CSC2311 Project Report

Investigating the Cost of an Authentication-Based Rowhammer Mitigation

Michal Fishkin

10 December, 2023

Introduction

As memory cells reduce in size, voltage can be leaked from one row to another and cause unwanted bit-flips. Rowhammer is a bit-flip attack which takes advantage of this property by continually opening and closing an **aggressor row** to cause bit-flips in neighbouring **victim rows** [1]. This attack can be especially devastating to the efficacy of neural networks, in which only a few bit-flips in weight stored in DRAM can dramatically reduce the accuracy of a deep neural network (DNNs) [2]. An example of the impact of such an attack can be seen in Figure 1, which indicates a performance drop of 75% for ResNet-20 after only 10 bit-flips.

Given the increasing prevalence of DNNs in technology, security is vital to ensure that such applications work safely and as expected. In this work, we investigate the efficiency of one security framework for securing against bit-flip attacks: **message authentication codes**. We parameterize inference efficiency as **arithmetic intensity** and find the parameter that optimizes performance while guaranteeing efficiency.

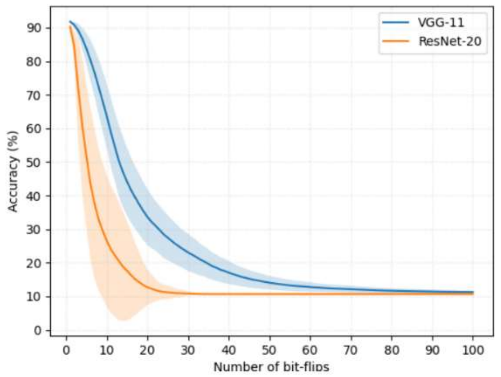

Figure 1: Accuracy of ResNet-20 and VGG-11 plotted against the number of bit-flips from a principled RowHammer Attack. Figure from [1].

Prior Work

The smallest number of activations it takes to flip a bit in neighbouring rows is referred to as the **RowHammer threshold (THR)**. As memory cells have shrunk, so has the THR. This decrease has posed challenges to researchers creating defenses [3].

Prior work in RowHammer has included victim-based mitigations, which issue refreshes to victim rows selectively, and aggressor-based mitigations, which lower the ability of an aggressor row to flip the bits of neighbors. Some victim-based mitigations include counters [4] and probabilistic refreshes [3]. Such defenses have been circumvented by distance-of-n attacks such as Google’s Half-Double Attack wherein mitigations cause additional activations [5]. Aggressor-based defenses such as [6] and [7] can surpass these new attacks, but as the RowHammer threshold decreases, so does their efficiency.

One work that introduces a defense that can withstand future threshold drops is PTGuard [8]. PTGuard protects against the Privilege Escalation Attack by using **Message Authentication Codes (MACs)** that are stored in the Page Frame Number (PFN). These Message Authentication Codes consist of hashes of the original data which are stored in a secure area, and which are compared to the rehashed values at runtime. Such hashes are sensitive to single bitflips and are thereby effective at stopping Privilege Escalation Attacks. For this project, our goal was to implement a similar framework in the context of DNNs and to analyse its performance. All code is available online[1].

Design

Our central assumption for this project is that **any** bit-flip to the original weight is unacceptable. For this reason, we chose to store the hashes of the weights separately from the weights themselves. The implementation of the defense is as follows:

1. Upon loading the weights, primary hashes are taken of a group of N weights.
2. These primary hashes are stored in a separate array in the DNN.
3. Before each inference, a group of N weights is hashed and checked for equivalence.

For the inference performance tests, we employed a simple model which maps images of handwritten digits to numbers zero through nine. The code was found online[2]. The specifications are shown in Table 1.

Metric	**Value**
Model Training Data	MNIST
Inference Number of Images	1000
Weight Data Type	Double (8-byte)
Architecture	
Input Layer Size	784
Hidden Layer Size	300
Output Layer Size	10

Table 1: Specifications of the DNN.

Two hashing functions were investigated: no-op, wherein all the processing for hashes was done, but no hash operation took place, and fasthash, which consists of a hashing algorithm with an 8-byte message[3]. The no-op inference statistics would illustrate how much of the overhead is due to the hashing itself and not of the pre-hash processing.

For each group number N, I ran the inference of the network wherein it hashed N weights at once. Metrics were gathered by running '`perf stat`' for instruction and cycle counts and '`perf mem report`' for memory access counts.

Results

The results below all have an x-axis which shows the number of weights hashed at the same time (N) in that inference cycle. Note that the axis is scaled logarithmically to the power of 2.

Figures 2 and 3 show the normalized instruction counts and memory counts of the hashed inference cycles. In both cases, the counts for the fasthash inference are several times higher than the counts for the no-op. The ratio starts out very high for N=1, but drops as N increases.

[1] The repository can be found at: https://github.com/WoodenPlancks/mnist-from-scratch.
[2] Original MNIST C implementation is at https://github.com/markkraay/mnist-from-scratch.
[3] From https://stackoverflow.com/questions/4340471/.

