

CAMI: A Context-Aware Isolation Architecture for GPU Memories

Hao Lan^{1,2,3}, Wei Yan^{1,2,3✉}, Qinfen Hao^{1,2,3}, Xiaochun Ye^{1,2}, Yier Jin⁴, Yong Liu^{2,5}, Ninghui Sun^{1,2,3}

¹ SKLP, Institute of Computing Technology, CAS, Beijing, China

² School of Computer Science and Technology, UCAS, Beijing, China

³ Zhongguancun Laboratory, Beijing, China

⁴ University of Science and Technology of China, Hefei, Anhui, China

⁵ Qi-AnXin Technology Group, QAX Security Center, Beijing, China

{ lanhao20s, yanwei, haoqinfen, yexiaochun, snh}@ict.ac.cn

jinyier@ustc.edu.cn

liuyong@zgclab.edu.cn

Abstract—The widespread use of GPUs in cloud and high-performance computing makes memory isolation a critical security requirement. While the programming model assumes that each thread local memory is private, the underlying hardware does not always enforce this guarantee. Weaknesses in address translation can allow one thread to access another local memory, creating a semantic gap that enables cross-thread corruption and exploitation. To address these challenges, we propose CAMI, a hardware-level framework that integrates fine-grained execution context into the memory translation pipeline. CAMI enforces a binding between the execution context of each memory access and the ownership of its target memory page, ensuring that even subtle inconsistencies in translation cannot be exploited. By introducing an efficient hardware enforcement unit within the MMU and extending page table entries with ownership metadata, CAMI achieves strong, fine-grained isolation while maintaining low performance overhead. We implement CAMI in a cycle-accurate GPU simulator and conduct comprehensive evaluations. Results show that CAMI effectively eliminates cross-thread memory access vulnerabilities with minimal runtime cost, offering a practical path toward secure and high-performance GPU architectures.

Index Terms—GPU Memory Isolation, Fine-Grained Access Control, Hardware Security Mechanisms

I. INTRODUCTION

In modern cloud environments, GPUs are consolidated to maximize utilization. Providers use techniques such as batching and containerization to run multiple workloads on the same device [1], [2]. While this model brings clear efficiency benefits, it also increases the demand for strong isolation mechanisms, since even small flaws can cause data leakage or compromise computation integrity [3]–[6].

Existing solutions focus mainly on coarse-grained boundaries. Container isolation relies on software, which is fragile against stack vulnerabilities [7], [8]. Hardware partitioning such as NVIDIA MIG provides stronger isolation, but only at the instance level [9]–[12]. Trusted execution environments (TEEs) offer hardware-enforced protection but entail considerable performance costs for communication-intensive AI workloads [13]–[17]. As a result, these solutions fall short

when faced with fine-grained isolation requirements inside a single GPU instance.

A more fundamental weakness lies in thread-level isolation. The GPU programming model promises that each thread has private local memory. However, the underlying hardware implementation does not fully enforce this guarantee. Recent studies have shown that a thread can obtain another thread local memory address and corrupt its private data. This semantic gap between the programming model and hardware creates a hidden attack surface. Once exploited, it enables powerful attacks such as stack corruption, code injection, and return-oriented programming (ROP), which cannot be mitigated by higher-level software defenses [18].

To address this problem, we present CAMI (Context-Aware Memory Isolation), a hardware framework that enforces thread-level memory isolation in GPUs. CAMI binds each memory access to its originating execution context, ensuring that a thread can only operate on its own private data. This binding is realized by extending page table entries (PTEs) with ownership information and introducing lightweight checks into the memory management unit (MMU) translation path. By aligning hardware behavior with the programming model, CAMI eliminates the semantic gap that attackers can exploit while maintaining negligible performance overhead. Our main contributions are as follows.

