EraseMe: A Defense Mechanism against Information Leakage exploiting GPU Memory

Hongyu Fang, Miloš Doroslovački, Guru Venkataramani
{hongyufang_ee,doroslov,guruv}@gwu.edu
The George Washington University
Washington, DC, United States

ABSTRACT

Graphics Processing Units (GPU) play a major role in speeding up computational tasks of the users, especially in applications such as high volume text and image processing. Recent works have demonstrated the security problems associated with GPU that do not erase the remnant data left behind by previous applications prior to OS context switching. In these attacks, adversaries are able to allocate their memory region on the same memory region used by previous applications and are able to steal their secrets. To overcome this problem, one needs to erase every modified memory page, and this process incurs very high latencies (order of several seconds to even minutes). In this work, we propose EraseMe, a lightweight, content-aware memory-cleansing framework that identifies and erases the sensitive memory pages left behind by victim applications. Our preliminary evaluation shows that EraseMe is able to increase the difficulty of image reconstruction by over 10× for the attacker.

KEYWORDS

Data Remanence attacks, Memory Security, Cyber-Defense

ACM Reference Format:
Hongyu Fang, Miloš Doroslovački, Guru Venkataramani. 2019. EraseMe: A Defense Mechanism against Information Leakage exploiting GPU Memory. In *Great Lakes Symposium on VLSI 2019 (GLSVLSI '19), May 9–11, 2019, Tysons Corner, VA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3299874.3318027

1 INTRODUCTION

Graphics Processing Units (GPU) are widely used in computer systems for acceleration of text processing and image rendering, machine learning and other computationally intensive applications. Recent work [2, 10, 13, 26] demonstrate that the GPU does not zero out physical memory pages in GPU global memory during page allocation owing to high performance overheads. Such remnant data contain rich information about the last application context such as websites browsed [10], or rendered images [26].

A straightforward solution to avoiding theft of remnant data in GPU memory is to blindly erase every memory page before it is allocated to the next context. However, as demonstrated by Pietro et al. [13], zeroing-out of 512MB GPU memory takes around 20 seconds which may negate the usefulness of GPUs meant to accelerate workload performance.

Prior work, such as Silent Shredder [1], was proposed to shred remnant data efficiently) without actually writing out zeroes in non-volatile memories and address their inherent write endurance issue. However, this mechanism requires use of specialized hardware for counter mode encryption that encrypt/decrypt memory pages, along with tracking page-level (major) counters and resetting of corresponding minor counters within the page to zero.

We note that, not all remnant data in GPU memory is vulnerable to giving away sensitive information about the users. Hence, one could potentially modify just secret-crunching regions of GPU-based applications and add extra functions to erase secrets at the end of executing the GPU kernel. However, this solution requires reprogramming several existing applications and relying on third party software developers to provide higher security to user data. Also, in many scenarios, the secrets of victim applications are not uniformly located in all memory pages. For instance, the information content in a PDF document is primarily concentrated in the text portion, and the white background region has no content. Through cleverly leveraging such information, we could aim for more targeted erasure of contents from GPU memory.

In this paper, we propose EraseMe, that uses simple histogram counter based information to target memory pages. The Operating System would erase these *targeted* memory pages with secrets before allocating them to a new context. With this design, the information leakage through GPU memory could be avoided for user-desired content while having low-cost implementation.

The main contributions of our work are:

- We propose EraseMe, a new security framework that guards against information leakage attacks by identifying GPU pages with high information content, and selectively erasing such pages before allocating them to the next context.
- We demonstrate the efficacy of our design in avoiding information leakage through data remanence using Tesseract report rendering benchmarks [16]. Our experimental results demonstrate that the difficulty of attackers to reconstruct information is increased by over 10×.

2 BACKGROUND

In this section, we study background on GPU and memory security.

