solutions are relatively inexpensive to implement but often incur significant performance overhead, making them unsuitable for real-time applications. Hardware-based solutions, in contrast, offer minimal performance impact and operate independently of software intervention. However, they typically require additional components, such as specialized logic or dedicated memory (e.g., caches or tightly coupled memory) to store security-related metadata. This added complexity increases hardware costs and verification time. LMI is a hardware-based solution designed to minimize implementation complexity by leveraging the unique characteristics of GPU programming.

C. CPU Memory Safety Solutions

Canary and Tripwires: These mechanisms allocate special memory regions around target objects to protect against malicious accesses. The canary method verifies the integrity of these regions, referred to as canaries, either at the end of program execution or periodically. In contrast, Tripwire detects unauthorized accesses in real time. While these methods introduce minimal performance overhead, their security coverage is limited, as they cannot detect non-adjacent accesses that bypass Canary/Tripwire regions. Califorms [53] achieves intra-object protection with minimal memory overhead by utilizing padding bytes for alignment. REST [56] further improves protection by integrating random tokens and cache-embedded token detectors to identify unauthorized accesses.

Bounds-checking: These mechanisms involve storing bounds information for allocated buffers and performing validity checks upon accesses. Intel MPX [48] associates bounds information with pointers and enforces bounds-checking during

TABLE II: Comparison of the security coverage of LMI with previous GPU memory safety mechanisms. Detailed test cases are explained in Table III.

Name	**Target**	**Base**	**Mechanism**	**Spatial Safety**				**Temporal Safety**	**Metadata Access**	**Perf. Overhead**
				Global	**Shared**	**Stack**	**Heap**			
Baggy Bounds [5]	CPU	SW	Pointer Aligning			●	●	○	No[a]	72%[b]
No-Fat [22]	CPU	HW	Pointer Aligning			○	●	◐	Yes	8%
C3 [36]	CPU	HW	Pointer Encryption			○	●	●	No	0.01%[c]
clArmor [17]	GPU	SW	Canary	◐	○	○	○	○	No	x1.48
GMOD [14]	GPU	SW	Canary	◐	○	○	○	○	No	x3.06
Compute Sanitizer [44]	GPU	SW	Tripwires	◐	◐	◐	◐	◐	Yes	x72.29
GPUShield [35]	GPU	HW	Pointer Tagging	●	○[d]	◐	◐	○	Yes	0.8%
cuCatch [58]	GPU	SW	Pointer Tagging	●	◐	●	○	◐	Yes	19%
IMT [57]	GPU	HW	Memory Tagging	●	○[d]	○	○	◐	Yes	2.69%
LMI	GPU	HW	Pointer Aligning	●	●	●	●	◐[e]	**No**	**0.2%**

[a] Only for 64-bit; 32-bit versions require memory accesses. [b] SPEC CPU2000. [c] Large hardware overhead due to a key stream generator per processor.
[d] Not explicitly mentioned in the paper. [e] Further temporal safety can be provided by pointer tracking (§XII-C).

memory accesses. However, it incurs significant performance penalties due to the propagation of bounds and multi-level bounds addressing.

Several studies [12], [38], [63]–[65] adopt hardware-assisted fat pointers, whereby pointers hold bounds and permission metadata. CHERI's use of 128-bit fat pointers requires changes in memory usage and binary structure. This approach exacerbates register pressure in GPUs, which rely on large register files to support thousands of parallel threads. Alternatively, HardBound [12] and Watchdog [38] store pointer metadata in shadow memory that mirrors allocated memory pages and stores metadata in locations mapped to pointer addresses.

CHEx86 [55] provides runtime memory safety with full binary compatibility. CHEx86 generates bounds-checking μops from macro-ops without modifying any source code. The bounds metadata is acquired from the OS symbol table. Furthermore, it tracks all operand registers to identify their buffer IDs and validates the pointers with buffer metadata when the registers are used for load/store instructions. This mechanism relies on rulebook-based register propagation, which requires regular updates whenever the ISA is changed.

Memory Tagging: Memory tagging mechanisms embed tags into pointers and perform tag validations whenever a pointer is dereferenced. Memory access is granted only if the pointer's tag matches the tag of the memory region being accessed. ARM's Memory Tagging Extension (MTE) [7] embeds a 4-bit tag within a pointer and links it to the corresponding memory region. While memory tagging approaches [7], [50] incur minimal performance overhead, the small tag size restricts security coverage because of a high likelihood of false positives. Increasing the number of tag bits can enhance security but demands extensions to relevant hardware components, such as the tag cache, tag checker, and cache metadata.

Pointer Tagging: These mechanisms [23], [32], [67] embed a pointer tag within the unused upper bits of pointers, enabling the retrieval of object metadata from memory using the tag. AOS [32] utilizes Arm Pointer Authentication Code (PAC) as unique keys to locate bounds information in memory, thereby eliminating the need for register extension and shadow memory. These keys index a bounds table in memory, and microarchitectural enhancements are introduced to obviate the need for explicit bounds-checking instructions. In-fat pointer [67] stores object metadata alongside in-memory-type metadata, ensuring protection at the subobject granularity.

Pointer Alignment: Baggy Bounds Checking [5] allocates 2^n-aligned memory buffers. This arrangement encodes memory region boundaries directly within address pointers, enabling simple pointer arithmetic to determine valid memory ranges. However, this approach leads to memory fragmentation in CPU applications, resulting in significant unused memory. NoFAT [22] leverages aligned memory allocation up to a 4KB page size. It employs a Memory Allocation Size Table (MAST) to track allocation sizes, which generates additional memory traffic for metadata access. C3 [36] proposes a stateless memory-safety technique that utilizes a radix-aligned pointer encoding scheme. C3 eliminates the need to access bounds metadata, but implementing a fully secure key stream generator poses hardware cost and verification challenges.

D. GPU Memory Safety Solutions

Earlier software-based GPU solutions [14], [17], [44] suffer from substantial performance overhead or limited security coverage. clARMOR [17] and GMOD [14] leverage lightweight canary mechanisms, detecting buffer overflows only when they access adjacent memory areas. NVIDIA's Compute Sanitizer detects memory safety violations by leveraging a tripwire mechanism. However, since Compute Sanitizer relies on binary instrumentation, it introduces significant performance overhead. Comparing previous proposals, cuCatch [58] offers better security coverage with lower performance overhead. It utilizes shadow-tagged memory to track memory objects and performs optimized bounds checking at the compiler level. However, it still incurs a 19% performance overhead. Additionally, cuCatch does not protect kernel heap memory.

Recently, hardware-based mechanisms have been proposed based on pointer tagging [35] or memory tagging [57] to overcome the performance overhead of software-only mechanisms. GPUShield [35] utilizes a hardware-based pointer tagging mechanism on GPU systems. It achieves negligible performance overhead due to its hardware bounds-checking approach. However, it offers limited fine-grained memory protection for buffers in the heap and local (stack) memory, and it does not support temporal safety. As observed in Figure 1, while some workloads, such as *bert* and *decoding*, primarily access global memory, others, like *lud_cuda* and *needle*, rely heavily on shared memory, accounting for over 80% of total memory accesses. Since GPUShield provides memory protection only for the global memory region, these profiling results underscore its limited coverage. IMT [57] addresses these limitations by providing fine-grained protection without incurring additional storage or memory bandwidth overhead, leveraging ECC redundancy. However, this approach is limited to high-end GPUs, as most commercially available GPUs lack ECC support.

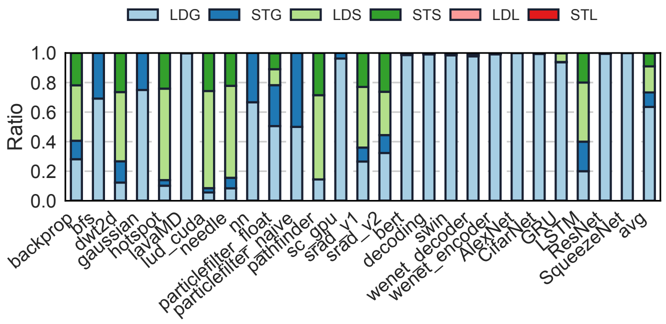

Fig. 1: Ratio of memory instructions per region in GPU workloads. The target memory region for each load/store instruction is categorized as follows: LDG/STG for global memory, LDS/STS for shared memory, and LDL/STL for local memory.