- We propose a novel hardware architecture for thread-level memory isolation on GPUs. Our novel context-aware approach embeds fine-grained execution context into the address translation process via extended PTEs to enforce per-thread ownership.
- We design an efficient hardware enforcement mechanism that implements selective access control within the MMU with minimal performance overhead, effectively decoupling strong security from high-performance computing requirements.
- We implement the CAMI framework in a cycle-level GPU simulator and conduct a comprehensive evaluation, demonstrating its effectiveness in mitigating architectural

vulnerabilities and its potential as a foundational framework for protecting fundamental GPU isolation units.

II. BACKGROUND

A. GPU Programming and Memory Model

The computational efficiency of modern GPUs stems from their hierarchical parallel execution model [19]. In programming paradigms such as CUDA, a computational task is organized as a grid consisting of multiple thread blocks. A thread block serves as the fundamental unit of resource allocation, scheduling, and synchronization [20]. Within each block, threads are further organized into warps, which follow the single-instruction multiple-thread (SIMT) execution model. This hierarchy not only enables large-scale parallelism but also defines natural boundaries for collaboration and isolation across different groups of threads.

To support this execution model, GPU architectures provide a hierarchical memory system aligned with the levels of parallelism. Global memory resides off-chip. It offers high-capacity storage accessible by all threads, but at the cost of relatively high latency. To mitigate this gap, each streaming multiprocessor (SM) integrates low-latency on-chip storage resources, including register files and shared memory, which support diverse access patterns. Additionally, when register pressure is high, the compiler may allocate per-thread local memory in off-chip device memory, though this remains logically private to each thread.

Two memory spaces are particularly relevant to this study. The first is local memory, which is semantically defined as private to each thread. It functions as a per-thread call stack, storing local variables and control-flow data. Preserving the confidentiality and integrity of local memory is therefore a fundamental security requirement of the GPU programming model. The second is shared memory, whose visibility is restricted to all threads within the same block. Shared memory is physically located on-chip, providing low-latency access and serving as a key mechanism for intra-block communication and synchronization. However, this shared accessibility also makes it a potential channel for inter-thread information leakage.

B. GPU Virtual Memory and Address Translation

Modern GPUs widely adopt virtual memory mechanisms, with the MMU serving as the core of the address translation process. When a GPU core issues a memory request, the corresponding virtual address (VA) is forwarded to the MMU. The MMU first probes its translation lookaside buffer (TLB) for a cached mapping. On a TLB hit, the virtual-to-physical address (PA) translation is completed with minimal latency.

On a TLB miss, the hardware page table walker (PTW) is invoked. The PTW traverses multi-level page tables stored in main memory to retrieve the required PTEs. This process incurs multiple high-latency memory accesses and represents a significant performance bottleneck in the GPU memory subsystem. As the fundamental unit of translation, each PTE encodes the VA-PA mapping together with basic access permission bits.

The address translation pipeline also serves as the hardware enforcement point for memory protection. However, in pursuit of optimizing accesses to diverse memory spaces such as global and local memory, GPUs do not rely on a fully uniform translation path. For conventional virtual addresses, the MMU follows the standard page table traversal. For certain special address spaces, specifically local memory, the hardware employs customized translation paths to optimize performance. Unlike the standard TLB lookup, these paths often utilize formulaic address calculation logic based on the thread context. This divergence at the implementation level is central to the complexity of enforcing strong memory isolation in GPU architectures.

C. Semantic Gap in GPU Thread Isolation

The security of fine-grained parallelism on GPUs fundamentally relies on the faithful execution of the programming model isolation boundaries by the hardware address translation mechanisms. Although the thread-level privacy of resources like local memory is a cornerstone of this model, the underlying architectural implementation employs complex and often non-uniform translation paths to optimize performance. This design leads to a deviation between the hardware behavior and the abstract promises of the programming model, thereby creating a potential security risk.

This deviation can be characterized as a “semantic gap”: a discrepancy between the abstract security guarantees promised to the programmer and the physical reality of how memory is accessed and managed by the hardware. This gap implies that isolation is not always an inherent property of the hardware but is instead contingent upon threads exclusively using the intended, context-aware access pathways. The existence of any alternative, context-agnostic hardware path to the same physical data represents a latent architectural vulnerability, as it provides a potential means to bypass the intended security checks.