2.1 GPU Execution Model

Execution of OpenCL-based GPU programs typically involve two part: kernels and host. The host executes on the host device. The

GLSVLSI '19, May 9–11, 2019, Tysons Corner, VA, USA

ACM ISBN 978-1-4503-6252-8/19/05…$15.00
https://doi.org/10.1145/3299874.3318027

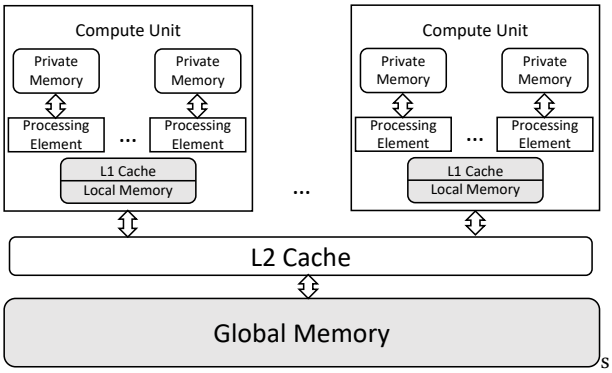

Figure 1: GPU memory hierarchy (Note that the gray memory blocks are not cleaned at the end of kernel execution.)

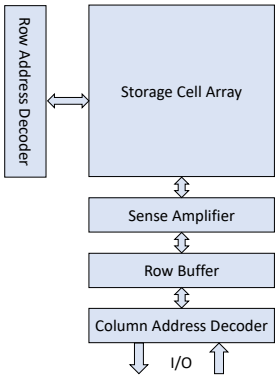

Figure 2: The organization of DRAM

kernels are configured by the host and execute on one or multiple OpenCL machine. In this work, the kernel is executed on GPU. The structure of GPU is shown in Figure 1. The instances of kernel are executed on processing elements, also known as GPU cores. Multiple processing elements form one compute unit. The GPU runs multiple kernels at the same time to accelerate the workloads.

2.2 GPU Memory Hierarchy

The memory hierarchy of GPU is shown in Figure 1. There are four types of memory in GPU: private memory, local memory, global memory and constant memory. Each private memory is attached to one processing element. Private memory is invisible to other processing elements in the same compute unit. Local memory is shared among all processing elements within one compute unit. We note that, in some GPUs, the L1 cache and local memory share the physical hardware and their sizes are configurable by users. The local memory and L1 cache implement coherence protocols among processing elements within one compute unit. The global memory is accessible to every compute unit. The constant memory is a read-only memory region within global memory. The size of constant memory is configured and initialized by the host. The coherence of memory blocks in global memory is not guaranteed.

Memory Type	Vulnerability Window	Size	Reset Latency
Shared Memory	Before End of Context	6MB	0.22ms
Global Memory	After End of Context	2GB	20s

Table 1: GPU memory properties in Nvidia GeForce GT640. Vulnerability window shows when the attacker kernel can directly read the victim kernel's remnant data.

2.3 DRAM Organization

The global memory of GPU is Dynamic Random Access Memory (DRAM) with 2GB to 8GB capacity. The organization of modern DRAM is shown in Figure 2. The stored bits can be located by its row address and column address. The row address firstly arrives to the DRAM. All bits within the row are read to row buffer. Then the column address arrives to row buffer and the content of accessed memory address is sent to memory bus (memory read) or written with the incoming data from memory bus (memory write). After the memory access, there are two policies to handle the data in row buffer: open page policy and close page policy. The open page policy does not precharge the row buffer until a different row in the same memory bank is accessed. The open page policy reduces the access latency for the same row (memory page) but increase the access latency of the row buffer miss. The close page policy recharge the row buffer right after the memory access. The close page policy makes the access latency of row buffer hit and row buffer miss equal which is slower than row buffer hit in open page policy and faster than row buffer miss in open page policy. The commercial DRAM may use both policies.

2.4 Remnant Data in GPU Memory

Prior work have demonstrated the security vulnerabilities arising from non-initilization in modern GPU memory during page allocation [10]. The local memory and private memory would not be erased at the end of context execution. The data in local and private memory are accessible for the host of next executing kernels before the host of last kernel ends its context. The remnant data in global memory are not erased during the page deallocation which makes the next context capable to access data from last context directly.

The attack on GPU memory can be categorized into two classes based on the different attack targets: local/private memory attack and global memory attack. The properties of these two level of GPU memories are shown in Table 1. To launch local/private memory attack, the attacker kernel keeps dumping memory from local/private memory after the end of victim kernel and before the end of victim context. To launch global memory attack, the attacker keeps requesting a memory chunk equal to the size of free global memory. Once some parts of global memory are released by victim, the data contents are visible to the attacker.

