Challenges and Design Considerations for Finding CUDA Bugs Through GPU-Native Fuzzing

Mingkai Li
Columbia University
USA
mingkai.li@columbia.edu

Joseph Devietti
University of Pennsylvania
USA
devietti@cis.upenn.edu

Suman Jana
Columbia University
USA
suman@cs.columbia.edu

Tanvir Ahmed Khan
Columbia University
USA
tk3070@columbia.edu

Abstract

Modern computing is shifting from homogeneous CPU-centric systems to heterogeneous systems with closely integrated CPUs and GPUs. While the CPU software stack has benefited from decades of memory safety hardening, the GPU software stack remains dangerously immature. This discrepancy presents a critical ethical challenge: the world's most advanced AI and scientific workloads are increasingly deployed on vulnerable hardware components.

In this paper, we study the key challenges of ensuring memory safety on heterogeneous systems. We show that, while the number of exploitable bugs in heterogeneous systems rises every year, current mitigation methods often rely on *unfaithful* translations, *i.e.*, converting GPU programs to run on CPUs for testing, which fails to capture the architectural differences between CPUs and GPUs. We argue that the faithfulness of the program behavior is at the core of secure and reliable heterogeneous systems design. To ensure faithfulness, we discuss several design considerations of a GPU-native fuzzing pipeline for CUDA programs.

CCS Concepts: • **Security and privacy → Software and application security**.

1 Introduction

The end of transistor scaling [16, 27, 48] has forced the industry to aggressively shift from homogeneous CPU-centric systems to heterogeneous architectures with tightly coupled CPUs and parallel accelerators, such as GPUs. Despite the fact that these heterogeneous systems are now the backbone of some of the most critical infrastructures, from large-scale machine learning models [55] in data centers to high-performance scientific simulations [50] in supercomputers, they are vulnerable in terms of security [22, 59] and reliability [30, 56]. In particular, while the CPU software stack has benefited from decades of hardening, *e.g.*, extensive static [14] and dynamic [29] analysis, memory safety tooling [46], and memory safe languages [33], the GPU software stack is comparatively newer and still developing its own testing and debugging ecosystem [3, 7]. As a result, prior work [22, 58, 59] has shown that bugs in heterogeneous systems can lead to sensitive user data leakage, silent data corruptions, or even allow attackers to bypass the host's security hardening mechanisms.

To bridge this gap in heterogeneous systems security, we argue that it is the system designer's ethical responsibility to validate the correctness of GPU programs natively on GPUs. Compromising this principle, current mitigation methods [11, 12, 43] transform heterogeneous GPU programs into homogeneous CPU programs for testing, leading to inaccuracies that allow critical bugs to bypass these mitigations.

Therefore, in this paper, we propose a GPU-native fuzzing pipeline that moves the testing logic directly to the hardware component that the GPU program runs on. Specifically, we discuss the challenges and the design considerations of implementing a GPU-native sanitization and fuzzing pipeline on commodity hardware. Utilizing dynamic binary instrumentation, context-sensitive fuzzing, and type-aware mutations, we aim to ensure memory safety on heterogeneous systems.

2 The Ethical Gap in Heterogeneous Systems Security

In this section, we characterize the widening security gap in heterogeneous systems due to GPU memory safety bugs and analyze why existing mitigation methods fail to ensure faithful protection on heterogeneous systems.

Rising numbers of GPU bugs. We investigate the prevalence of GPU bugs by counting the number of exploitable bugs [1, 10], *i.e.*, Common Vulnerabilities and Exposures (CVE), over the years. Figure 1 shows the corresponding results for two major GPU vendors: (1) NVIDIA and (2) AMD. As we show, with the rising popularity of machine learning workloads, the number of exploitable GPU bugs are rapidly rising over the last few years.

HotEthics 2026, March 22, 2026, Pittsburgh, PA, USA
Submitted 31 January 2026; accepted 13 February 2026.

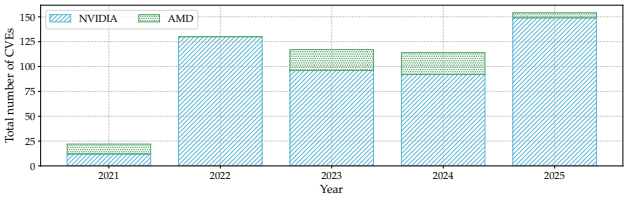

Figure 1. As heterogeneous systems become popular, the number of exploitable bugs in these systems also rises.