A pioneering study by Guo et al. provides compelling empirical evidence for the existence of such a semantic gap [18]. Their work on modern GPUs revealed a “dual-mapping” vulnerability in which thread-private local memory is accessible via two distinct translation paths. While legitimate accesses utilize safe, context-aware instructions like LDL and STL that rely on hardware-managed offsets, the same physical memory is also mapped to a context-agnostic generic virtual address. Critically, the GPU instruction set acts as a “bridge” by allowing threads to traverse from the safe path to the unsafe one using standard global memory instructions such as LDG and STG. This capability bypasses context-based isolation checks and enables a thread to corrupt the private stack of another thread simply by constructing its aliased virtual address.

III. THREAT MODEL

In this study, we consider a threat model centered on an attacker’s ability to compromise the fundamental thread-level isolation of GPU local memory. The attacker and victim

threads are assumed to execute within the same protection domain, such as a single user process, and are co-located within a single thread block. This setting is representative in parallel algorithms that rely on intra-block thread collaboration. The attacker can deploy arbitrary CUDA kernels, access block-level shared memory as a communication channel, and issue standard GPU instructions.

The attacker’s goal is to break the thread-private guarantees of the local memory by exploiting architectural inconsistencies in the GPU address translation pathways. An attacker can leverage certain instructions or hardware mechanisms to discover a context-agnostic virtual address that maps to a victim thread private physical memory. By accessing this address, the attacker aims to gain arbitrary read and write access to the victim stack. This potent capability enables further exploits, including leakage of sensitive data that violates confidentiality or the corruption of stack contents that hijacks control flow and compromises integrity.

Our defense scope is strictly limited to mitigating these cross-thread local memory attacks that occur within a single thread block by exploiting address translation mechanisms to bypass hardware isolation. The proposed CAMI framework intercepts such attacks at the MMU level. Other threats, such as off-chip DRAM attacks, privileged software compromise, and broader microarchitectural side channels, are beyond the scope of this model.

IV. DESIGN

A. Overview

To overcome the architectural limitations of thread-level memory isolation in GPUs, we propose CAMI. CAMI addresses the semantic gap between programming abstractions and hardware enforcement by embedding execution context directly into the address translation process. In this way, CAMI provides strong per-thread isolation while preserving the performance efficiency required by modern GPU workloads. The framework is guided by three core objectives.

- **Strong security:** the defense must fundamentally prevent isolation bypasses that arise from inconsistencies in memory translation, providing hardware-enforced protection for the basic execution units of the GPU.
- **Fine granularity:** protection must be enforced at the level of threads, in order to meet the requirements of complex privacy-preserving workloads.
- **High performance:** the additional security mechanisms must impose minimal overhead on the critical paths of computation and memory access, avoiding the sharp trade-offs between performance and security.

To achieve these objectives, CAMI introduces a core design principle: binding the execution context of a memory access with the ownership of the target memory page. This principle directly bridges the semantic gap between the isolation guarantees of the programming model and the actual enforcement in hardware.

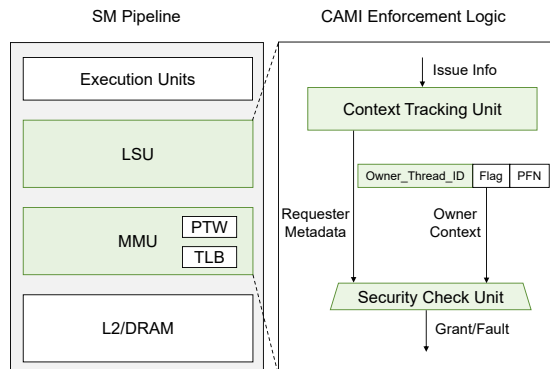


Fig. 1. CAMI architecture overview.

