

¹ SafeCUDA: Software-only Memory Safety for GPUs via PTX Guards and Managed Shadow Metadata Cache

³ Kiran P. Das  ¹, Navin Kumar A. O.  ¹, Anirudh Sridhar  ¹, and Manju G.  ¹

⁵ 1 School of Computer Science and Engineering, Vellore Institute of Technology (VIT) University,
⁶ Chennai Campus, Tamil Nadu, India

DOI: [10.xxxxxx/draft](https://doi.org/10.xxxxxx/draft)

Software

- ⁷ ▪ [Review](#) 
- ⁸ ▪ [Repository](#) 
- ⁹ ▪ [Archive](#) 

Editor: [Open Journals](#) 

Reviewers:

- ¹¹ ▪ [@openjournals](#)

Submitted: 01 January 1970

Published: unpublished

License

Authors of papers retain copyright¹⁰ and release the work under a¹¹ Creative Commons Attribution 4.0¹² International License ([CC BY 4.0](#)).¹³

⁷ Summary

⁸ Memory safety remains as one of the largest and most difficult to address concerns in the world
⁹ of programming. This is the case especially for CUDA-based applications that operate at large
¹⁰ and distributed scale. Errors such as Buffer Overflow, Use-after-free, Double-free, and race
¹¹ conditions can lead to silent data corruption, unexpected behaviour or complete application
¹² failure, and most importantly, memory leaks can lead to many security issues with varying
¹³ impact. While mature CPU-side tools such as AddressSanitizer ([Serebryany et al., 2012](#)) and
¹⁴ Valgrind ([Nethercote & Seward, 2007](#)) exist, equivalent solutions for GPUs are limited because
¹⁵ of CUDA's highly decoupled architecture, which often requires source code modifications or
¹⁶ specialized hardware to achieve memory safety.

¹⁷ SafeCUDA introduces a software-only memory-safety enforcement system for such CUDA
¹⁸ applications, designed to detect and prevent memory-safety related errors without modifying
¹⁹ GPU drivers or hardware, or the source code of the project itself. The proposed solution
²⁰ proposes a two-tier validation system, with both the compile-time and run-time processes that
²¹ validate and check for memory safety issues.

²² Experimental evaluation shows that SafeCUDA effectively identifies and reports memory safety
²³ violations while maintaining low runtime and memory overhead. This, combined with the fact
²⁴ that there is no source code modification or specialized hardware required for the proposed
²⁵ system to function, show that a software-only memory protection system is feasible and
²⁶ achievable for CUDA, thus providing a valuable debugging and verification tool for GPU
²⁷ developers. This work contributes toward safer and more reliable GPU software development
²⁸ by bridging the gap between performance-driven parallel programming and robust memory
²⁹ safety guarantees.

³¹ Statement of need

³² Graphics Processing Units (GPUs) have become a fundamental component in modern com-
³³ puting, enabling high performance execution of parallel workloads across fields such as Deep
³⁴ Learning, Data Analytics, Scientific Simulation, etc. However, as applications and projects
³⁵ scale to higher levels of complexity, ensuring memory safety becomes a bigger and bigger
³⁶ challenge, due to the highly decoupled nature of CUDA ([Miele, 2016](#)). The split host-device
³⁷ architecture in CUDA brings persistent challenges for developers targeting GPUs to ensure
³⁸ safe memory operations. Prior tools, cuCatch ([Ziad et al., 2023](#)) and GPUArmour ([Tarek Ibn
39 Ziad et al., 2025](#)), require specific hardware or a runtime overhead. Software-only solutions,
⁴⁰ mostly used today, are bound to particular hardware generations and rewriting of the kernels.

41 SafeCUDA addresses this need by providing a lightweight, drop-in memory safety framework
 42 that works transparently with any existing CUDA project. It offers both spatial and temporal
 43 safety through the validation of every GPU memory access against its metadata recorded at
 44 the time of allocation. Operating entirely through interposition and PTX instrumentation,
 45 SafeCUDA integrates seamlessly with existing build systems and workflows, providing an
 46 accessible path to safer GPU software.