Implications of GPU memory safety bugs. GPU memory safety bugs have significant implications. In particular, they lead to return-oriented programming (ROP) attacks [22, 38]. Such attacks undermine the effectiveness of machine learning workloads [22], and even allow attackers to perform arbitrary computations with a Turing-complete gadget set [38]. Furthermore, recent work [44] also shows that malicious attackers can exploit these bugs to compromise the confidentiality of machine learning workloads.

Existing work and why they fall short. Recognizing the urgent need of GPU memory safety, academic researchers and industry practitioners have proposed a wide range of techniques [7, 17, 19, 26, 32, 43, 45, 47, 57, 60] to detect GPU memory safety bugs. Unfortunately, these techniques fail to achieve faithfulness, efficiency, and scalability at the same time. Specifically, existing work [11, 12, 43] transforms heterogeneous GPU programs into homogeneous CPU programs to leverage existing CPU-focused infrastructure [20, 40]. While such solutions help detect bugs in GPU programs, the complex architectural differences between CPUs and GPUs make it very challenging to ensure the faithfulness of the program behavior across this translation process, causing false positives and negatives. Moreover, existing GPU-native solutions [7, 32, 47, 57] either incurs high run-time overhead [7], low fuzzing throughput [57], poor detection accuracy [17, 19], or can not be applied to many real-world GPU applications [32, 47] due to NVIDIA's closed-source ecosystem [4, 8, 9]. Furthermore, some existing solutions [26, 45, 60] require customized hardware changes, and thus are not deployable on commodity GPUs. Given these limitations, it is significantly challenging to detect GPU memory safety bugs through fuzzing, a widely-used technique to detect CPU memory safety bugs. Next, we describe the key challenges of adopting fuzzing for GPUs in §3.

3 Challenges of Securing CUDA Programs

In this section, we discuss the key challenges of ensuring memory safety for CUDA programs through fuzzing: (1) lack of sanitization, (2) input mutation, (3) coverage tracking, and (4) fuzzing harnesses.

Lack of address sanitization. Address sanitization is a widely-used dynamic technique to detect memory safety bugs that are difficult to find using static analysis [14]. Address sanitization enables finding such bugs by adding a number of dynamic checks or instrumentation [40]. Such an instrumentation accesses metadata to detect bugs like buffer overflows and use-after-frees [47]. Once the instrumentation detects such a bug, it crashes the program's execution, improving the bug detection capability of automated techniques like fuzzing [29]. While there exists a wide range of address sanitization techniques for CPUs [6, 18, 31, 34, 40, 52–54], they do not generalize to GPUs [32, 47].

Lack of input mutation. Automated techniques like fuzzing start testing programs with an initial set of user-provided input seeds [36] and randomly change them to detect bugs [21]. As randomly generated inputs are mostly invalid [21], *i.e.*, programs reject them quickly, fuzzing techniques improve the possibility to generate valid inputs through input mutation [15]. Input mutation makes small changes to existing inputs that may still keep the input valid, while also testing new program behavior, such as application logic not tested previously. While there exist many input mutation techniques to detect CPU memory safety bugs [21], they unfortunately do not apply to GPUs [43, 57]. This gap exists because current fuzzing frameworks lack the GPU architecture–specific domain knowledge required to design mutation operators that can effectively trigger memory safety bugs in highly parallel CUDA kernels.

Lack of coverage tracking. Automated bug detection techniques, such as fuzzing, aim to test a program across a large number of unique behavior or application logic [41]. To quantify the effectiveness of testing different application logic, fuzzing techniques leverage code coverage that measures what fraction of a program gets executed during a test for a specific input [13]. If a specific input triggers executions of new control-flow edges (*i.e.*, branches), or new statements in general, fuzzing techniques also reuse the input for further mutation [21]. As code coverage enables fuzzing techniques to measure test effectiveness, while also helping generate interesting inputs, coverage-guided fuzzing has become an effective technique to find bugs in real-world applications [13, 37, 39]. Alas, coverage-guided fuzzing is challenging for heterogeneous systems due to difficulty to track coverage for CUDA code that run on GPUs [43, 57].