CAMI incorporates two key enhancements to realize binding in the architecture. First, it extends PTEs to carry fine-grained ownership metadata. Second, it integrates a lightweight security checker into the MMU to enforce access control at the final stage of address translation. Together, these mechanisms ensure that every translation-based memory access is explicitly verified against its ownership, thereby providing strong thread-level isolation without compromising GPU performance. The overall architecture of CAMI is illustrated in Fig. 1. The CAMI framework achieves this core design principle through the introduction of two cooperative functional modules, which are systematically integrated into the GPU memory access pipeline.

- **Context Tracking Unit (CTU):** Located in the load/store unit (LSU), the CTU captures the execution context of the issuing thread at instruction dispatch and associates it with the corresponding memory request.
- **Security Check Unit (SCU):** Serving as the CAMI decision engine, the SCU is integrated into the MMU. It receives the context provided by the CTU together with ownership metadata extracted from the address translation path, performs hardware-level permission validation, and delivers the final decision of either permit or fault.

B. Management of Ownership Metadata

CAMI materializes abstract ownership semantics as verifiable physical attributes by extending the PTEs, the atomic unit of address translation, to embed ownership metadata. Embedding ownership information directly in the PTEs ensures that every translation, whether resolved by a TLB hit or a page table walk, is intrinsically bound to the security metadata. This design guarantees universality and mandatory enforcement. Because the PTE shares its lifecycle with the mapped page, ownership is consistently preserved across allocation, use, and deallocation. The approach directly builds on the existing page table architecture while maintaining compatibility with the broader virtual memory subsystem.

The extended PTE introduces two dedicated fields: `Owner_Thread_ID` and a `Protection_Flag`. The `Owner_Thread_ID` stores the unique hardware identifier of

the thread that owns the memory page. The `Protection_Flag` is set only for thread-private local memory pages to activate SCU verification. For global or shared memory pages, this flag remains unset, instructing the SCU to bypass checks and preserving standard access semantics. Recording this identifier in the PTE transforms the software-level abstraction of “thread-private” memory into a hardware-verifiable attribute.

The establishment of ownership metadata follows a hardware–software co-design strategy. When a local memory page is first allocated, the GPU driver or firmware page-fault handler creates the corresponding PTE. At this point, the driver reads the thread context ID from GPU state registers and writes it into the `Owner_Thread_ID` field. We extend kernel data structures to carry default isolation policies and emulate this behavior in helper functions for memory allocation, ensuring that ownership is initialized correctly at the beginning of the page lifecycle. When a page is swapped out or released, the corresponding PTE is invalidated, and the embedded ownership information becomes obsolete.

Embedding ownership metadata in the PTE also raises consistency considerations. A change in ownership or a page release requires not only updating the PTE but also invalidating stale TLB entries across GPU cores, a process known as TLB shutdown. However, local memory exhibits highly stable ownership: from allocation to release, a page `Owner_Thread_ID` remains constant. As a result, shutdowns due to ownership updates are practically unnecessary. Ownership establishment and destruction naturally align with page allocation and deallocation, making existing memory management mechanisms sufficient without introducing new hardware coherence protocols.

C. Context-Aware MMU Enforcement

The enforcement of ownership metadata is realized through enhancements to the GPU memory subsystem. Two additional hardware modules operate in tandem to implement this mechanism: the CTU within the LSU and the SCU embedded in the MMU. Together, these modules enforce ownership validation directly in the address translation pipeline.

The CTU is a lightweight logic block colocated with the LSU. Its role is to capture the global hardware identifier of the issuing thread at the moment a local memory access instruction is dispatched. The identifier is then transmitted as part of the memory request packet metadata, providing the SCU with low-latency and reliable context information. For accesses targeting local memory, the hardware preserves lane-level metadata to ensure precise thread identification. By positioning the CTU at the front of the memory pipeline, identity information is captured at the point of request generation, eliminating the need for reconstruction within the MMU.