A straightforward solution to solve the remnant memory problem in GPU is to erase memory pages after the end of kernels or context. As shown in Table 1, for private/local memory, the solution incurs lower overhead because the size of private and local memory is usually 16KB or 64KB per core. It takes less than 20ms to erase entire private/local memory after the end of kernel. However, the global memory of modern GPU is usually at gigabyte level. The

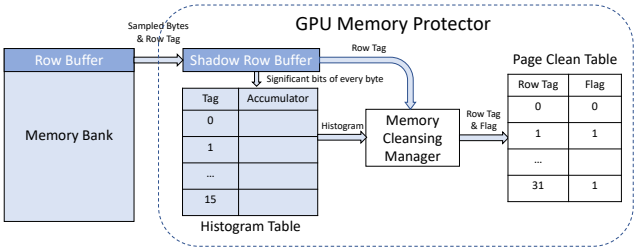

Figure 3: EraseMe design overview (The shaded components are implemented through hardware while the white components are implemented in software.)

zero-out of entire global memory takes from 20 seconds to 80 seconds for different machines [13]. Considering the fact that GPU is a performance-oriented machine, the high latency during context switch is undesirable.

2.5 Prior Studies in Memory Security

Security problem in CPU memory hierarchy is widely studied. Prior works demonstrated the information leakage problem through side and covert channels in memory and CPU caches [22, 25]. Previous studies have proposed hardware designs to defend main memory [14] and caches [3, 7–9, 18, 23] against information leakage attacks. Other works that profile software, detect memory bugs and increase software robustness [4, 6, 11, 12, 15, 19–21, 24] can be used in tandem to improve overall system security.

3 THREAT MODEL

After getting the remnant data of victim kernel, the attacker needs to reconstruct user-intelligible, useful information. Usually this step would happen through computer recognition algorithms because the remnant data is large and is in byte format, which leaves no pattern for human eyes to recognize. In this paper, the attacker first locates the position of the secret from all remnant data using Fast Fourier Transform. Then the attacker recognizes the text in the image using Optical Character Recognition (OCR) or identify the images using pre-trained machine learning models.

4 DESIGN

In this section, we propose the design of EraseMe, a low-overhead and user-configurable solution to information leakage exploiting GPU memory. EraseMe adds a hardware-performance-counter-like component to GPU memory, and collects statistics about information inside every memory page. The user can configure the system to identify valuable information inside pages, and can request the OS to erase some targeted memory pages with valuable content.

The structure of EraseMe is shown in Figure 3. A shadow row buffer is attached to each row buffer of memory bank in GPU global memory. The shadow row buffer samples and copies the bits stored in row buffer for histogramming. The memory cleansing manager implements user-configurable rules to analyze histogram data, and decide whether the memory page needs to be cleaned after deallocation. The memory cleansing manager updates the decision in the *page-clean* table which contains the memory page addresses and a flag indicating whether the page was deemed sensitive (using user-configured rules). Among all components in EraseMe, the shadow row buffer and histogrammer are implemented in hardware and added to every row buffer in GPU global memory. The memory cleansing manager is an application which reads histograms periodically and decides whether to clean the page after deallocation. The page-clean table is implemented in software as a protected data structure by the Operating System, which will access it upon page deallocation. The memory page with sensitive information would be cleaned before being allocated to the next process.

5 EXPERIMENTAL SETUP

We run experiments on AMD Radeon RX 470 GPU and analyze the remnant data in GPU global memory. We run Document Viewer as the victim. It renders reports from Tesseract Benchmarks [16] in gray scale.

5.1 Attacker and Information Extraction

The goal of attacker is to reconstruct the texts in rendered reports from remnant data in GPU as shown in prior work [26]. The attacker converts reconstructed images to texts using Tesseract OCR Engine [16]. The result of attack is evaluated by the accuracy of reconstruction. The reconstruction score is computed using FuzzyWuzzy [5], a robust string matching algorithm by comparing the reconstructed text and the original text. The goal of EraseMe is to stop the attacker from reconstructing the texts in reports by removing the memory pages containing the texts. To identify texts among all remnant data, we note that the entropy of memory pages should be high because memory pages with just plain white or black pixels do not contain any reconstructable information.

6 EVALUATION

Figure 4 shows the text image before and after the initialization by EraseMe. The EraseMe removes 20% of memory pages with highest entropy where each memory page contains 4096 pixels. Before being erased, the text in image is clear and could be accurately identified by Optical Character Recognition (OCR). After the cleansing of EraseMe, the text in image is not recognizable for human eye. Besides, an automated OCR cannot reconstruct the original text either. The output of OCR engine is full of meaningless characters.