Lack of fuzzing harness. To test programs for memory safety bugs, fuzzing techniques also require fuzzing harnesses to invoke programs with mutated inputs [28]. Fuzzing harness ensures testing programs properly by setting up specific contexts programs require [42]. Initializing programs with necessary contexts, fuzzing harness avoids costly false positives or negatives [42], increasing the utilization of testing resources [42]. Unfortunately, existing fuzzing harnesses to test CPU programs do not generalize to GPUs as CUDA programs running on GPUs require specific context initialization, just-in-time (JIT) compilation, and CPU-GPU collaborative execution [43, 57].

4 The GPU-Native Design

To address these challenges, we propose a GPU-native design that utilizes dynamic binary instrumentation to faithfully sanitize and fuzz CUDA programs.

4.1 GPU-Native Address Sanitization and Coverage Tracking

Address Sanitizer. We are designing an address sanitization software tool to detect memory safety bugs for GPUs. A key requirement for an address sanitizer to be practical is to accurately detect memory safety bugs for as many programs as possible at a reasonable overhead. We are achieving this by performing the address sanitization natively on the GPUs instead of CPUs.

We are building an address sanitizer that can detect memory safety bugs in both closed-source and open-source CUDA kernels on off-the-shelf NVIDIA GPUs. Unlike existing solutions that require custom hardware [26] or support only open-source CUDA kernels [32], our address sanitizer will detect memory safety bugs even for closed-source CUDA kernels in commodity NVIDIA GPUs. We are implementing such an address sanitizer using NVBit [49], NVIDIA's dynamic binary instrumentation tool. Using NVBit, we are performing the address sanitization on the GPU itself. The GPU's parallelism will help our implementation speed up address sanitization. Such a GPU-native solution will also enable our technique to detect new types of GPU memory safety bugs that existing techniques fail to detect.

Using NVBit, our sanitizer instruments GPU memory access instructions to detect memory safety bugs. Our sanitizer performs this bug detection along with the execution of GPU CUDA kernels by leveraging GPU parallelism. In particular, our sanitizer will maintain metadata for each unit (*e.g.*, 4 bytes) of GPU global, local, and shared memory, along with metadata for pointers. While accessing memory bytes with a pointer, our sanitizer looks up the metadata corresponding to both the memory bytes and pointers to identify different classes of GPU memory safety bugs.

Coverage Tracking. We are also building a software-only coverage profiling mechanism. The goal of our coverage profiler is to help detect memory safety bugs by improving the effectiveness of fuzzing. Toward this goal, our coverage profiler supports both open-source and closed-source CUDA kernels. To help detect as many bugs as possible, our coverage profiler works on commodity NVIDIA GPUs. Similar to our address sanitizer, our coverage profiler also achieves these goals by using NVBit.

Using NVBit, our coverage profiler instruments GPU control flow instructions during the execution of CUDA kernels. Specifically, our coverage profiler maintains metadata for each control flow instruction. While executing a control flow instruction, the instrumentation logic of our coverage profiler updates the corresponding metadata to count the number of executions. We use these execution counts as feedback to improve the effectiveness of our fuzzer (§4.2).

4.2 Context-Sensitive Fuzzing and Type-Aware Mutations

Detecting memory safety bugs on GPUs requires invoking GPU CUDA kernels with specific contexts and inputs. To ensure specific contexts and inputs, we are proposing context-specific fuzzing and type-aware mutations.

Context-Sensitive Fuzzing. Closed-source NVIDIA libraries often involve multiple layers of abstraction without exposing many low-level kernels for direct usage. Instead, these low-level kernels can only be invoked by high-level kernels. As a result, testing a significant fraction of these libraries' functionality requires setting up a specific chain of functions active on the call stack, *i.e.*, *context*. Furthermore, while running CUDA kernels, GPUs compile the machine-independent Parallel Thread Execution (PTX) code to machine-dependent Source and Assembly (SASS) code just in time [47]. Just-in-time compilation in such a specific context also incurs significant overhead for fuzzing [57]. Therefore, we are enabling context-sensitive fuzzing with low overhead by leveraging the open-source CUDA library samples [5].