The SCU is the final decision point of CAMI enforcement logic. Integrated into the MMU pipeline after address translation but before memory requests are issued, the SCU ensures that enforcement does not interfere with the performance-critical TLB lookup. It takes two inputs: the requester ID

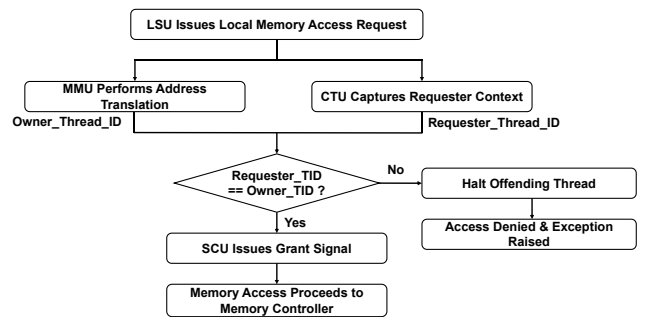


Fig. 2. CAMI security check workflow.

from the CTU and the owner ID extracted from the PTE or TLB entry. In hardware, the SCU is deliberately simple, consisting of an N-bit comparator and minimal control logic. The comparator checks the equality of the two identifiers and produces a one-bit signal indicating whether the access is permitted or denied.

D. Security Check Workflow

The SCU is tightly synchronized with the address translation pipeline. For a TLB hit, the process unfolds as a pipelined sequence. First, the LSU issues a memory request, the virtual address enters TLB lookup, and the CTU captures the requester ID to forward alongside the request. Next, upon lookup completion, the TLB returns both the physical mapping and the stored owner ID. Finally, the SCU comparator evaluates the two identifiers and generates a single-bit control signal indicating permit or fault.

This workflow overlaps with existing pipeline stages, introducing minimal latency, which modern pipelining techniques can effectively hide. During TLB misses, when the PTW is active and the pipeline stalls, the SCU remains idle. Since page table walks dominate latency in this path, the SCU overhead is completely masked.

When the SCU detects a violation, it raises a hardware security fault rather than allowing the access. The exception halts the execution of the offending warp and marks its state as a memory access violation. Violation metadata, including the requester thread ID and faulting address, is recorded into dedicated registers. These registers are accessible to system software for error handling, auditing, or further analysis. This fault mechanism ensures that violations are isolated and reported without corrupting system state or interfering with unrelated threads.

V. EXPERIMENT

A. Experimental Setup

We conduct all experiments on GPGPU-Sim v4.0, configured to model the NVIDIA Volta microarchitecture [21]. The main architectural parameters are summarized in Table I, representing a typical high-performance GPU suitable for evaluating both performance and security trade-offs. Two system configurations are considered: baseline, corresponding to

the unmodified simulator without additional security mechanisms, and CAMI, which integrates our context-aware isolation framework under the strict `THREAD_PRIVATE` policy.

TABLE I
KEY PARAMETERS OF THE SIMULATED GPU ARCHITECTURE.

Parameter	Value
SMs	80 @ 1.132 GHz
L1 Data Cache (per SM)	32 KB
Shared Memory (per SM)	96 KB
Shared L2 Cache	6 MB, 24-way, 32 Slices
DRAM	0.85 GHz

For workloads, we use the Rodinia v3.1 benchmark suite, a standard collection in GPU architecture research [22]. We select eight representative applications with diverse characteristics, ranging from memory-intensive kernels such as Streamcluster to compute-intensive workloads such as Hotspot. This selection allows us to capture a broad spectrum of access patterns and computational behaviors, ensuring that the evaluation of CAMI covers both computation-heavy and memory-bound scenarios.

B. Security Validation

To evaluate the effectiveness of CAMI against attacks that exploit inconsistencies in GPU address translation, we design a set of proof-of-concept (PoC) kernels that explicitly attempt to violate thread-level isolation. To intuitively illustrate the attack and defense processes, we introduce a visualization methodology referred to as the memory access fingerprint.

The fingerprint is represented as a two-dimensional scatter plot that records memory access patterns during program execution. The Y-axis denotes the local memory offset, which logically corresponds to the virtual address space allocated to each hardware thread. The X-axis represents the issuing thread within a warp. Each access is visualized as a data point, with its shape and color encoding both the operation type (read or write) and the thread that initiated it. In a secure system, legitimate accesses should remain strictly confined to their respective thread regions. Any data point intruding into another thread region with a mismatched color directly indicates a successful cross-thread attack.