III. Threat Model

Our threat model aligns with prior research on GPU memory security. We assume that an attacker is aware of vulnerabilities in the target program and leverages malicious inputs to gain unrestricted memory read/write access. This access can be exploited to steal, corrupt, or modify user data, disrupt control flow, or produce incorrect outputs within the same cloud server [51]. We further assume that the attacker cannot alter the binary image, as doing so would already provide the elevated privileges typically sought through buffer overflow attacks. GPU side-channel attacks, such as covert channels [15], [16] and power side-channels [26], [41], are beyond the scope of this work.

IV. LMI Architecture

Figure 2 presents an overview of LMI. To achieve scalable and stateless fine-grained memory safety, LMI performs bounds checking only with size information embedded in the pointer's upper bits by utilizing 2^n-aligned memory allocation.

A. Benefits of Power-of-Two Aligned Memory

The bounds-checking mechanism typically relies on three key components: 1) the *base address* of the buffer, 2) the *buffer size*, and 3) the *buffer ID* used to locate the corresponding buffer metadata. We observed that 2^n-aligned memory allocation enables the bounds-checking operation to work using only size information for the following reasons.

1) Easy Base Address Retrieval: With 2^n-aligned memory allocation, the base address is always aligned to the buffer size. This alignment ensures that, regardless of how many pointer arithmetic operations have been performed, the base address of the buffer referred to by the pointer can be calculated using the pointer value and the size information. For example, if a pointer has the value `0x12345678` and the buffer size is 256B, the base address of the buffer is `0x12345600`. Even if the pointer value is updated to `0x123456FF`, the base address remains consistently `0x12345600`.

2) Enabling Program-Wide Pointer Verification: When a pointer value is updated to an out-of-bounds address, such as `0x12345700` in the previous example, the base address retrieved from the pointer becomes incorrect. To address this issue, LMI adopts the *Correct-by-Construction* principle. This principle, inspired by the concept of designing systems or software for inherent correctness and defect prevention, emphasizes ensuring correctness through design rather than relying solely on post-development testing and debugging. LMI ensures pointer validity during pointer generation and performs verification on every pointer arithmetic instruction (e.g., `IADD`, `IMOV`). By doing so, it establishes that pointers remain structurally correct for subsequent accesses. The use of 2^n-aligned memory allocation significantly simplifies bounds-checking operations, enabling program-wide pointer verification with minimal performance overhead. In Section VII, we will discuss how LMI efficiently implements this approach by leveraging 2^n-aligned pointers.

3) Compact Size Encoding: To eliminate the need for accessing in-memory metadata, LMI leverages unused upper bits in pointers to store size information. However, these upper bits have become limited due to architectural changes in CPUs and GPUs (§IV-B2). By employing 2^n-aligned memory allocation, buffer sizes can be represented solely by their extent values in a power-of-two exponential form. To save bits for size representation, we use an encoding scheme where the minimum alignment size ($256 = 2^8$) is encoded as 1, and the maximum size of 256 GiB (2^{38}) is encoded as the extent value 31. Additionally, the buffer size is constrained by system memory limits or device-specific restrictions, such as those set using `cudaDeviceSetLimit()`. These constraints prevent users from assigning unrealistically large buffer sizes (e.g., 128 GiB or 256 GiB) to extent values. Extent values that exceed practical buffer sizes can be repurposed to encode debugging information, such as error types (e.g., spatial or temporal safety violations).

4) No Buffer ID: Bounds-checking can now be performed without accessing memory, thanks to the use of 2^n-aligned pointers. Therefore, buffer IDs, which were previously used as keys to search metadata in memory, are no longer required.

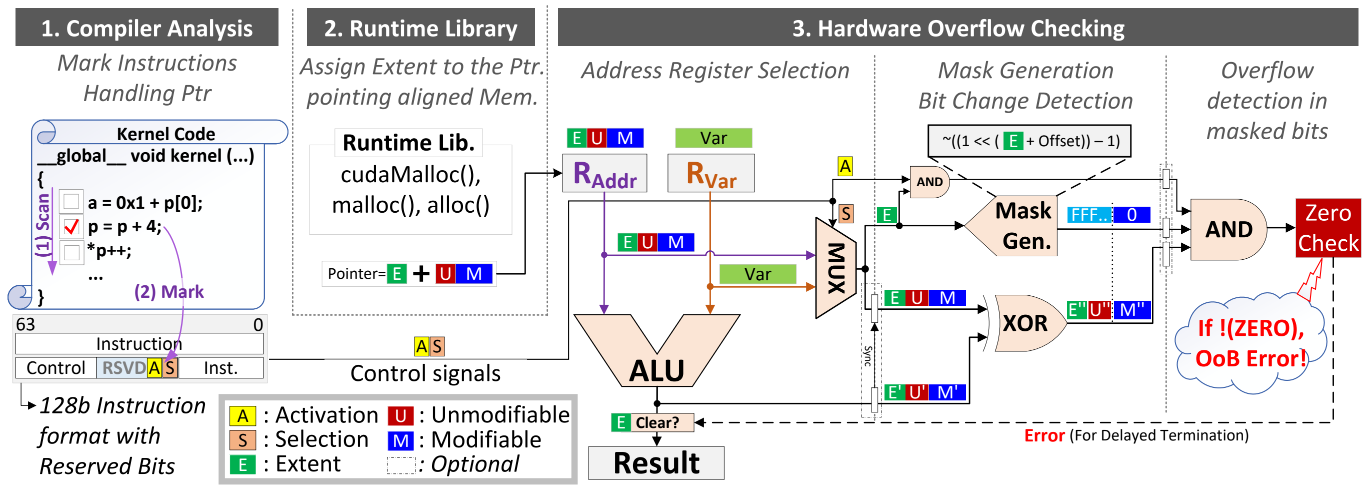

Fig. 2: LMI Architecture Overview.

B. Design Decisions Behind LMI

The main objective of LMI is to provide practical and fine-grained GPU memory safety in scenarios where numerous threads operate in parallel, continuously allocating and deallocating substantial volumes of memory. Achieving this objective requires careful consideration of the following aspects:

1) Performance Impact of Bounds Metadata Access: To support fine-grained memory safety, minimizing or eliminating memory accesses for retrieving bounds information is critical for performance. Unlike host-side memory allocation functions such as `cudaMalloc()`, device-side `malloc()` or `alloc()` are invoked concurrently by numerous threads. This concurrency can result in significant additional memory accesses for bounds information retrieval. Prior Roofline analyses of GPU applications [61] highlight that memory bandwidth often serves as the primary performance bottleneck. Consequently, these extra memory accesses for bounds metadata retrieval can lead to substantial performance slowdowns.

2) Squeezing Unused Upper Bits in Pointers: Previous memory and pointer tagging solutions [7], [32], [35], [36] have utilized unused upper bits in pointers to store memory object identifiers. This approach offers a convenient mechanism to propagate buffer IDs during pointer operations without additional management overhead. However, the availability of these upper bits has become constrained due to recent updates in CPUs and GPUs, including:

- The introduction of the `.unified` attribute [46] in the Hopper architecture, which enables a unified virtual memory address space. This feature allows the host and other devices in the system to directly reference variables.
- The expansion of virtual memory address space from 48 bits to 57 bits [2], [24] in systems where CPUs share a unified virtual memory address map with GPUs.
- The growing demand for large memory in GPU systems, driven by memory-intensive applications such as Large Language Models (LLMs). The shift towards multi-GPU setups for enhanced performance [59] further amplifies the memory requirements.

3) Increasing Hardware Verification Cost: In modern SoC development, hardware verification accounts for a substantial portion of the overall cost, especially at advanced process nodes [19]. The integration of even small cache logic components requires extensive verification of system IPs, including the network-on-chip (NoC) and memory controllers, significantly increasing verification expenses. Thanks to LMI's memory-independent and per-thread architecture, LMI minimizes its impact on the system and reduces verification complexity.

C. Challenges in Adopting CPU Solutions for the GPU

1) Architectural Differences: Directly adopting CPU solutions for GPUs incurs significant hardware costs and performance overhead without considering the unique characteristics of the GPU system. The GPU's method of accessing memory buffers differs from that of CPU code. CPU code frequently stores to and loads from pointers, while the GPU accesses arrays using base addresses and indices for the following reasons:

Thread Divergence: Programs involving pointer chasing can result in significant thread divergence, where threads within a warp (or workgroup) take divergent branches, executing different instructions. This thread divergence can lead to computational serialization, undermining the efficiency of SIMT execution on GPUs.