47

48 Implementation and architecture

49 SafeCUDA has a dual-phase model: runtime interposition and compile-time instrumentation.

- 50 1. Runtime layer: A preloaded shared library (`libsafecuda.so`) is added using `LD_PRELOAD`
 51 which hooks CUDA runtime APIs such as `cudaMalloc`, `cudaFree`, and `cudaLaunchKernel`.
 52 It maintains a shadow metadata table storing base addresses, block sizes, and flags for
 53 each allocation. Every access during execution is checked in parallel on the GPU against
 54 this metadata to ensure validity. (Henriksen, 2021; Lee et al., 2022)
- 55 2. Compile-time layer: A compiler wrapper, `sf-nvcc`, intercepts PTX code emitted by the
 56 NVIDIA CUDA Compiler and injects bounds-checking instructions for all global memory
 57 operations. The wrapper preserves normal build semantics while ensuring every memory
 58 load or store call passes through a validation step.

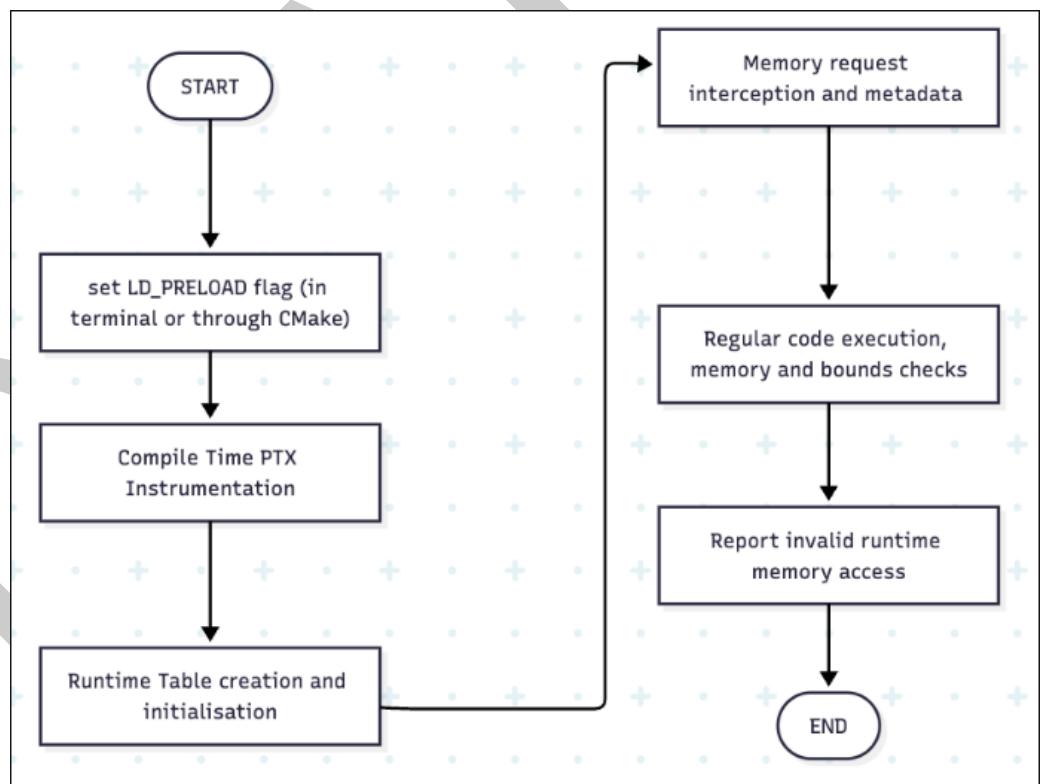


Figure 1: Run-time `sf-nvcc` execution

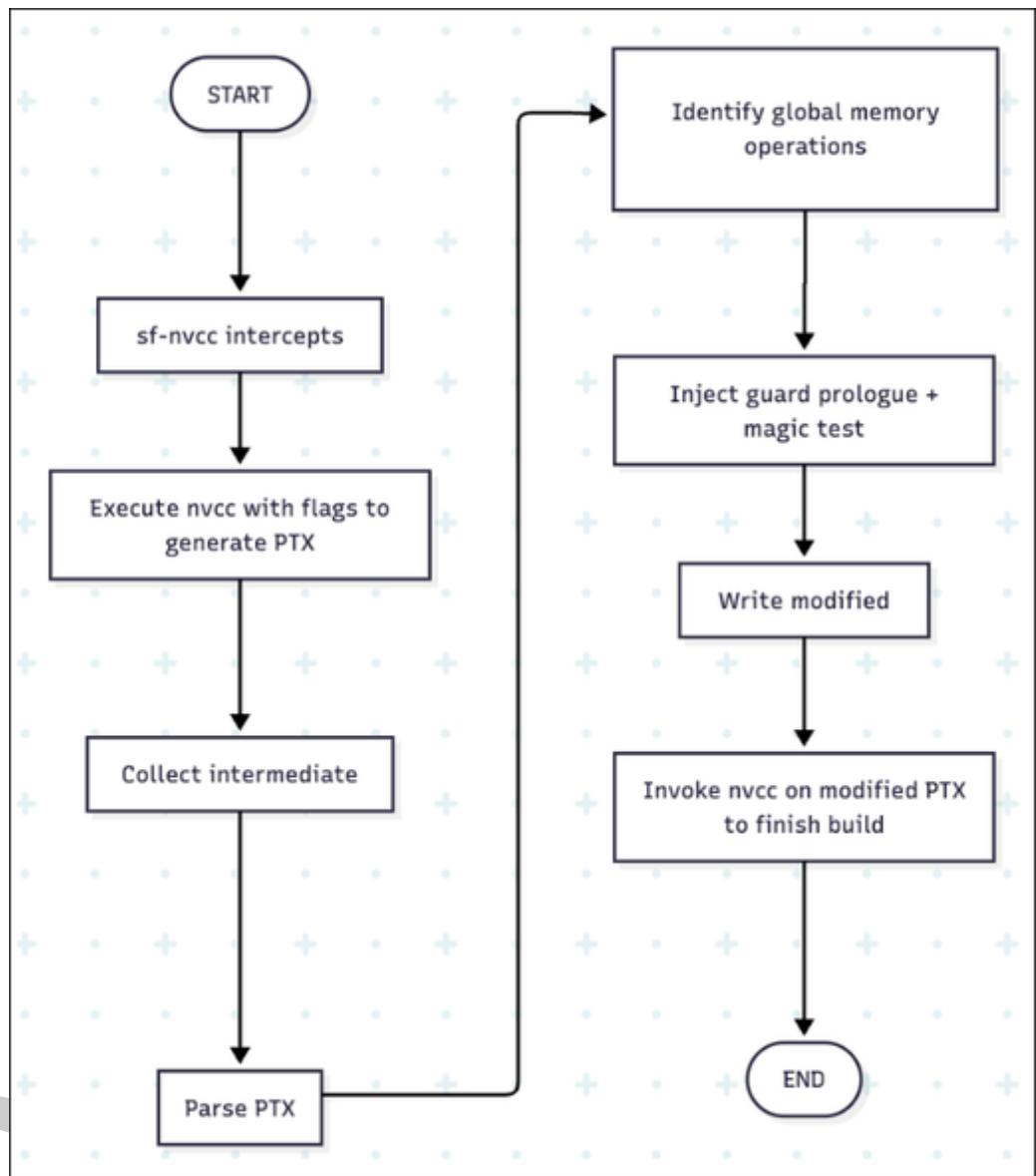


Figure 2: Compile-time sf-nvcc execution and PTX instrumentation

⁵⁹ A compact 16-byte header preceding each allocation links memory blocks to metadata entries.
⁶⁰ Violations are recorded atomically in a reserved table entry and surfaced as a runtime exception
⁶¹ at the next synchronization point, mainly `cudaDeviceSynchronize`.

⁶²

⁶³ Performance and validation

⁶⁴ Evaluation was performed on NVIDIA GTX 1650 Ti and RTX 3060 Ti GPUs with CUDA
⁶⁵ 12.9. The benchmarking suite was implemented in Python for modularity and reproducibility,
⁶⁶ orchestrates dual builds: one using vanilla nvcc, the other using sf-nvcc instrumentation. Each
⁶⁷ build path targets a spectrum of test kernels chosen to represent both familiar and edge-case
⁶⁸ usage scenarios.

⁶⁹ Each test is subject to 5 warm-up runs, discarded to mitigate initialization bias, followed by

500 precise, measured iterations to build robust averages and percentile statistics. The results
 are stored in CSV format for further analysis and visualization.