The open-source CUDA library samples contain extensive examples that NVIDIA provides to demonstrate the usage of its closed-source CUDA libraries. These examples invoke high-level kernels by setting up the necessary contexts, while also running low-level kernels. We are leveraging these library examples to ensure proper contexts for fuzzing. To use these examples for fuzzing, we will divide their execution among different phases, such as *initialization* phase, *computation* phase, *termination* phase.

The initialization phase will include calling functions to set up the contexts, allocating memory, and copying memory bytes from CPUs to GPUs. The computation phase will include calling the higher-level kernel functions to start GPU computations, while also executing following lower-level kernels. Finally, the termination phase includes calling functions to copy memory bytes from GPUs to CPUs, while also synchronizing and freeing memory bytes.

To divide the execution of library examples among these different phases, we will leverage compiler-based automated techniques [23, 51] while also exploring LLM-assisted manual strategies [25, 35]. Once we divide the execution among different phases, we will amortize the initialization and termination phases across many instances of the computation phase. In particular, we will wrap the execution of the computation phase with our own fuzzing loop. This fuzzing loop will record the input parameters to the high-level kernels, while also setting up the coverage profiling and sanitization infrastructure. Then, the fuzzing loop will run one instance of the computation phase. Based on this instance's outcome (*e.g.*, coverage), the fuzzing loop will then mutate the input and rerun the computation phase. Once the fuzzing loop finds

an interesting instance of the computation phase (*e.g.*, crash, sanitization failure) or completes a pre-determined number of instances for the computation phase, it will finish its execution and invoke the termination phase. By amortizing the initialization and termination phases, our context-sensitive fuzzing will improve the effectiveness and efficiency of memory safety bug detection for GPU CUDA kernels.

Type-Aware Mutations. We are using mutation-based fuzzing to detect memory safety bugs for GPU kernels. In particular, we are leveraging type-aware mutations corresponding to different argument types for different GPU kernels, including integers, floating points, and arrays. Prior work [15] has shown that coverage-guided byte-level mutations fail to bypass the shallow argument-type checks for machine learning kernels. To bypass such checks and test interesting functionality, we are working on type-aware mutations.

As we are currently studying CVEs corresponding to CUDA kernels, we observe that these kernels suffer crashes for some specific input arguments. Such input arguments trigger edge-case behavior, including integers with large positive or negative values overflowing arrays, empty arrays, and arrays with multiple dimensions. Based on our observation, we are using the following mutation strategies for various input arguments for GPU CUDA kernels:

- **Integer arguments**: If a CUDA kernel takes integers as input arguments, we mutate the value of these integer arguments across zero, the maximum positive, and the minimum negative values.
- **Floating point arguments**: Floating point values include different components, such as sign, mantissa, and exponent. Consequently, when a CUDA kernel takes floating-point values as input arguments, we mutate the value of these arguments by mutating these components using several strategies. For example, we vary the 1-bit sign across 0 and 1. For mantissa and exponent, we use several mutation operators, such as flipping one or multiple bits and adding or subtracting values.
- **Arrays**: If a GPU kernel takes arrays as input arguments, we mutate the value of these arguments in two different ways: (1) value mutation and (2) pointer mutation. While mutating array values, we use arrays with large positive and negative values, arrays with dimensions different from the dimension the kernel expects (*e.g.*, 2-dimensional array when the kernel expects a 1-dimensional array). While mutating array pointers, we use pointers to GPU memory space that are different from the GPU memory space the kernel expects. For example, if the GPU kernel expects an argument array allocated on global memory, we mutate it to have a different array allocated on local/shared memory.

5 Preliminary Experimental Results

In this section, we demonstrate the capability of our coverage profiler (§4.1) in analyzing closed-source CUDA kernels. In particular, we present preliminary experimental results for 11 cuBLAS libraries from NVIDIA's CUDA library samples [5].

Table 1. Coverage statistics of 11 cuBLAS library samples. BB in the table indicates basic blocks in GPU kernels. The geometric mean of basic block coverage is only 25.98%.

Library	Total BBs	Hit BBs	BB Cov. (%)	Hit Edges
amax	95	47	49.47	61
amin	106	47	44.34	59
asum	14	9	64.29	13
axpy	30	7	23.33	7
copy	35	6	17.14	6
dot	57	23	40.35	27
nrm2	340	177	52.06	189
rot	81	10	12.35	10
rotm	132	12	9.09	12
scal	19	4	21.05	4
swap	77	10	12.99	10
GeoMean	—	—	**25.98**	—

Experiment Setup. We conduct preliminary experiments on a Linux server running Ubuntu 22.04 and Linux kernel version 5.15.0. The server has two AMD EPYC 7763 processors, 256 GB RAM, and two NVIDIA A100 GPUs. We use NVIDIA driver version 590.48.01 and CUDA version 13.1.