Memory Latency Hiding: GPUs are optimized to hide memory access latency by concurrently executing computation and memory access. However, pointer-chasing programs often have unpredictable access patterns, complicating the effective overlap of memory access and computation.

2) Workload Characteristics: Pointer-chasing programs align more naturally with CPUs due to their flexible control flow capabilities, which enable efficient handling of irregular memory access patterns. GPUs, on the other hand, are better

```
__device__ uint64_t func() {
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    int* p = (int*)malloc(id * sizeof(int));
}
```

Fig. 3: Variable-size heap allocations by threads in the same warp.

suited for regular data-parallel workloads, such as matrix operations, image processing, and simulations. This mismatch makes pointer chasing less suitable for GPU architectures.

D. Limitations in Existing GPU Solutions

Heap and Stack Memory Protection: GPUShield [35] performs region-based bounds checking, treating heap or stack (local) memory as a single large memory chunk. While this approach can detect buffer overflows across the entire heap or stack, it fails to identify overflows within individual buffers in a single kernel thread. Recent studies [21] show that malicious inputs, even within the same thread, can cause buffer overflows that alter the return address, enabling Return-Oriented Programming (ROP) attacks [52]. This vulnerability serves as the foundation for the Mind Control Attack [51].

Per-Thread Bounds Checking: Figure 3 illustrates that threads within the same warp can allocate buffers of different sizes. Consequently, applying warp-level bounds checking, as GPUShield does, becomes challenging. Additionally, cuCatch does not cover `malloc()` calls inside kernel code. In Section V-B, we discuss how LMI addresses this limitation.

E. Rediscovering the Potential of Pointer Alignment for GPUs

Pointer alignment mechanisms, such as Baggy Bounds, have straightforward implementations. However, they suffer from memory fragmentation issues and limitations in supporting pointer store/restore mechanisms. Consequently, most hardware techniques rely on metadata to ensure memory safety. Despite these challenges, we observed that LMI does not incur significant memory overhead due to the characteristics of GPU programs and frameworks for the following reasons.

Aligned GPU Buffers: GPU programs typically allocate buffers aligned to 2^n bytes to enable efficient memory access and management during parallel execution. Figure 4 shows memory fragmentation across the benchmarks listed in Table V. To evaluate memory overhead due to fragmentation, we measured the peak RSS (Resident Set Size) for both the base and LMI cases, then calculated the relative increase in the LMI case. The results indicate that *hotspot* and *srad* exhibit negligible memory fragmentation, while *backprop* and *needle* experience fragmentation overheads of 85.9% and 92.9%, respectively. This is due to their buffers consisting of power-of-two-sized allocations with additional bytes for header information. Nevertheless, the geometric mean memory overhead remains relatively low at 18.73%.

Pre-existing Memory Fragmentation in Kernel `malloc()`: Similar to modern multi-threaded dynamic memory allocators [20], the CUDA kernel's `malloc()` manages memory layouts for efficient parallel memory allocation. As shown in Figure 5, each thread can allocate memory in different buffer groups, enabling multiple threads to concurrently manipulate

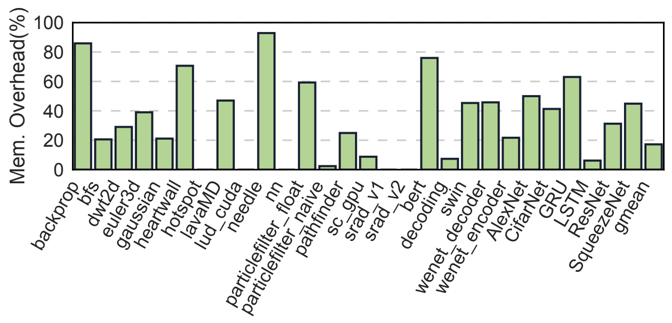

Fig. 4: Memory overhead caused by 2^n-aligned memory buffers.

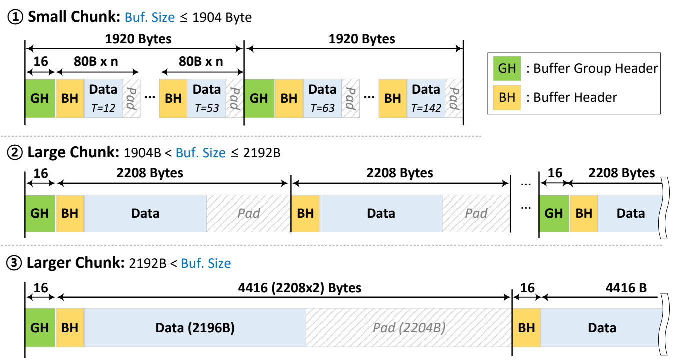

Fig. 5: Demonstration of `malloc()` in a CUDA kernel. The kernel `malloc()` allocates buffers with sizes aligned to specific numbers, such as multiples of 80 bytes or 2208 bytes, introducing memory fragmentation.

memory allocation headers. Buffers with smaller allocation sizes share a common group header, which reduces memory overhead for headers. The `malloc()` function manages buffers as multiples of a chunk unit, which varies based on the allocation size. If the requested memory size is not aligned with the chunk unit, it results in memory fragmentation of up to 50%, as seen in LMI.

V. LMI Allocation Mechanism

A. Pointer Structure for LMI

In alignment with numerous prior studies [7], [35], [36], LMI uses the upper bits of the pointer to store the size of the buffer pointed to by the pointer. As discussed in Section IV-A, a 5-bit encoding is a practical choice for expressing buffer size information. This representation is compact enough to be embedded within the upper bits of the pointer, even when accounting for potential future address bit extensions. Figure 6 illustrates how the 64-bit pointer address is divided into Extent, Unmodifiable, and Modifiable segments.

1) Extent Bits (E): The top five most significant bits (MSBs) in a 64-bit pointer store the buffer size data. These bits are referred to as Extent Bits (with offset). The buffer size can be derived from the extent bits using the following equation:

$$E = \lceil \max(\log_2(K), \log_2(S)) \rceil - \log_2(K) + 1,$$

where K is the minimum allocation size, and S is the requested memory size. To distinguish invalid pointers, which

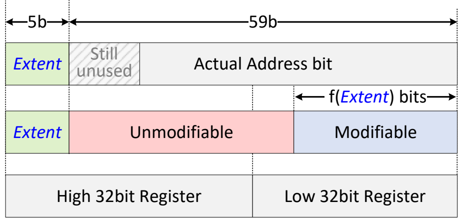

Fig. 6: 64-bit pointer structure and its mapping to two 32-bit physical registers. The extent field indicates both the size and pointer validity, while the unmodifiable bits serve as an ID for tracking pointer liveness, as detailed in Section VIII.

have an extent value of 0, we add 1 to the buffer size calculation. This enables encoding up to 31 ($2^5 - 1$) distinct 2^n-aligned buffer sizes. We select $K = 256$ to minimize the number of embedded bits, leveraging the default 256-byte GPU allocation size. This configuration allows buffer sizes ranging from 256 bytes to 256 GiB.

2) Unmodifiable Bits (UM) and Modifiable Bits (M): The address bits, other than the Extent Bits (E), represent the actual memory address used for memory operations. The Extent Bits determine the number of Modifiable Bits (M), while the remaining bits become Unmodifiable Bits (UM), which must remain unchanged throughout the pointer's lifetime. LMI performs bounds-checking by ensuring that these unmodifiable bits are not modified.

B. Memory Allocation for Each Memory Type

Given the diverse memory types within a GPU system, it is essential to formulate appropriate allocation policies tailored to each memory region. Notably, GPU memory allocation, akin to that of CPUs, provides virtually contiguous memory buffers for local, shared, and global memory. LMI's OCU operates based on virtual memory addresses, irrespective of the physical contiguity of the buffers.

Global Memory: Memory buffers in global memory are typically allocated through device-specific memory allocation functions, such as `cudaMalloc()`, and the same principles extend to deallocation functions, such as `cudaFree()`. When `cudaMalloc()` is invoked, the memory manager, which allocates memory on the device, determines the allocation size by rounding it up to the smallest 2^n size. Subsequently, it embeds the 5-bit extent value into the top 5 MSBs of the pointer for further bounds-checking operations. When `cudaFree()` is invoked, the extent bits are set to 0 to invalidate the pointer and prohibit further access. Temporal safety is discussed in detail in Section VIII.