This indicates that the hash function itself causes the majority of the overhead of the instructions and the memory accesses.

The exact increase caused by the hash function alone is visualized in Figure 4 and 5. These plots show the ratio between the fasthash and no-op values in Figures 2 and 3. Note that the ratio of memory ops is consistently higher than the ratio of the instruction counts.

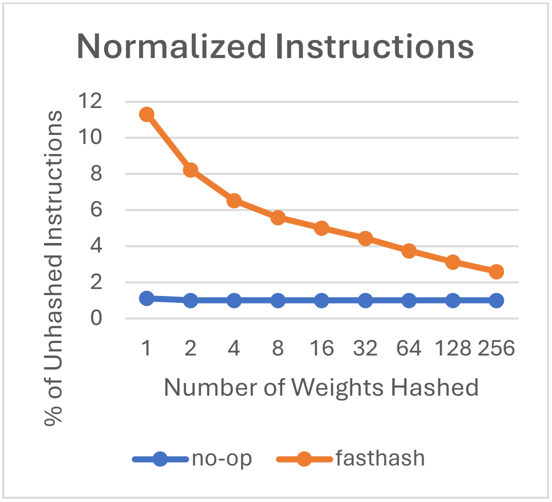

Figure 2: The number of instructions of the hashed inference proportional to the unhashed inference.

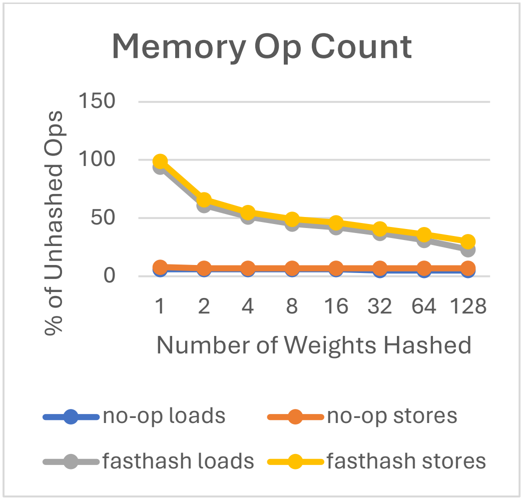

Figure 3: The number of memory accesses of the hashed inference proportional to the unhashed inference.

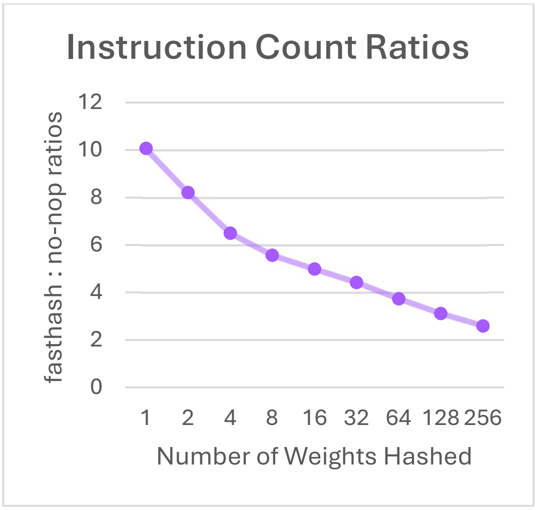

Figure 4: The fasthash:no-op ratio of normalized instruction counts.

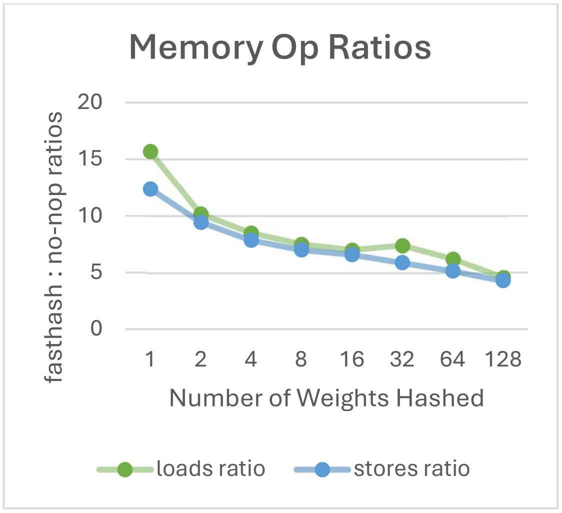

Figure 5: The fasthash:no-op ratio of the normalized memory operation counts.

Arithmetic intensity (AI) is the ratio between the number of instructions and the number of memory accesses. A higher number indicates more 'work' being done for any given memory access, and this is a good indicator of performance. In Figure 6, the AIs of both the fasthash inference and the no-op inference are plotted proportional to the unhashed AI.

The no-op AI hovers at around 80% of the unhashed AI until the 32 weight point, in which it goes up slightly. The fasthash AI stays around 50% with fluctuations of around 8% from the lowest (N = 64) to the highest (N = 2). These fluctuations are better displayed in Figure 6.