Based on this methodology, we implement three representative attacks that capture distinct threat behaviors: integrity violation, confidentiality leakage, and control-flow hijacking. These cover the essential ways in which cross-thread memory corruption undermines GPU execution. The consolidated results are summarized in Fig. 3.

Integrity Attack. In the first scenario, we simulate a direct attempt to compromise computational integrity. Here, the attacker thread (T1) leverages the address translation vulnerability to locate the local stack frame of the victim thread (T0) and overwrite a critical variable. As shown in Fig. 3 (a1), under the baseline system we observe a red cross marker in T0 region, indicating that T1 has successfully issued an illegal write to T0 private data. Even with a generic virtual address,

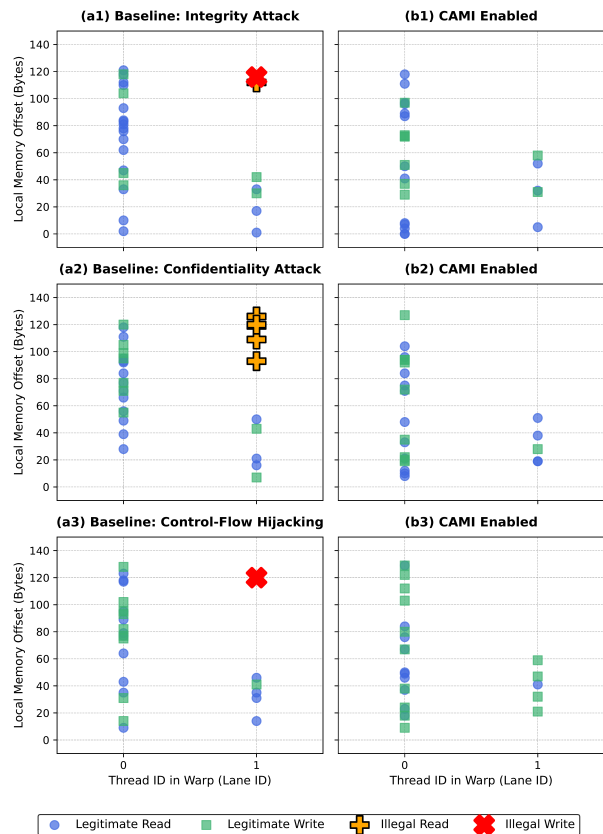


Fig. 3. Security validation across attack scenarios.

the access must traverse the MMU where the victim PTE enforces ownership. In contrast, Fig. 3 (b1) demonstrates CAMI defense: the SCU validates ownership during the final stage of address translation, detects the mismatch between requester TID and memory owner TID, and aborts the offending warp through a hardware exception. Consequently, no cross-thread data points appear, confirming CAMI ability to preserve data integrity.

Confidentiality Attack. The second scenario models an attempt to exfiltrate sensitive information. Instead of overwriting, the attacker thread (T1) tries to read a secret variable (e.g., an intermediate cryptographic key) from the victim local memory. Fig. 3 (a2) illustrates the baseline result: orange markers appear in T0 region, showing that T1 successfully performed an illegal read. This outcome demonstrates the lack of confidentiality protection in the unmodified system. With CAMI enabled (Fig. 3 (b2)), all such accesses are blocked by the SCU, which prevents the disclosure of data across thread boundaries. The absence of cross-colored markers within T0 region shows that CAMI enforces confidentiality guarantees in the same robust manner as it enforces integrity.