Heap Memory: Heap memory is allocated by the `malloc()` runtime library on the device. The procedure to prepare the buffer and pointer is the same as for global memory, except that device allocator functions `malloc()` and `free()` are used instead of `cudaMalloc()` and `cudaFree()`.

Shared Memory: Shared memory is allocated during a kernel launch. The total size of shared memory is provided as a parameter during the kernel launch. Consequently, the responsibility for aligning shared memory falls upon the kernel driver.

```
__global__ void dummy(int size) {
    int buf[96]; ..
}
```

(a) Original code for stack allocation.

```
// Beginning of the function dummy
    _Z7dummy2i:
    .text._Z7dummy2i:
// Load stack pointer from constant memory
    MOV R1, c[0x0][0x28] ;
// Secure 0x60 (96 bytes) memory with a memory
operation.
    IADD3 R1, R1, -0x60, RZ;
```

(b) Compiled SASS code for stack allocation.

Fig. 7: Stack memory allocation.

Stack Memory: Unlike the allocation process for global or heap memory buffers, stack buffers are established by the compiler. Figure 7 illustrates how a buffer declared in kernel C++ code is transformed into CUDA's SASS code. Interestingly, the pointer to the top of the stack is transmitted via constant bank 0 (`c[0x0]`), which subsequently secures the buffer region by subtracting the buffer size from the stack pointer. To support stack protection within LMI, the GPU driver first identifies the aligned memory address and stores this address within the corresponding constant memory. When the compiler generates code to set up the stack, it includes instructions to subtract an amount equal to the buffer size, rounded up to the nearest power of two, from the stack's top address.

VI. LMI Compiler Analysis

A. Tracking Pointer Arithmetic Instructions

Bounds-checking is typically performed on LD/ST instructions, where it is evident that the operands are addresses. However, for integer instructions, processors cannot easily determine whether their operands are pointers without support from a software framework.

To address this limitation, we devised an LLVM [33] pass that analyzes kernel code to identify LLVM pointer arithmetic instructions and pinpoint which operands contain pointer values. This approach not only ensures accurate runtime validation but also eliminates unnecessary bounds-checking.

Figure 8 illustrates the LLVM pass responsible for identifying instructions with pointer operands. Information gathered from the LLVM IR analysis is passed as metadata to the back-end and utilized for microcode generation.

Notably, LMI restricts the storage of pointers in memory. GPU programs typically avoid this practice due to the inefficiency of pointer chasing, which stems from high memory latency. This observation is supported by an extensive analysis of SASS code from 80 applications across diverse benchmarks, including the Heterogeneous System Benchmark [11], Graph Processing [42], Deep Learning Benchmark [29], and Large Language Models [47], using the `cuobjdump` tool. This characteristic allows our system to implement an efficient bounds-checking mechanism without the overhead of tracking pointers

```
for (BasicBlock &BB : F) {
  for (Instruction &I : BB) {
// Check instructions with pointer operands.
// This information will be used during assembly code
generation.
    if (I.getNumOperands() > 0) {
      for (Use &U : I.operands()) {
        if (U->getType()->isPointerTy()) {
          Mark_on_instruction(U);
          ...
```

Fig. 8: LLVM code snippet to locate instructions with pointer operands.

stored in memory, as demonstrated in CHEx86 [55]. Memory safety for addresses stored in memory will be addressed in future work.

The compiler front-end identifies instructions with pointer operands, and the gathered information is delivered to the back-end as metadata. The back-end then generates assembly instructions with two hint bits, utilizing the reserved (RSVD) field in the instruction microcode, as shown in Figure 9.

B. GPU Instruction Microcode Format

Zhe et al. [27] reveal that NVIDIA GPUs use a 128-bit instruction format, which is larger than that of CPUs and older-generation GPUs. This expanded format includes not only opcodes and source and destination registers but also control flow information. Such a design reduces runtime scheduling overhead and simplifies the hardware scheduler's implementation. Their study also reveals that 14 bits in the instruction format, located between the control information and the instruction code, are currently unused. We confirmed that NVIDIA GPUs with Compute Capability 7.0–7.2 have 14 reserved bits, while those with Compute Capability 7.5–9.0 have 13.

This reserved space enables us to repurpose two bits in the instruction microcode for our OCU (Overflow Checking Unit). These bits serve as two hints: 1) **Activation (A)**, and 2) **Address Register Selection (S)**. The microcode with these hint bits, as shown in Figure 9, can be leveraged to determine whether the current instruction requires a bounds check (Bit A) and to identify which register holds the base pointer value (Bit S). For GPUs utilizing a 64-bit ISA, such as AMD and Intel models, new opcodes for memory ALU operations could be introduced to implement our approach. As we discuss in Section VII, this requires only a small number of instructions, such as integer arithmetic or bit-wise operations.

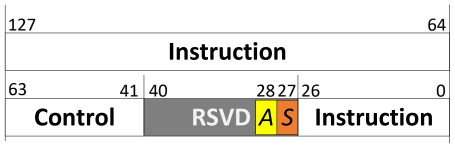

Fig. 9: CUDA's 128-bit instruction microcode format. The 28th bit ('A'), shown in the yellow box, serves as the activation bit, indicating that the instruction involves pointer handling and requires bounds checking. The 27th bit ('S'), shown in the orange box, functions as the selection bit, specifying the operand that contains the pointer address.

VII. Hardware Overflow Checking Unit

The ultimate goal of the hardware OCU is to determine whether the current ALU operation illicitly alters the upper bits beyond the range of its memory buffer. The OCU consists of a multiplexer (MUX), a mask generator, an XOR gate, an AND gate, a comparator to check for zero, and logic to clear extent bits to 0. The location of the bound-checking unit within a streaming multiprocessor (SM) core is depicted in Figure 10. Notably, OCUs are only added to integer ALUs, as FPUs are not used for pointer calculations. An Extent Checker (EC), which ensures the extent bit is zero during load/store (LD/ST) operations, is also necessary to avoid false positives. Further discussion will be presented in Section XII-A.

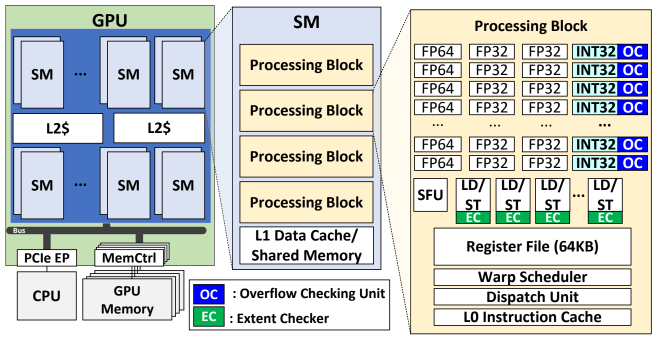

Fig. 10: LMI's OCU and EC (Extent Checker) in GPU Architecture. Bound-checking units are only required for integer ALUs.

A. Checking Hint Bits in Microcode

After a kernel launch, the instruction decoder delivers the hint bits ([27:28]) to LMI's OCU. The OCU examines the activation bit ([28]) to determine whether the current instruction involves pointer operations. If it is marked, the OCU then checks the selection bit ([27]) to identify the ALU input operand holding the memory address.

B. Mask Generation and Overflow Detection

At this stage, the OCU generates a mask derived from the extent bits. The extent bit size might not exactly match the buffer size; therefore, the mask generator accounts for the minimum memory allocation size (default: 256). Based on the extent value, the mask generator creates an address mask to capture changes in the modifiable bits (with a size less than $2^{f(\text{Extent})}$). Concurrently, an XOR operation is performed between the selected input register and the ALU output to identify which bits have been modified by the pointer arithmetic operation. The input register value is stored in a queue and synchronized with the order of incoming outputs to ensure the proper input register corresponds to the current result.

C. Overflow Detection within the Masked Bit Region

Overflow detection is straightforward. The OCU performs an AND operation between the address mask and the XOR result from the previous stage. A nonzero result indicates an overflow in bits exceeding the current buffer allocation size. If an error is detected, instead of immediately generating the error, the extent bits are cleared to zero. An error will subsequently be raised by the EC (Extent Checker) in the LSU when the extent value is zero. The rationale behind the delayed termination is discussed in Section XII-A.