Results. Table 1 shows the coverage statistics of 11 library samples from cuBLAS [2], which is NVIDIA's proprietary, closed-source library that provides highly-optimized GPU-accelerated implementations of fundamental linear algebra operations. The results indicate that inputs from CUDA library samples [5] achieve relatively low code coverage, with a geometric mean of only 25.98% across the tested library samples. Specifically, while `asum` reaches the highest coverage of 64.29%, `rotm` exhibits the lowest coverage of merely 9.09%. This significant gap suggests a substantial gap in state exploration for complex GPU programs, a challenge we aim to address with our GPU-native fuzzing pipeline.

6 Conclusion

The widening security gap in heterogeneous systems presents a critical ethical concern. Therefore, in this paper, we propose a GPU-native fuzzing pipeline that combines dynamic binary instrumentation with context-sensitive fuzzing to ensure faithful bug detection of CUDA programs. Our design aims to replace existing transformation-based mitigation solutions to achieve more precise bug detection on heterogeneous systems.

Acknowledgments

This work was supported in part by the Center for Ubiquitous Connectivity (CUbiC) and is sponsored by Semiconductor Research Corporation (SRC) and Defense Advanced Research Projects Agency (DARPA) under the JUMP 2.0 program. Results presented in this paper were obtained using Chameleon Cloud [24] supported by the National Science Foundation.

References

[1] [n. d.]. *AMD Product Security*. https://www.amd.com/en/resources/product-security.html

[2] [n. d.]. *cuBLAS | NVIDIA Developer*. https://developer.nvidia.com/cublas

[3] [n. d.]. *CUDA-GDB | NVIDIA Developer*. https://developer.nvidia.com/cuda-gdb

[4] [n. d.]. *CUDA Libraries Documentation*. https://docs.nvidia.com/cuda-libraries/index.html

[5] [n. d.]. CUDA Library Samples. https://github.com/NVIDIA/CUDALibrarySamples.

[6] [n. d.]. Hardware-assisted AddressSanitizer | Android Open Source Project. https://source.android.com/docs/security/test/hwasan. [Online; accessed 1-April-2025].

[7] [n. d.]. *NVIDIA Compute Sanitizer | NVIDIA Developer*. https://developer.nvidia.com/compute-sanitizer

[8] [n. d.]. *NVIDIA CUDA Compiler Driver NVCC*. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/

[9] [n. d.]. *NVIDIA CUTLASS Documentation*. https://docs.nvidia.com/cutlass/latest/

[10] [n. d.]. *NVIDIA Product Security*. https://www.nvidia.com/en-us/product-security/

[11] [n. d.]. *triton-lang/triton-cpu: An experimental CPU backend for Triton*. https://github.com/triton-lang/triton-cpu

[12] [n. d.]. *Triton-San: Toward Precise Debugging of Triton Kernels via LLVM Sanitizers*. https://llvm.org/devmtg/2025-10/slides/technical_talks/lu.pdf

[13] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1032–1043.

[14] Brian Chess and Gary McGraw. 2004. Static analysis for security. *IEEE security & privacy* 2, 6 (2004), 76–79.

[15] Neophytos Christou, Di Jin, Vaggelis Atlidakis, Baishakhi Ray, and Vasileios P. Kemerlis. 2023. IvySyn: Automated Vulnerability Discovery in Deep Learning Frameworks. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 2383–2400. https://www.usenix.org/conference/usenixsecurity23/presentation/christou

[16] William J Dally, Yatish Turakhia, and Song Han. 2020. Domain-specific hardware accelerators. *Commun. ACM* 63, 7 (2020), 48–57.

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

[18] Gregory J. Duck and Roland H. C. Yap. 2018. EffectiveSan: type and memory error detection using dynamically typed C/C++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 181–195. https://doi.org/10.1145/3192366.3192388

[19] Christopher Erb and Joseph L Greathouse. 2018. Clarmor: A dynamic buffer overflow detector for opencl kernels. In *Proceedings of the International Workshop on OpenCL*. 1–2.