Control-Flow Hijacking. Finally, we simulate a control-flow hijacking attack, which represents the most severe threat. In this case, T0 executes a function call and saves its return address on the local stack. The attacker thread (T1) attempts to

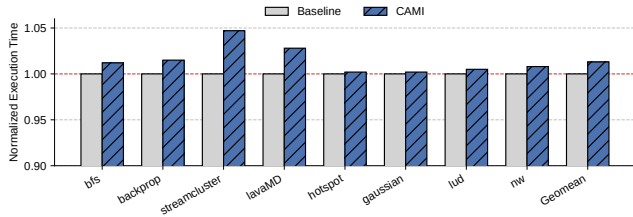


Fig. 4. Performance overhead of CAMI on Rodinia benchmarks, normalized to a baseline without system protection.

tamper with this return address by performing an illegal write. Fig. 3 (a3) shows that under the baseline system, a red cross marker appears in T0 region, indicating successful corruption of the control-flow metadata. Once the function returns, the victim execution is fully hijacked. In sharp contrast, Fig. 3 (b3) shows that CAMI successfully blocks the attack at the hardware level: the SCU intercepts the illegal write, ensuring that critical control-flow information such as return addresses is protected with the same rigor as ordinary data. This highlights CAMI ability to guarantee both data and control-flow integrity.

In summary, across all three representative scenarios, the memory access fingerprint visualization clearly shows that while the baseline GPU system permits cross-thread corruption and leakage, CAMI consistently enforces thread-level ownership and eliminates illegal accesses. These results validate CAMI as a comprehensive defense mechanism against integrity, confidentiality, and control-flow threats in GPU execution.

C. Performance and Overhead Analysis

To assess the performance implications of CAMI, we measure its impact on overall execution time across diverse workloads. Fig. 4 reports the normalized execution time of eight representative Rodinia benchmarks under the strict `THREAD_PRIVATE` policy, with the Baseline system set to 1.0. The results show that the overhead introduced by CAMI remains negligible across all workloads.

A more detailed analysis based on memory access characteristics further clarifies these findings. For compute-intensive applications such as Hotspot and Gaussian, the high computation-to-memory ratio reduces the frequency of CAMI checks, resulting in negligible performance overhead. Even for memory-intensive workloads such as Streamcluster and LavaMD, which are highly sensitive to latency and bandwidth, the additional cost remains minimal, with the largest slowdown observed in Streamcluster at only 4.7%. This is primarily due to the increased memory traffic for metadata fetching during TLB misses. Across all benchmarks, the geometric mean overhead is 1.3%, demonstrating that CAMI delivers strong hardware-level isolation while incurring only minor performance penalties.

VI. RELATED WORK

The demand for GPU security has led to diverse isolation mechanisms across software and hardware. Existing solutions can be broadly classified into software-based isolation, TEEs and hardware resource partitioning. While each approach addresses certain threats, they often suffer from coarse granularity, high overhead, or insufficient protection against architectural vulnerabilities.

Software-based isolation. Confidential containers and related techniques provide process-level protection through standard system software. Their flexibility comes at the cost of a large trusted software base, where a single kernel or driver vulnerability can compromise the entire isolation boundary.

GPU TEEs. Both academic and industrial efforts have explored TEEs on GPUs. Research prototypes such as Graviton demonstrate the feasibility of building encrypted trusted computing bases, while commercial solutions like NVIDIA H100 extend CPU-side confidential computing to GPUs with hardware support for encryption and attestation [23]. These approaches provide strong guarantees against privileged and physical adversaries but remain coarse-grained, leaving intra-process threats unaddressed.

Hardware partitioning. NVIDIA MIG enables strong instance-level separation by splitting cores and memory. Cache partitioning provides finer granularity but requires static division of scarce cache space, which disrupts performance-critical sharing [24].

Positioning of this work. CAMI complements these approaches by targeting the most fine-grained boundary: threads within a GPU instance. Rather than physical partitioning, it enforces context-aware checks in the translation path, providing strong thread-level isolation with negligible overhead.