VIII. Temporal Memory Safety in LMI

Temporal memory safety ensures that memory is not accessed after it has been freed or reallocated. LMI enforces temporal safety by invalidating pointers when their target memory buffers are freed or go out of scope due to a function return. Specifically, the LMI compiler pass inserts instructions to nullify a pointer's $extent$ field either immediately after a `free()` function call or just before returning to the caller function. During load and store operations, the Extent Checker (EC) evaluates the pointer's $extent$ value. If the $extent$ is zero, the EC raises an error, using a mechanism similar to that employed for detecting pointer arithmetic overflow.

This approach is particularly effective for GPU programs, which often access array-like data structures using a fixed `base_address` and variable offsets. Since the `base_address` remains unchanged and is reused until the program terminates, invalidating the $extent$ in the `base_address` pointer effectively ensures temporal safety.

However, while this temporal safety mechanism effectively protects typical GPU programs, it faces challenges when handling pointer copies. Only pointers explicitly passed to the `free()` function are invalidated, leaving duplicated pointers valid, as shown in Figure 11. This limitation is addressed in Section XII-C, where we propose an enhanced mechanism incorporating additional hardware and runtime support.

```
int* A = malloc(sizeof(int) * 4);
B = A[0]; // No error: safe access. A has a valid extent.
C = A + 1;
free(A); // Pointer A will be invalidated after this.
D = A[0]; // Error: unsafe access. A is invalid.
E = A + 1;
F = E[0]; // Error: unsafe access. E is invalid.
G = C[0]; // No error but UNSAFE access. C is not invalidated.
```

Fig. 11: LMI Temporal safety: While `free(A)` invalidates the pointer `A`, it does not invalidate a copied pointer, such as `C`.

IX. Security Evaluation

Table III provides a summary of the security coverage for each mechanism. The security benchmarks are reconstructed based on the descriptions of security evaluations in the cuCatch [58] paper, as detailed memory safety test cases in cuCatch are not publicly available.

Memory safety is broadly categorized into two types: spatial and temporal. Spatial memory safety is further subdivided by memory types: global, heap (device), local (stack), and shared memory. For local memory, test cases include single-buffer and multi-buffer scenarios, evaluating vulnerabilities within a frame, across frames, and beyond local memory boundaries. Similarly, shared memory tests cover single-buffer and multi-buffer cases, assessing vulnerabilities within shared memory, beyond shared memory, and overflows between static and dynamic shared memory. For global, heap, and local memory, the test cases address two types of out-of-bounds (OoB) errors: (1) adjacent OoB and (2) non-adjacent OoB. Additionally, intra-object OoB errors, which occur between two fields within the same structure, are tested.

Temporal memory safety is categorized into four types: (1) Use-After-Free (UAF) for global and heap memory, (2) Use-After-Scope (UAS) for local memory, (3) invalid free, and (4) double-free. Each category includes two subcategories: (a) immediate errors, where a pointer is dereferenced immediately after being freed, and (b) delayed errors, where a pointer is dereferenced after the memory allocator has potentially reassigned the memory. Furthermore, test cases for each category examine errors involving either (i) the original pointer or (ii) a copied pointer (§VIII).

TABLE III: Security Evaluation: Please note that the security evaluations are based on the descriptions provided in each paper.

	Violation	**Test**	**GMOD**	**GPUShield**	**cuCatch**	**LMI**
Spatial	Global OoB	2	1	2	2	2
	Heap OoB	3	0	1	0	3
	Local OoB	8	0	2	6	8
	Shared OoB	6	0	0	5	6
	Intra OoB	3	0	0	0	0
	Coverage		4.8%	28.6%	61.9%	85.7%
Temporal	UAF[a]	8	0	0	4	4
	UAS[b]	4	0	0	4	4
	Invalid free	2	2	2	2	2
	Double free	2	2	2	2	2
	Coverage		25%	25%	75.0%	75.0%

[a] Use-After-Free. [b] Use-After-Scope.

A. Spatial Memory Safety

We evaluate 19 OOB test cases for global, heap, local, and shared memory regions, and 3 for intra-OOB. GMOD [14], which relies on Canary, failed to detect non-adjacent access cases in global memory and does not provide protection for heap, local, and shared memory. GPUShield supports protection for global memory, but it offers only coarse-grained protection for heap and local memory. cuCatch does not protect the device heap and provides limited protection in a single buffer and within the same frame for local memory, as well as for dynamically allocated multiple buffers in shared memory. LMI provides better protection across all memory types, including heap, due to its versatile and straightforward memory protection scheme. LMI protects statically allocated shared memory objects, and a similar approach could be extended to dynamic allocations. However, we do not consider this in this work because (1) dynamic allocations in shared memory are handled implicitly by proprietary driver code, and (2) it could lead to significant fragmentation of the shared memory pool due to its comparatively small size. Instead, LMI provides coarse-grained protection for the dynamically allocated pool as a whole. Finally, like other schemes, LMI does not protect against OOB reads/writes across different fields within the same structure.

B. Temporal Memory Safety

We evaluate 16 test cases for temporal memory vulnerabilities, including UAF, UAS, invalid-free, and double-free scenarios. cuCatch demonstrates a low probability of missing delayed UAF and UAS errors. In contrast, LMI detects both immediate and delayed UAF errors by verifying the Extent bits in pointers (§VIII). However, LMI cannot detect issues involving copied pointers, a limitation addressed by cuCatch. Neither GMOD nor GPUShield provides protection for temporal safety. Protection against invalid free and double-free scenarios is provided by basic CUDA functions.

X. Evaluation Method

To evaluate the performance of LMI, we used MacSim [31], a heterogeneous cycle-level GPU simulator. CUDA traces for the simulation were generated using NVBit [60]. The baseline GPU configuration is detailed in Table IV.

TABLE IV: Configuration for NVIDIA GPU simulations.

SM Core	80 Cores @ 2GHz
Scheduler	4 warps scheduler/SM, GTO
L1 Cache	96KB, 30 cycle latency
L2 Cache	4.5 MB, 24-way, 200 cycle latency
DRAM	8GB HBM

We evaluate a range of CUDA GPU benchmarks, as summarized in Table V, including the Rodinia Benchmark [11], FasterTransformer [47], and Tango [29]. Additionally, to assess the performance impact on mission-critical applications, we examine widely used models from the Autonomous Driving (AD) domain, such as MOTR [68] for multiple object tracking, DETR [10] for object detection, BEVerse [69] for occupancy prediction, and Segformer [66] for map segmentation.

TABLE V: Benchmarks for evaluation.

Benchmark Suite	Benchmark
Rodinia	backprop, bfs, dwt2d, gaussian, hotspot, lavaMD, lud_cuda, needle, nn, particlefilter_float, particlefilter_naive, pathfinder, sc_gpu, srad_v1, srad_v2
Tango	AlexNet, CifarNet, GRU, LSTM
FasterTransformer	bert, decoding, swin, wenet_decoder, wenet_encoder
AD	BEVerse, DETR, MOTR, segformer

A. Comparison with the Hardware/Compiler Mechanisms

We first compare the performance of LMI with two existing mechanisms: the hardware-based GPUShield [35] and the compiler-based Baggy Bounds [5]. The GPUShield mechanism is implemented in MacSim according to the methodology outlined in its original paper, and its performance is evaluated using GPU benchmarks. We evaluate Baggy Bounds by injecting bounds-checking SASS instructions after each pointer operation.

B. Comparison with Software DBI Mechanisms

We also measured the performance overhead of LMI's DBI (Dynamic Binary Instrumentation) implementation using NVBit [60]. This is implemented by injecting code to call the bounds-checking function into every instruction that accesses memory regions allocated via `cudaMalloc()`. To achieve this, we first verified that each benchmark in Table V uses global memory by calling `cudaMalloc()` and passing the resulting pointers to CUDA kernels. The bounds-checking logic for LMI involved identifying whether an instruction operand was a pointer to the global memory region. For such operands, we inserted bounds-checking operations immediately after the instruction. We tracked the registers and unified registers associated with these pointers and injected bounds-checking instructions accordingly. Additionally, we extended the bounds-checking logic to instructions that access shared and local memory regions. Using NVBit's `getMemorySpace()` function, we identified relevant instructions, such as `LDS`, `LDL`, `STS`, and `STL`, and added the corresponding bounds-checking operations. For comparison, we also measured the performance of *memcheck* in Compute Sanitizer [58]. *Memcheck* is a tripwire-based runtime tool that detects out-of-bounds or misaligned memory accesses, including those involving global, local, shared, and atomic instructions. While both LMI and *memcheck* introduce overhead by injecting bounds-checking instructions around memory-related operations, LMI additionally incurs overhead for *non-*LD/ST operations, whereas *memcheck*'s impact is confined to memory LD/ST operations.