[20] Andrea Fioraldi, Dominik Christian Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*, Yuval Yarom and Sarah Zennou (Eds.). USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[21] Patrice Godefroid. 2020. Fuzzing: Hack, art, and science. *Commun. ACM* 63, 2 (2020), 70–76.

[22] Yanan Guo, Zhenkai Zhang, and Jun Yang. 2024. {GPU} Memory Exploitation for Fun and Profit. In *33rd USENIX Security Symposium (USENIX Security 24)*. 4033–4050.

[23] Shinnung Jeong, Chihyo Ahn, Huanzhi Pu, Jisheng Zhao, Hyesoon Kim, and Blaise Tine. 2025. Inside VOLT: Designing an Open-Source GPU Compiler. *arXiv preprint arXiv:2511.13751* (2025).

[24] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbah, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 219–233. https://www.usenix.org/conference/atc20/presentation/keahey

[25] Robert Tjarko Lange, Aaditya Prasad, Qi Sun, Maxence Faldor, Yujin Tang, and David Ha. 2025. *The AI CUDA engineer: Agentic CUDA kernel discovery, optimization and composition*. Technical Report. Technical report, Sakana AI, 02 2025.

[26] Jaewon Lee, Euijun Chung, Saurabh Singh, Seonjin Na, Yonghae Kim, Jaekyu Lee, and Hyesoon Kim. 2025. Let-Me-In:(Still) Employing In-pointer Bounds Metadata for Fine-grained GPU Memory Safety. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1648–1661.

[27] Charles E Leiserson, Neil C Thompson, Joel S Emer, Bradley C Kuszmaul, Butler W Lampson, Daniel Sanchez, and Tao B Schardl. 2020. There's plenty of room at the Top: What will drive computer performance after Moore's law? *Science* 368, 6495 (2020), eaam9744.

[28] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 475–485.

[29] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the art. *IEEE Transactions on Reliability* 67, 3 (2018), 1199–1218.

[30] Jinkun Lin, Ziheng Jiang, Zuquan Song, Sida Zhao, Menghan Yu, Zhanghan Wang, Chenyuan Wang, Zuocheng Shi, Xiang Shi, Wei Jia, et al. 2025. Understanding Stragglers in Large Model Training Using What-if Analysis. *arXiv preprint arXiv:2505.05713* (2025).

[31] Hao Ling, Heqing Huang, Chengpeng Wang, Yuandao Cai, and Charles Zhang. 2024. GIANTSAN: Efficient Memory Sanitization with Segment Folding. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir (Eds.). ACM, 433–449. https://doi.org/10.1145/3620665.3640391

[32] Hongyi Lu, Fengwei Zhang, Zhenkai Zhang, Shuai Wang, and Yanan Guo. 2026. CuSafe: Capturing Memory Corruption on NVIDIA GPUs. In *Proceedings of the 35th USENIX Security Symposium (USENIX Security 2026)*. USENIX Association.

[33] Nicholas D Matsakis and Felix S Klock. 2014. The rust language. In *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology*. 103–104.

[34] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for c. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 245–258. https://doi.org/10.1145/1542476.1542504

[35] Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. 2025. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517* (2025).

[36] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. {MoonShine}: Optimizing {OS} fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 18)*. 729–743.

[37] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. Vuzzer: Application-aware evolutionary fuzzing. In *2017 Network and Distributed System Security (NDSS) Symposium:[Proceedings]*. Internet Society, 1–14.

[38] Jonas Roels, Adriaan Jacobs, and Stijn Volckaert. 2025. CUDA, Woulda, Shoulda: Returning Exploits in a SASS-y World. In *Proceedings of the 18th European Workshop on Systems Security*. 40–48.

[39] Kosta Serebryany. 2016. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 157–157.

[40] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Annual Technical Conference, USENIX ATC 2012, Boston, MA, USA, June 13-15, 2012*, Gernot Heiser and Wilson C. Hsieh (Eds.). USENIX Association, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[41] Abhishek Shah, Dongdong She, Samanway Sadhu, Krish Singal, Peter Coffman, and Suman Jana. 2022. Mc2: Rigorous and efficient directed greybox fuzzing. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2595–2609.