Figure 5 shows the relationship between percentage of high entropy pages removed by EraseMe and scores obtained from reconstruction results. The score evaluates the information reconstructed by the attacker by comparing the output of OCR engine and the original texts using FuzzyWuzzy [5] algorithm. The highest output of the algorithm is 100, which indicates that the two compared strings are identical. Our results show that, by removing only 20% memory pages with highest entropy, EraseMe can reduce the reconstruction score of attacker from 80 to less than 10 where the attacker can barely get any useful information. In other words, EraseMe increases the difficulty of page reconstruction by over 10× for the attacker. With 40% memory pages being removed, the EraseMe can make 99% of information not recognizable to the attacker.

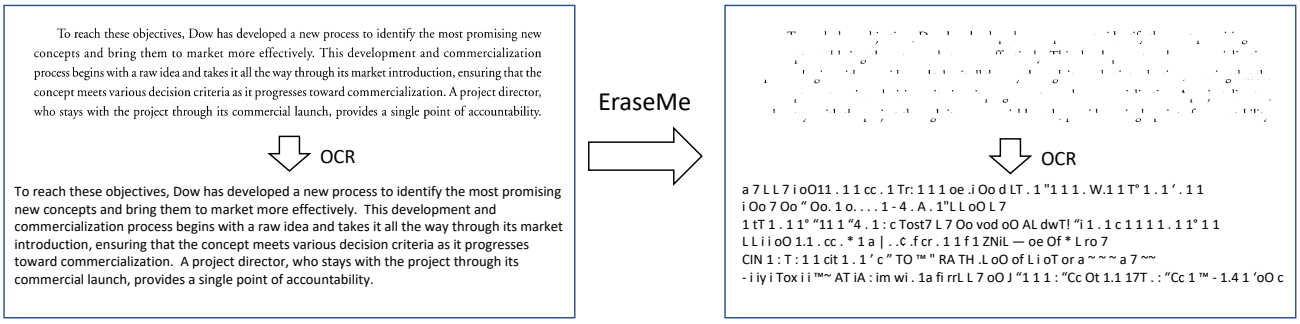

Figure 4: Erasure efficacy of EraseMe. The upper left is the original image containing 3300 x 500 pixels. The upper right is the image cleaned by EraseMe. The bottom parts show the result of Optical Character Recognition Engine.

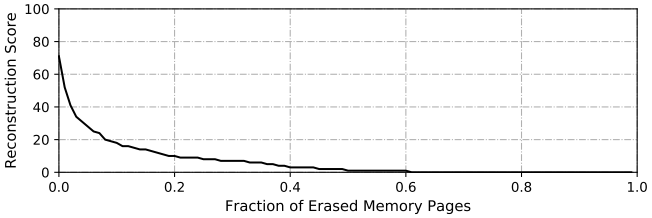

Figure 5: The fraction of initialized memory page and the attacker's reconstruction score.

7 CONCLUSION

In this paper, we proposed EraseMe, a light-weight framework to identify and remove the sensitive data in GPU global memory to prevent potential attacker to reconstruct the remnant data from previous applications.

8 ACKNOWLEDGEMENT

This manuscript is based on work supported by the US National Science Foundation under grant CNS-1618786, and Semiconductor Research Corp. (SRC) contract 2016-TS-2684.

REFERENCES

[1] Amro Awad, Pratyusa Manadhata, Stuart Haber, Yan Solihin, and William Horne. Silent shredder: Zero-cost shredding for secure non-volatile main memory controllers. In *ACM SIGARCH Computer Architecture News*, volume 44, 2016.

[2] Xavier Bellekens, Greig Paul, James M Irvine, Christos Tachtatzis, Robert C Atkinson, Tony Kirkham, and Craig Renfrew. Data remanence and digital forensic investigation for cuda graphics processing units. In *FIP/IEEE International Symposium on Integrated Network Management*, 2015.

[3] Jie Chen and Guru Venkataramani. Cc-hunter: Uncovering covert timing channels on shared processor hardware. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 216–228. IEEE, 2014.

[4] Jie Chen, Guru Venkataramani, and H Howie Huang. Repram: Re-cycling pram faulty blocks for extended lifetime. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12. IEEE, 2012.