XI. Results

A. LMI Vs. Hardware/Compiler Mechanisms

Figure 12 presents the normalized execution times of Baggy Bounds Checking, GPUShield [35], and LMI relative to the baseline. LMI achieves near-zero performance overhead across all GPU benchmarks, with an average overhead of just 0.22%. While GPUShield also demonstrates competitive performance, LMI significantly reduces overhead in benchmarks such as needle and LSTM, where the overhead drops from 42.5% and 24.0% to 0.043% and 0.0016%, respectively. The primary reason for the performance difference arises from GPUShield's L1 RCache latency. In memory-intensive workloads, most of the memory requests hit the L1 Dcache. Using the same hardware configuration as GPUShield, where the L1 Dcache is larger than the L1 RCache, situations occur where L1 D$ hits and L1 R$ misses frequently for uncoalesced memory operations, especially in benchmarks such as needle and LSTM. However, since the performance of LMI is not affected by memory access patterns, LMI can achieve comparable performance to the baseline.

In contrast, Baggy Bounds Checking incurs substantially higher overhead, averaging 87%, with peak overhead reaching 503% for compute-bound applications. These results align with the findings in the original paper, which reported a 72%

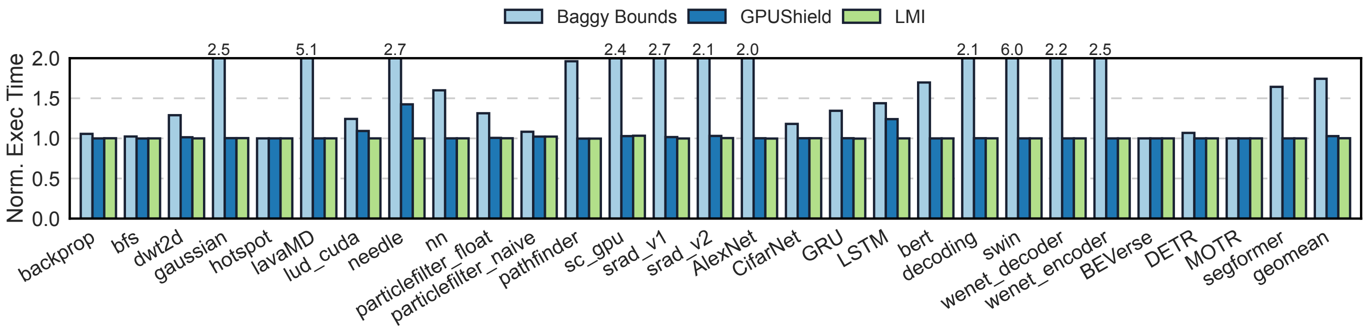

Fig. 12: Performance comparison among Baggy bounds [5] for GPU, GPUShield [35], and LMI.

average performance overhead on SPEC CPU2000 benchmarks. By leveraging hardware support, LMI offers a substantial performance advantage over Baggy Bounds Checking, a software-based approach naively adapted to GPUs, thereby demonstrating the effectiveness of its design.

B. LMI with DBI Implementation Vs. Compute-Sanitizer

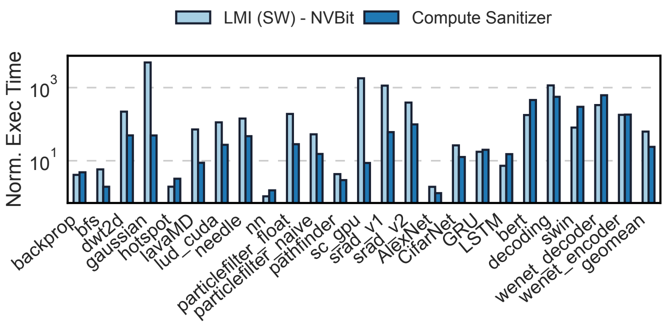

Fig. 13: Performance comparison between LMI with DBI and NVIDIA's Compute Sanitizer.

The DBI approaches for memory-bound checking show a significant overhead in most GPU benchmarks. Figure 13 illustrates the normalized execution time on a logarithmic scale.[1] LMI with DBI implementation has a geometric mean performance overhead of 72.95 times, while Memcheck's overhead is 32.98 times across all benchmarks. `memcheck` significantly outperforms LMI by DBI in cases like Gaussian, while LMI performs better in others, like swin, This performance variability is due to differences in the ratio of LMI bound checks to LD/ST instructions, with ratios of 67.14 for Gaussian and 28.13 for swin. The higher ratio for Gaussian indicates greater performance degradation with LMI, whereas the lower ratio for swin suggests less impact.

Breaking down the performance of the DBI implementation is challenging because the CUDA framework does not support the simultaneous use of the Nsight Performance Analysis tool with compute-sanitizer or NVBit. This limitation complicates the analysis of LMI's internal performance when using DBI. However, leveraging Linux's `perf` utility, we determined that the JIT compilation process in `cuda-memcheck` introduces an average overhead of 5.2%, indicating that most of the performance overhead originates from executing the inserted instructions. This finding aligns with NVIDIA's NVBit paper [60], which reports an average JIT compilation overhead of 4% and a peak overhead of 20%.

[1]AD benchmarks are excluded due to compatibility issues with NVBit and out-of-memory errors with compute-sanitizer.

C. LMI OCU Size and Latency Evaluation

We evaluate the hardware overhead of OCU using Cadence tools with the FreePDK45nm library. The critical path latency is measured at 0.63 ns, corresponding to a maximum frequency (f_{max}) of 1.587 GHz. Considering that modern GPUs operate at frequencies exceeding 3 GHz, we incorporate two register slices into LMI's logic to accommodate the higher clock rates. This modification introduces a three-cycle delay due to the bounds-checking logic for pointer operations.

As shown in Table VI, the hardware cost of LMI is significantly smaller compared to other schemes that involve additional cache hierarchy, SRAM, or extra ECC logic. Furthermore, the verification scope of LMI is minimized, as its impact is confined to the Integer ALU and LSU logic.

TABLE VI: Hardware overhead comparison based on their descriptions (T: Thread, W: Warp, SM: Streaming Multiprocessor, C: Core).

Target	Additional Logic	Gate(GE[a])	SRAM (Byte)	To Be Verified
No-Fat	Bounds checking, base computing	59,476/C	1024/C	LSU, NoC, cache
C3	Keystream generator	27,280[b]/C	0	LSU, NoC, cache
IMT	Tag logic in ECC	900/SM	0	Memctrl, ECC, cache
GPU Shield	2-Level cache, comparator	1000/W	910/W	LSU, NoC, cache
LMI	4x gate, subtract, shift, comparator	153/T	0	ALU (INT only), LSU

[a]Gate Equivalent
[b]Based on Ascon [8] implementation.

XII. Discussion

A. False Positive Avoidance: Delayed Termination

LMI detects overflows in pointer operations before actual memory accesses occur. However, as illustrated in Figure 14, it is common for programs to increment an array pointer beyond its bounds and then exit the loop or return from the function without accessing the out-of-bounds memory address. To handle this scenario, LMI implements delayed program termination, ensuring that termination occurs only when an out-of-bounds pointer is used for actual memory access. This behavior is controlled by the EC in the LSU, which checks the extent bits to confirm they are non-zero before permitting memory access.

```
int * start = (int *)malloc(16*sizeof(int));
int * end  = start + len(start);
for ( int * ptr = start ; ptr < end ; ptr++ )
    * ptr ++;
```

Fig. 14: The example demonstrates a false positive scenario. In the final loop iteration, `ptr` points to an out-of-bounds memory location, specifically `start[16]`. However, since the loop exits without accessing this memory, no system error should be triggered.

B. False Negative Avoidance: No Immediate Value Assignment

The correct-by-construction rule of LMI can be violated through direct assignments of immediate values to pointers. Such assignments allow pointers to be set to unverified values with invalid extent bits. To prevent this, LMI employs static-time analysis to detect `inttoptr` and `ptrtoint` instructions in LLVM IR, generating a compiler error when these instructions are encountered. This restriction is consistent with prior studies [5], [55].