[42] Gabriel Sherman and Stefan Nagy. 2025. No Harness, No Problem: Oracle-guided Harnessing for Auto-generating C API Fuzzing Harnesses. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 775–775.

[43] Saurabh Singh, Ruobing Han, Jaewon Lee, Seonjin Na, Yonghae Kim, Taesoo Kim, and Hyesoon Kim. 2026. CuFuzz: Hardening CUDA Programs through Transformation and Fuzzing. *arXiv preprint arXiv:2601.01048* (2026).

[44] Tyler Sorensen and Heidy Khlaaf. 2024. LeftoverLocals: Listening to LLM responses through leaked GPU local memory. *arXiv preprint arXiv:2401.16603* (2024).

[45] Michael B. Sullivan, Mohamed Tarek Ibn Ziad, Aamer Jaleel, and Stephen W. Keckler. 2023. Implicit Memory Tagging: No-Overhead Memory Safety Using Alias-Free Tagged ECC. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 67, 13 pages. https://doi.org/10.1145/3579371.3589102

[46] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 48–62.

[47] Mohamed Tarek Ibn Ziad, Sana Damani, Aamer Jaleel, Stephen W. Keckler, and Mark Stephenson. 2023. cuCatch: A Debugging Tool for Efficiently Catching Memory Safety Violations in CUDA Applications. *Proc. ACM Program. Lang.* 7, PLDI, Article 111 (jun 2023), 24 pages. https://doi.org/10.1145/3591225

[48] Thomas N Theis and H-S Philip Wong. 2017. The end of moore's law: A new beginning for information technology. *Computing in science & engineering* 19, 2 (2017), 41–50.

[49] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W Keckler. 2019. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 372–383.

[50] Jacob Wahlgren, Gabin Schieffer, Ruimin Shi, Edgar A León, Roger Pearce, Maya Gokhale, and Ivy Peng. 2025. Dissecting CPU-GPU unified physical memory on AMD MI300A APUs. In *2025 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 368–380.

[51] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. 2016. gpucc: an open-source GPGPU compiler. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. 105–116.

[52] Zheng Yu, Ganxiang Yang, and Xinyu Xing. 2024. ShadowBound: Efficient Heap Memory Protection Through Advanced Metadata Management and Customized Compiler Optimization. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, Davide Balzarotti and Wenyuan Xu (Eds.). USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/yu-zheng

[53] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. 2021. SANRAZOR: Reducing Redundant Sanitizer Checks in C/C++ Programs. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, Angela Demke Brown and Jay R. Lorch (Eds.). USENIX Association, 479–494. https://www.usenix.org/conference/osdi21/presentation/zhang

[54] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu. 2022. Debloating Address Sanitizer. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 4345–4363. https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-yuchen

[55] Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Huazuo Gao, Jiashi Li, Liyue Zhang, Panpan Huang, Shangyan Zhou, Shirong Ma, et al. 2025. Insights into deepseek-v3: Scaling challenges and reflections on hardware for ai architectures. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. 1731–1745.

[56] Wenxin Zheng, Bin Xu, Jinyu Gu, and Haibo Chen. 2025. {SAVE}:{Software-Implemented} Fault Tolerance for Model Inference against {GPU} Memory Bit Flips. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 1585–1604.

[57] Yuhao Zhou, Peng Jia, Jiayong Liu, and Ximing Fan. 2025. Fuzz4Cuda: Fuzzing Your NVIDIA GPU Libraries Through Debug Interface. *Computers & Security* (2025), 104754.

[58] Jiaxun Zhu, Minghao Lin, Tingting Yin, Zechao Cai, Yu Wang, Rui Chang, and Wenbo Shen. 2024. CrossFire: Fuzzing macOS Cross-XPU Memory on Apple Silicon. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 3749–3762.

[59] Ruofan Zhu, Ganhao Chen, Wenbo Shen, Lyuye Zhang, Dakun Shen, Rui Chang, and Yanan Guo. [n. d.]. Demystifying and Exploiting ASLR on NVIDIA GPUs. ([n. d.]).

[60] Mohamed Tarek Ibn Ziad, Sana Damani, Mark Stephenson, Stephen W Keckler, and Aamer Jaleel. 2025. GPUArmor: A Hardware-Software Co-design for Efficient and Scalable Memory Safety on GPUs. *arXiv preprint arXiv:2502.17780* (2025).