[5] A Cohen. Fuzzywuzzy: Fuzzy string matching in python, 2011.

[6] Ioannis Doudalis, James Clause, Guru Venkataramani, Milos Prvulovic, and Alessandro Orso. Effective and efficient memory protection using dynamic tainting. *IEEE Transactions on Computers*, 61(1):87–100, 2012.

[7] Hongyu Fang, Sai Santosh Dayapule, Fan Yao, Miloš Doroslovački, and Guru Venkataramani. A noise-resilient detection method against advanced cache timing channel attack. In *2018 52nd Asilomar Conference on Signals, Systems, and Computers*, pages 237–241. IEEE, 2018.

[8] Hongyu Fang, Sai Santosh Dayapule, Fan Yao, Miloš Doroslovački, and Guru Venkataramani. Prefetch-guard: Leveraging hardware prefetches to defend against cache timing channels. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 187–190. IEEE, 2018.

[9] Hongyu Fang, Sai Santosh Dayapule, Fan Yao, Miloš Doroslovački, and Guru Venkataramani. Prodact: Prefetch-obfuscator to defend against cache timing channels. *International Journal of Parallel Programming*, pages 1–24, 2018.

[10] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. Stealing webpages rendered on your browser by exploiting gpu vulnerabilities. In *2014 IEEE Symposium on Security and Privacy (SP)*, pages 19–33. IEEE, 2014.

[11] Yongbo Li, Fan Yao, Tian Lan, and Guru Venkataramani. Sarre: semantics-aware rule recommendation and enforcement for event paths on android. *IEEE Transactions on Information Forensics and Security*, 11(12):2748–2762, 2016.

[12] Jungju Oh, Christopher J Hughes, Guru Venkataramani, and Milos Prvulovic. Lime: A framework for debugging load imbalance in multi-threaded execution. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 201–210. ACM, 2011.

[13] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. Cuda leaks: a detailed hack for cuda and a (partial) fix. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(1):15, 2016.

[14] Ali Shafiee, Rajeev Balasubramonian, Mohit Tiwari, and Feifei Li. Secure dimm: Moving oram primitives closer to memory. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 428–440. IEEE, 2018.

[15] Jianli Shen, Guru Venkataramani, and Milos Prvulovic. Tradeoffs in fine-grained heap memory protection. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 52–57. ACM, 2006.

[16] Ray Smith. An overview of the tesseract ocr engine. In *Document Analysis and Recognition, 2007. ICDAR 2007. Ninth International Conference on*, volume 2, pages 629–633. IEEE, 2007.

[17] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[18] Guru Venkataramani, Jie Chen, and Milos Doroslovacki. Detecting hardware covert timing channels. *IEEE Micro*, 36(5):17–27, 2016.

[19] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. Memtracker: An accelerator for memory debugging and monitoring. *ACM Trans. Archit. Code Optim.*, 6(2):5:1–5:33, July 2009.

[20] Guru Venkataramani, Christopher J Hughes, Sanjeev Kumar, and Milos Prvulovic. Deft: Design space exploration for on-the-fly detection of coherence misses. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(2):8, 2011.

[21] Hongfa Xue, Yurong Chen, Fan Yao, Yongbo Li, Tian Lan, and Guru Venkataramani. Simber: Eliminating redundant memory bound checks via statistical inference. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 413–426. Springer, 2017.

[22] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. Are coherence protocol states vulnerable to information leakage? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 168–179. IEEE, 2018.

[23] Fan Yao, Hongyu Fang, Milos Doroslovacki, and Guru Venkataramani. Towards a better indicator for cache timing channels. *arXiv preprint arXiv:1902.04711*, 2019.

[24] Fan Yao, Yongbo Li, Yurong Chen, Hongfa Xue, Tian Lan, and Guru Venkataramani. Statsym: vulnerable path discovery through statistics-guided symbolic execution. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 109–120. IEEE, 2017.

[25] Fan Yao, Guru Venkataramani, and Miloš Doroslovački. Covert timing channels exploiting non-uniform memory access based architectures. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, pages 155–160. ACM, 2017.

[26] Zhe Zhou, Wenrui Diao, Xiangyu Liu, Zhou Li, Kehuan Zhang, and Rui Liu. Vulnerable gpu memory management: towards recovering raw data from gpu. *Proceedings on Privacy Enhancing Technologies*, 2017(2):57–73, 2017.