To assess the feasibility of this restriction, we compiled 57 kernel files from the Rodinia, HeteroMark, GraphBig, and Tango benchmarks using `clang++14`. None of these kernels contained `ptrtoint` or `inttoptr` instructions. Furthermore, analyzing 111 files from the CUDA samples revealed three instances of `inttoptr` casting. In all cases, the pointers involved were confined to inlined cooperative group functions (e.g., `cg::this_grid()`), which are inaccessible to users. In 46 kernel files from *FasterTransformer*, we identified one instance where a `void*` pointer was cast to a 64-bit integer and then to a float pointer. To resolve this, we modified the code to cast the pointer directly to `float**` instead of `int*` during the initial type cast. Notably, this kernel is not part of the build process for the evaluation targets.

C. Enhanced UAF Protection with Pointer Liveness Tracking

In Section VIII, we discussed the current limitations of LMI's temporal safety. In this section, we outline potential directions for enhancing temporal safety in future work. Several studies, such as DangNull [34] and CETS [40], use shadow-object memory to track the liveness of pointers and their derivatives. These methods introduce significant performance overhead due to shadow-object traversal and increased memory usage. In contrast, LMI utilizes the *UM* bits (see §V-A2) for efficient pointer liveness tracking. These *UM* bits, which remain constant throughout program execution, serve as unique identifiers for memory buffers. Fortunately, in LMI, only one buffer exists with the same *UM* due to memory allocation constraints, ensuring that the *UM* bits uniquely identify a memory buffer. By tracking the *UM* bits, LMI avoids the need to trace all derivative pointers, enabling efficient liveness tracking without excessive overhead. The proposed procedure is outlined in Algorithm 1.

Algorithm 1 Pointer Liveness Tracking Algorithm.

```
 1: function MALLOC_HOOKED( $allocSize$ )
 2:     $allocSize \leftarrow least\_PoW2(N)$
 3:     $bufPtr \leftarrow malloc(N)$
 4:     $um \leftarrow pick\_unmodifiable(allocSize, bufPtr)$
 5:     if $!pageInvalidOpt \lor (allocSize \leq pageSize/2)$ then
 6:         $register(um)$
 7:     end if
 8: end function
 9: function FREE_HOOKED( $bufPtr$ )
10:     $size \leftarrow pickSize(bufPtr)$
11:     if $!pageInvalidOpt\ \&\ (size \leq pageSize)$ then
12:         $um \leftarrow pick\_unmodifiable(size, bufPtr)$
13:         if $um \in S$ then
14:             $deregister(um)$
15:         end if
16:     else
17:         $Invalidate\_pages(size, bufPtr)$  ▷ Unmapping pages associated with the buffer.
18:     end if
19: end function
```

To further optimize memory usage for memory object tracking, we can incorporate memory map invalidation techniques inspired by previous studies [3], [62]. By leveraging 2^n-aligned memory size, we ensure that large memory allocations ($>$ pageSize/2) are assigned to dedicated memory pages. For example, a 48KB allocation is rounded up to a full 64KB page, guaranteeing that no other buffers reside on that page. This design eliminates the need to track individual pointers, as entire pages associated with deallocated buffers can be invalidated. Enabled by setting the $pageInvalidOpt$ environment variable, this optimization balances the trade-off between the overhead of memory page invalidation and the reduction in Membership Table entries.

XIII. Conclusions

The growing importance of ensuring robust GPU systems stems from their critical role in decision-making applications. However, GPU programming languages inherit memory vulnerabilities, and adapting CPU-centric solutions to GPUs' highly parallel architectures often proves inefficient. To address these challenges, LMI introduces a fine-grained memory safety mechanism specifically designed for GPUs. It incorporates an optimized hardware bounds-checking mechanism capable of handling the stress of thousands of simultaneous memory-accessing threads. By leveraging power-of-two-aligned pointers and static analysis, this approach exploits the unique characteristics of GPU programs to provide both spatial and temporal memory safety with minimal hardware overhead and negligible performance impact.

XIV. Acknowledgements

This research is based on work partially supported by NSF 2316176 and the Intelligence Advanced Research Projects Activity (IARPA) through the Advanced Graphical Intelligence Logical Computing Environment (AGILE) research program.

References

[1] Advanced Micro Devices (AMD), "Hip: C++ heterogeneous-compute interface for portability," https://github.com/ROCm-Developer-Tools/HIP, 2021.

[2] Advanced Micro Devices (AMD), "Amd epyc 9004 series architecture overview," AMDEPYC9004SeriesArchitectureOverview, 2022.

[3] S. Ainsworth and T. M. Jones, "Markus: Drop-in use-after-free prevention for low-level languages," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 578–591.

[4] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with wit," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, ser. SP '08. USA: IEEE Computer Society, 2008, p. 263–277. [Online]. Available: https://doi.org/10.1109/SP.2008.30

[5] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors," in *Proceedings of the 18th Conference on USENIX Security Symposium*, ser. SSYM'09. USA: USENIX Association, 2009, p. 51–66.

[6] J. P. Anderson *et al.*, "Computer security technology planning study," ESD-TR-73-51, Tech. Rep., 1972.

[7] Arm, "Armv8.5-a memory tagging extension," 2021. [Online]. Available: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf

[8] Ascon, "Ascon hardware implementation." [Online]. Available: https://ascon.iaik.tugraz.at/implementations.html

[9] B. Bierbaumer, J. Kirsch, T. Kittel, A. Francillon, and A. Zarras, "Smashing the stack protector for fun and profit," in *ICT Systems Security and Privacy Protection: 33rd IFIP TC 11 International Conference, SEC 2018, Held at the 24th IFIP World Computer Congress, WCC 2018, Poznan, Poland, September 18-20, 2018, Proceedings 33*. Springer, 2018, pp. 293–306.

[10] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, "End-to-end object detection with transformers," in *European conference on computer vision*. Springer, 2020, pp. 213–229.

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

[12] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hardbound: Architectural support for spatial safety of the c programming language," in *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA: Association for Computing Machinery, 2008, pp. 103–114.

[13] D. Dhurjati, S. Kowshik, and V. Adve, "Safecode: enforcing alias analysis for weakly typed languages," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 144–157. [Online]. Available: https://doi.org/10.1145/1133981.1133999

[14] B. Di, J. Sun, D. Li, H. Chen, and Z. Quan, "GMOD: A dynamic GPU memory overflow detector," in *Proceedings of the 27th ACM International Conference on Parallel Architecture and Compilation Techniques (PACT)*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1–13. [Online]. Available: https://doi.org/10.1145/3243176.3243194

[15] S. B. Dutta, H. Naghibijouybari, N. Abu-Ghazaleh, A. Marquez, and K. Barker, "Leaky buddies: Cross-component covert channels on integrated cpu-gpu systems," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 972–984.

[16] S. B. Dutta, H. Naghibijouybari, A. Gupta, N. Abu-Ghazaleh, A. Marquez, and K. Barker, "Spy in the gpu-box: Covert and side channel attacks on multi-gpu systems," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589080

[17] C. Erb, M. Collins, and J. L. Greathouse, "Dynamic buffer overflow detection for gpgpus," in *Proceedings of the 15th International Symposium on Code Generation and Optimization (CGO)*. Piscataway, NJ, USA: IEEE, 2017, pp. 61–73.

[18] A. Esteva, K. Chou, S. Yeung, N. Naik, A. Madani, A. Mottaghi, Y. Liu, E. Topol, J. Dean, and R. Socher, "Deep learning-enabled medical computer vision," *NPJ digital medicine*, vol. 4, no. 1, p. 5, 2021.

[19] U. Farooq and H. Mehrez, "Pre-silicon verification using multi-fpga platforms: A review," *Journal of Electronic Testing*, vol. 37, no. 1, pp. 7–24, 2021.

[20] S. Ghemawat and P. Menage, "Tcmalloc: Thread-caching malloc," 2009.

[21] Y. Guo, Z. Zhang, and J. Yang, "{GPU} memory exploitation for fun and profit," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 4033–4050.

[22] M. T. Ibn Ziad, M. A. Arroyo, E. Manzhosov, R. Piersma, and S. Sethumadhavan, "No-fat: Architectural support for low overhead memory safety checks," in *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA)*. Piscataway, NJ, USA: IEEE Press, 2021, pp. 916–929.

[23] M. T. Ibn Ziad, M. A. Arroyo, E. Manzhosov, and S. Sethumadhavan, "ZerØ: Zero-overhead resilient operation under pointer integrity attacks," in *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA)*. Piscataway, NJ, USA: IEEE, 2021, pp. 999–1012.