Benchmarks encompass:

1. Large vector operations, emulating scientific workloads with contiguous memory strides
2. Parallel sum reduction, capturing kernel patterns in HPC
3. Memory copy and scaling, reflecting real-world ML preprocessing pipelines
4. Realistic high-compute kernels, typically found in deep learning training and scientific simulation
5. Synthetic memory stress test (maximizing global memory traffic)
6. Synthetic multi-allocation stress test (fragmented, sparse allocation patterns)

#	Tool	Mean (ms)	Median (ms)	p_low (ms)	p_high (ms)	Tmin (ms)	Tmax (ms)	Overhead %
1	nvcc	300.749	300.037	282.261	322.302	265.386	358.500	1.613
	sfn-vcc	305.601	304.967	286.635	326.219	274.762	353.142	1.613
2	nvcc	303.903	302.927	285.911	326.853	274.559	381.088	0.765
	sfn-vcc	306.229	305.003	288.196	327.579	279.251	352.858	0.765
3	nvcc	303.066	302.582	281.770	326.216	269.097	340.626	4.568
	sfn-vcc	316.910	316.042	296.699	337.887	285.498	367.778	4.568
4	nvcc	327.359	327.188	306.121	351.574	290.211	364.239	0.737
	sfn-vcc	329.771	328.946	307.766	354.655	299.902	373.471	0.737
5	nvcc	312.259	310.644	292.855	336.859	281.209	379.072	11.068
	sfn-vcc	346.821	345.683	321.741	373.270	309.254	456.770	11.068
6	nvcc	355.931	351.681	330.349	401.468	311.496	427.474	17.419
	sfn-vcc	417.930	418.180	393.012	444.448	382.542	467.195	17.419

Average overhead remained below 5% for realistic programs and up to 17% under extreme synthetic tests with excessive allocations. SafeCUDA successfully caught all induced violations, including out-of-bounds and use-after-free errors, demonstrating accurate detection and strong consistency across multiple architectures.

Discussion

SafeCUDA achieves strong memory safety guarantees entirely in software, bridging the gap between high-performance GPU programming and reliable error detection. Its PTX-level integration and LD_PRELOAD runtime interposition make it both portable and transparent, requiring no developer-side code changes.

Limitations include dependence on Linux environments and potential performance degradation in highly allocation-heavy synthetic workloads. Future work includes extending coverage to shared and texture memory spaces, adding multi-GPU synchronization, and optimizing metadata search using hybrid indexing.

95 Acknowledgements

96 The authors thank Dr. Manju G, Associate Professor, School of Computer Science and
97 Engineering, VIT Chennai Campus, for her guidance and supervision.

98

99 References

- 100 Henriksen, T. (2021). Bounds checking on GPU. *International Journal of Parallel Programming*,
101 49, 1–20. <https://doi.org/10.1007/s10766-021-00703-4>
- 102 Lee, J., Kim, Y., Cao, J., Kim, E., Lee, J., & Kim, H. (2022). Securing GPU via region-based
103 bounds checking. *Proceedings of the 49th Annual International Symposium on Computer
104 Architecture (ISCA '22)*, 27–41. <https://doi.org/10.1145/3470496.3527420>
- 105 Miele, A. (2016). Buffer overflow vulnerabilities in CUDA: A preliminary analysis. *Journal of
106 Computer Virology and Hacking Techniques*, 12(2), 113–120.
- 107 Nethercote, N., & Seward, J. (2007). Valgrind: A framework for heavyweight dynamic binary
108 instrumentation. *Proceedings of the 28th ACM SIGPLAN Conference on Programming
109 Language Design and Implementation (PLDI '07)*, 89–100. <https://doi.org/10.1145/1250734.1250746>
- 111 Serebryany, K., Bruening, D., Potapenko, A., & Vyukov, D. (2012). AddressSanitizer: A fast
112 address sanity checker. *Proceedings of the 2012 USENIX Annual Technical Conference
113 (USENIX ATC '12)*, 28–39.
- 114 Tarek Ibn Ziad, M., Damani, S., Stephenson, M., Keckler, S. W., & Jaleel, A. (2025).
115 GPUArmor: A hardware-software co-design for efficient and scalable memory safety on
116 GPUs. *arXiv Preprint*. <https://doi.org/10.48550/arXiv.2502.17780>
- 117 Ziad, M., Damani, S., Jaleel, A., Keckler, S., & Stephenson, M. (2023). cuCatch: A debugging
118 tool for efficiently catching memory safety violations in CUDA applications. *Proceedings of
119 the ACM on Programming Languages*, 7, 124–147. <https://doi.org/10.1145/3591225>