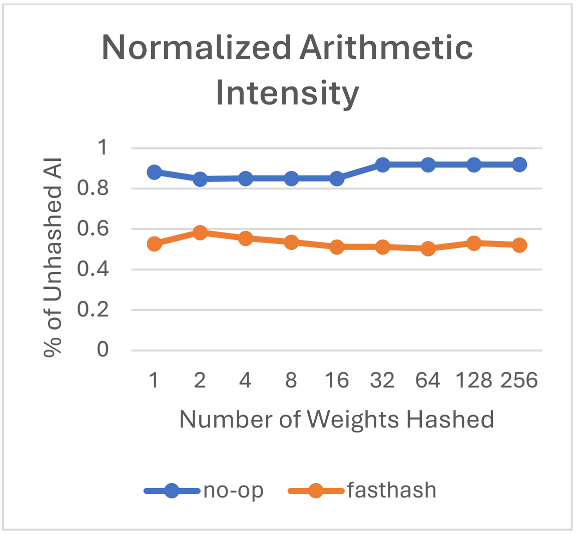

Figure 6: The hashed:unhashed arithmetic intensity ratio of both hash types.

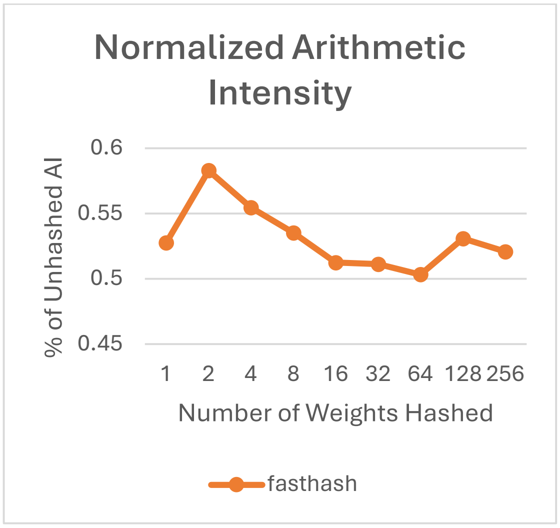

Figure 7: The hashed:unhashed arithmetic intensity ratio of fasthash.

Note that the decrease in AI from fasthash is due to a large increase in memory accesses without a proportional increase in instruction count. This can be seen in Figures 4 and 5, where the proportional increase of the instruction count relative to baseline is ~10x, but the memory access count increases ~15x.

Takeaways and Future Directions

The use of MACs for runtime weight checking provides a strong defense against RowHammer attacks for neural networks. Even one bit-flip could change the hash profoundly enough to ensure detection. In this project, we saw that while the processing for hash functions results in relatively low arithmetic intensity overheads of around 15%, the hash function itself can drop performances to around 50%. In the case of the MNIST DNN specified by Table 1, we see that the **best number of weights to hash at a time for optimizing of arithmetic intensity is N = 2**, with lower numbers greater than N = 1 performing better.

While the model we looked at was relatively simple – a fully connected neural network with a total of 238200 weights - more complex models could benefit from different hashing patterns. Future works can investigate alternative weight-grouping algorithms and alternative hash types. One method of particular interest is AES, wherein memory accesses can be reduced.

References

[1] O. Mutlu and J. S. Kim, "RowHammer: A Retrospective," CoRR, 2019.

[2] F. Yao, A. S. Rakin and D. Fan, "DeepHammer: Depleting the Intelligence of Deep Neural Networks through Targeted Chain of Bit Flips," CoRR, 2020.

[3] O. Mutlu, A. Olgun and A. G. Yağlikçi, "Fundamentally Understanding and Solving RowHammer," in ASPDAC '23, Tokyo, Japan, 2023.

[4] Y. Park, W. Kwon, E. Lee, T. J. Ham, J. H. Ahn and J. W. Lee, "Graphene: Strong yet Lightweight Row Hammer Protection," in 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Athens, Greece, 2020.

[5] A. K. Gruss, J. Juffinger, S. Qazi, Y. Kim, M. Lipp, N. Boichat, E. Shiu and M. N. Daniel, "Half-Double: Hammering From the Next Row Over," 31st USENIX Security Symposium (USENIX Security 22), pp. 3807-3824, 2022.

[6] A. G. Yağlikçi, A. Giray, M. Patel, J. S. Kim, R. Azizi, A. Olgun, L. Orosa, H. Hassan, J. Park, K. Kanellopoulos, T. Shahroodi, S. Ghose and O. Mutlu, "BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows," in 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), 2021.

[7] G. Saileshwar, B. Wang, M. Qureshi and P. J. Nair, "Randomized row-swap: mitigating Row Hammer by breaking spatial correlation between aggressor and victim rows," in Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), New York, NY, USA, 2022.

[8] A. Saxena, G. Saileshwar, J. Juffinger, A. Kogler, D. Gruss and M. Qureshi, "PT-Guard: Integrity-Protected Page Tables to Defend Against Breakthrough Rowhammer Attacks," in 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Porto, Portugal, 2023.