[24] Intel, "5-level paging and 5-level ept," https://www.intel.com/content/www/us/en/content-details/671442/5-level-paging-and-5-level-ept-white-paper.html, 2017.

[25] J. Janai, F. Güney, A. Behl, A. Geiger *et al.*, "Computer vision for autonomous vehicles: Problems, datasets and state of the art," *Foundations and Trends® in Computer Graphics and Vision*, vol. 12, no. 1–3, pp. 1–308, 2020.

[26] H. Jeon, N. Karimian, and T. Lehman, "A new foe in gpus: Power side-channel attacks on neural network," in *2021 22nd International Symposium on Quality Electronic Design (ISQED)*, 2021, pp. 313–313.

[27] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the nvidia volta gpu architecture via microbenchmarking," 2018.

[28] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of c," in *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '02. USA: USENIX Association, 2002, p. 275–288.

[29] A. Karki, C. P. Keshava, S. M. Shivakumar, J. Skow, G. M. Hegde, and H. Jeon, "Detailed characterization of deep neural networks on gpus and fpgas," in *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs*, ser. GPGPU '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 12–21. [Online]. Available: https://doi.org/10.1145/3300053.3319418

[30] Khronos Group, "The OpenCL specification," https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf, 2015.

[31] H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, T. Pho, H. Kim, and R. Hadidi, "MacSim: A cpu-gpu heterogeneous simulation framework user guide," 2012, https://github.com/gthparch/macsim.

[32] Y. Kim, J. Lee, and H. Kim, "Hardware-based always-on heap memory safety," in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Piscataway, NJ, USA: IEEE, 2020, pp. 1153–1166.

[33] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO): Feedback-Directed and Runtime Optimization*. USA: IEEE Computer Society, 2004, pp. 75–86.

[34] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, "Preventing use-after-free with dangling pointers nullification." in *NDSS*, 2015.

[35] J. Lee, Y. Kim, J. Cao, E. Kim, J. Lee, and H. Kim, "Securing gpu via region-based bounds checking," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 27–41. [Online]. Available: https://doi.org/10.1145/3470496.3527420

[36] M. LeMay, J. Rakshit, S. Deutsch, D. M. Durham, S. Ghosh, A. Nori, J. Gaur, A. Weiler, S. Sultana, K. Grewal, and S. Subramoney, "Cryptographic capability computing," in *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. New York, NY, USA: Association for Computing Machinery, 2021, p. 253–267. [Online]. Available: https://doi.org/10.1145/3466752.3480076

[37] N. D. Matsakis and F. S. Klock II, "The rust language," in *ACM SIGAda Ada Letters*, vol. 34, no. 3. ACM, 2014, pp. 103–104.

[38] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Watchdog: Hardware for safe and secure manual memory management and full memory safety," in *Proceedings of the 39st Annual International Symposium on Computer Architecture (ISCA)*. USA: IEEE Computer Society, 2012, pp. 189–200.

[39] S. Nagarakatte, J. Zhao, M. M. Martin, and S. 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*, ser. PLDI '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 245–258. [Online]. Available: https://doi.org/10.1145/1542476.1542504

[40] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Cets: Compiler enforced temporal safety for c," in *Proceedings of the 2010 International Symposium on Memory Management*, ser. ISMM '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 31–40. [Online]. Available: https://doi.org/10.1145/1806651.1806657

[41] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, "Rendered insecure: Gpu side channel attacks are practical," ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2139–2153. [Online]. Available: https://doi.org/10.1145/3243734.3243831

[42] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, "Graphbig: Understanding graph computing in the context of industrial solutions," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. New York, NY, USA: Association for Computing Machinery, 2015, pp. 1–12. [Online]. Available: https://doi.org/10.1145/2807591.2807626

[43] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, 2007, pp. 89–100.

[44] NVIDIA, "Compute sanitizer." [Online]. Available: https://docs.nvidia.com/compute-sanitizer/ComputeSanitizer/index.html

[45] NVIDIA., "CUDA Toolkit Documentation," https://docs.nvidia.com/cuda/index.html, 2017.

[46] NVIDIA, "Parallel thread execution ISA version 8.5," 2020. [Online]. Available: https://docs.nvidia.com/cuda/pdf/ptx_isa_8.5.pdf

[47] NVIDIA, "Fastertransformer," https://github.com/NVIDIA/FasterTransformer/, 2021.

[48] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel MPX explained: A cross-layer analysis of the Intel MPX system stack," *Proceedings of the ACM Measurement and Analysis of Computing Systems (POMACS)*, vol. 2, no. 2, pp. 1–30, Jun. 2018.

[49] OpenACC, "Openacc specification," 2022. [Online]. Available: https://www.openacc.org/specification

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

[51] S.-O. Park, O. Kwon, Y. Kim, S. K. Cha, and H. Yoon, "Mind control attack: Undermining deep learning with GPU memory exploitation," *Computers & Security*, vol. 102, p. 102115, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404820303886

[52] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, pp. 1–34, 2012.

[53] H. Sasaki, M. A. Arroyo, M. T. I. Ziad, K. Bhat, K. Sinha, and S. Sethumadhavan, "Practical byte-granular memory blacklisting using califorms," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 558–571.

[54] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (ATC)*. USENIX, 2012, pp. 309–318.

[55] R. Sharifi and A. Venkat, "CHEx86: Context-sensitive enforcement of memory safety via microcode-enabled capabilities," in *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 762–775.

[56] K. Sinha and S. Sethumadhavan, "Practical memory safety with REST," in *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*. Piscataway, NJ, USA: IEEE Press, 2018, pp. 600–611.

[57] M. B. Sullivan, M. T. I. Ziad, A. Jaleel, and S. 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*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589102

[58] M. Tarek Ibn Ziad, S. Damani, A. Jaleel, S. W. Keckler, and M. Stephenson, "Cucatch: A debugging tool for efficiently catching memory safety violations in cuda applications," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, jun 2023. [Online]. Available: https://doi.org/10.1145/3591225

[59] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[60] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "NVBit: A dynamic binary instrumentation framework for nvidia gpus," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 372–383. [Online]. Available: https://doi.org/10.1145/3352460.3358307

[61] Y. Wang, C. Yang, S. Farrell, T. Kurth, and S. Williams, "Hierarchical roofline performance analysis for deep learning applications," *CoRR*, vol. abs/2009.05257, 2020. [Online]. Available: https://arxiv.org/abs/2009.05257

[62] B. Wickman, H. Hu, I. Yun, D. Jang, J. Lim, S. Kashyap, and T. Kim, "Preventing {Use-After-Free} attacks with fast forward allocation," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2453–2470.

[63] J. Woodruff, A. Joannou, H. Xia, A. Fox, R. M. Norton, D. Chisnall, B. Davis, K. Gudka, N. W. Filardo, A. T. Markettos, M. Roe, P. G. Neumann, R. N. M. Watson, and S. W. Moore, "CHERI concentrate: Practical compressed capabilities," *IEEE Transactions on Computers (TC)*, vol. 68, no. 10, pp. 1455–1469, 2019.

[64] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *Proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA)*. Piscataway, NJ, USA: IEEE Press, 2014, pp. 457–468.

[65] H. Xia, J. Woodruff, S. Ainsworth, N. W. Filardo, M. Roe, A. Richardson, P. Rugg, P. G. Neumann, S. W. Moore, R. N. M. Watson, and T. M. Jones, "CHERIvoke: Characterising pointer revocation using CHERI capabilities for temporal memory safety," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 545–557.

[66] E. Xie, W. Wang, Z. Yu, A. Anandkumar, J. M. Alvarez, and P. Luo, "Segformer: Simple and efficient design for semantic segmentation with transformers," *Advances in neural information processing systems*, vol. 34, pp. 12 077–12 090, 2021.

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

[68] F. Zeng, B. Dong, Y. Zhang, T. Wang, X. Zhang, and Y. Wei, "Motr: End-to-end multiple-object tracking with transformer," in *European Conference on Computer Vision (ECCV)*, 2022.

[69] Y. Zhang, Z. Zhu, W. Zheng, J. Huang, G. Huang, J. Zhou, and J. Lu, "Beverse: Unified perception and prediction in birds-eye-view for vision-centric autonomous driving," *arXiv preprint arXiv:2205.09743*, 2022.