VII. CONCLUSION

The widespread adoption of GPUs in cloud and high-performance computing makes memory isolation a fundamental security requirement. This paper has examined architectural inconsistencies in GPU address translation that undermine thread-level isolation, exposing vulnerabilities that higher-level software defenses cannot contain. To address this gap, we introduce CAMI, a context-aware memory isolation framework that embeds execution context into the hardware translation path by extending page table entries with ownership metadata and integrating lightweight enforcement logic in the MMU. Our cycle-accurate evaluation shows that CAMI effectively prevents a wide range of translation-based cross-thread attacks targeting integrity, confidentiality, and control flow, while incurring an average performance overhead of less than 5% across diverse GPU workloads. These results demonstrate that CAMI provides a practical and efficient foundation for fine-grained GPU memory isolation, complementing existing coarse-grained security mechanisms and paving the way for more secure GPU architectures.

REFERENCES

- [1] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of large-scale multi-tenant GPU clusters for DNN training workloads," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 947–960.
- [2] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 463–479.
- [3] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, "Rendered insecure: GPU side channel attacks are practical," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 2139–2153.
- [4] Y. Kim, J. Lee, and H. Kim, "Hardware-based always-on heap memory safety," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 1153–1166.
- [5] J. Wei, Y. Zhang, Z. Zhou, Z. Li, and M. A. Al Faruque, "Leaky DNN: Stealing deep-learning model secret with GPU context-switching side-channel," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020, pp. 125–137.
- [6] 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*, 2023, pp. 1–13.
- [7] Confidential Containers Project, "Confidential containers," <https://confidentialcontainers.org/>, accessed: 2025-09-13.
- [8] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 2015, pp. 171–172.
- [9] NVIDIA Corporation, "Nvidia Multi-Instance GPU User Guide," <https://docs.nvidia.com/datacenter/tesla/mig-user-guide/>, 2020, accessed: 2025-09-13.
- [10] M. T. I. Ziad, M. A. Arroyo, E. Manzhosov, R. Piersma, and S. Sethumadhavan, "No-fat: Architectural support for low overhead memory safety checks," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 916–929.
- [11] 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*, 2022, pp. 27–41.
- [12] Y. Zhang, H. Yu, C. Han, C. Wang, B. Lu, Y. Li, X. Chu, and H. Li, "Missile: Fine-grained, hardware-level GPU resource isolation for multi-tenant DNN inference," *arXiv e-prints*, pp. arXiv–2407, 2024.
- [13] S. Volos, K. Vaswani, and R. Bruno, "Graviton: Trusted execution environments on GPUs," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 681–696.
- [14] I. Jang, A. Tang, T. Kim, S. Sethumadhavan, and J. Huh, "Heterogeneous isolated execution for commodity GPUs," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 455–468.
- [15] J. Zhu, R. Hou, X. Wang, W. Wang, J. Cao, B. Zhao, Z. Wang, Y. Zhang, J. Ying, L. Zhang *et al.*, "Enabling rack-scale confidential computing using heterogeneous trusted execution environment," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1450–1465.
- [16] T. Hunt, Z. Jia, V. Miller, A. Szekely, Y. Hu, C. J. Rossbach, and E. Witchel, "Telekine: Secure computing with cloud GPUs," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 817–833.
- [17] Y. Deng, C. Wang, S. Yu, S. Liu, Z. Ning, K. Leach, J. Li, S. Yan, Z. He, J. Cao *et al.*, "Strongbox: A GPU tee on arm endpoints," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 769–783.
- [18] 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.
- [19] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for?" *Queue*, vol. 6, no. 2, pp. 40–53, 2008.
- [20] NVIDIA Corporation, *Parallel Thread Execution ISA Version 8.2*, 2023, v8.2. [Online]. Available: <https://docs.nvidia.com/cuda/archive/12.2.1/parallel-thread-execution/index.html>
- [21] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-sim: An extensible simulation framework for validated GPU modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 473–486.
- [22] 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 *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 2009, pp. 44–54.
- [23] NVIDIA, "Nvidia confidential computing," <https://www.nvidia.com/en-us/data-center/solutions/confidential-computing/>, 2022.
- [24] Y. Liang, X. Li, and X. Xie, "Exploring cache bypassing and partitioning for multi-tasking on GPUs," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 9–16.