MACMul: Protecting NNs from Rowhammer Attacks with Apache TVM

Luc Baier-Reinio

University of Toronto

lucbr@cs.toronto.edu

Abstract

Deep Neural Networks have gained widespread popularity in recent years. These models can solve complex tasks; however, research has shown that their accuracy can be depleted by a few critical bit-flips using a Rowhammer attack. Apache TVM is a machine learning compiler that allows users to perform custom transformations and other operations on the neural network's computational graph. Leveraging Apache TVM, MACMul is an integrity protection defense that verifies message authentication codes of the model's parameters at inference time to detect tampered weights. MACMul detects ≥ 6 random bit-flips on a DNN with nearly 100% accuracy, and we argue that any single critical bit-flips would also be identified with high probability. We analyze the security and performance guarantees of MACMul and discuss future improvements to this system. This project is open-sourced and can be found at github.com/lbaierreinio/macmul.

1 Introduction

Deep Neural Networks (DNNs) have become ubiquitous in medicine, autonomous vehicles, natural language processing, and more. These models' accuracy depends on their weights, which are optimized to minimize a loss function through extensive training on datasets. Research has shown that the accuracy of these models can be significantly degraded by identifying and flipping a small number of critical bits within these weights.

Unfortunately, progressive bit-flip attacks (PBFAs) do just that. If an adversary has ample knowledge of the architecture of a model (this is a reasonable assumption since many models have been open-sourced), then they may perform a progressive search to identify bits within the model's parameters that have the highest gradient with respect to the loss. Then, the adversary may use Rowhammer techniques to flip these bits and destroy the system's integrity. Most notably, recent studies have shown that a small number of intelligently selected bit-flips may be enough to seriously compromise a system [2].

A set of Integrity Protection defenses against Rowhammer attacks have recently been proposed. PT-Guard stores and verifies MACs within PTEs and detects arbitrary bit-flips [1]. HASHTAG computes a unique signature from the DNN before deployment, which is used to validate the DNN's integrity during run-time [3]. Inspired by these defenses, we propose MACMul, an integrity protection defense that uses Apache TVM and AES-CMAC to detect tampered weights in neural networks. MACMul reliability detects as few as 6 random bit-flips, and we claim that single critical bit-flips will also be detected with high probability. MACMul also provides two tunable parameters which are used to manage the trade-off between security and performance.

2 Background

2.1 Message Authentication Codes

Message Authentication Codes are typically used in computer networks to authenticate the origin and integrity of a message sent from a sender to a receiver. The sender uses the MAC system to generate a tag of the message using an encryption algorithm and a secret key. This tag is appended and sent along with the message to the receiver. The receiver, who knows the same encryption algorithm and secret key, computes a tag of the message and compares it with the tag from the sender. If the tags are the same, and the algorithm provides a low-collision guarantee, the receiver can be confident that the message was not corrupted [7].

2.2 AES-CMAC

CMACs are a family of Message Authentication Codes that are constructed using a block cipher, which encrypts data in fixed-size blocks to produce a cipher text. AES-CMAC is an implementation of a CMAC that uses AES, which is a symmetric block cipher algorithm with a block size of 128 bits and a key of the same length [5] [6]. AES-CMAC accepts a secret key, message of variable length, and length of message, and returns a MAC [8]. AES-CMAC is responsible for chunking the message into 128-bit blocks. Each plaintext block is XORed with the previous ciphertext block before being encrypted with the cipher algorithm.

2.3 Apache TVM

Apache TVM is an open-sourced machine-learning compiler framework that aims to optimize machine-learning models for any hardware platform. TVM has two key features: (1) The ability to compile deep learning models into minimum deployable modules. (2) Infrastructure to automatically generate and optimize models on more backends with better performance [4].

2.4 Previous Works

Defenses against such attacks typically take two forms. Firstly, model enhancement defenses attempt to make the model more robust against bit flips. In other words, the model should be able to withstand a certain fixed number of bit flips and maintain a high degree of accuracy. For example, prior works diffuse the effect of bit-flips across many weights, thereby lessening the impact of a bit-flip [10]. Another model enhancement defense incorporates early-exit classification, which means the model will terminate the inference after a certain number of layers if it is already sufficiently confident in its prediction. Since researchers know which bits/layers are likely to be vulnerable, early exit of the neural network may avoid encountering corrupted weights [9]. The second form of defense is integrity protection. These solutions seek to detect and notify users of bit-flip attacks during the inference process. Firstly, a MAC of the weights will be calculated when the model is first deployed as a ground truth. Then, when the weights are accessed, the MAC is re-computed, and compared against the ground truth. If the results differ, then the user can be notified that the weights have been compromised. From there, the deployed model can be evicted and reloaded with the ground-truth weights [3]. Both defenses have drawbacks. Model enhancement strategies typically do not know whether PBFAs are occurring, and thus cannot notify the user of the attack. Additionally, model enhancement strategies only increase the number of bit-flips required to degrade a model. As Rowhammer attacks become more prevalent with increasing DRAM density, it's worth questioning how long current model enhancement defenses can maintain their strength. Integrity protection defenses tend to incur significant storage and computation overheads, making them less feasible to implement in practice. Furthermore, integrity protection defenses do not usually correct the bit-flips.

3 Methodology

3.1 Rowhammer in Software

Since we are considering defenses, there is no need to implement a real-world Rowhammer attack. We already have direct access to the weights in memory when we are testing. Therefore, we wrote a simple method that accepts a set of parameters, randomly selects a weight matrix, then a random parameter, and flips a random bit. We verify that the Rowhammer attack causes model degradation by repeatedly performing Rowhammer attacks on MLPs trained on the MNIST dataset until accuracy falls below 25%. Most bit-flips do not impact the model accuracy, yet certain bit-flips cause the model accuracy to drop significantly, pointing to the existence of critical bit-flips.

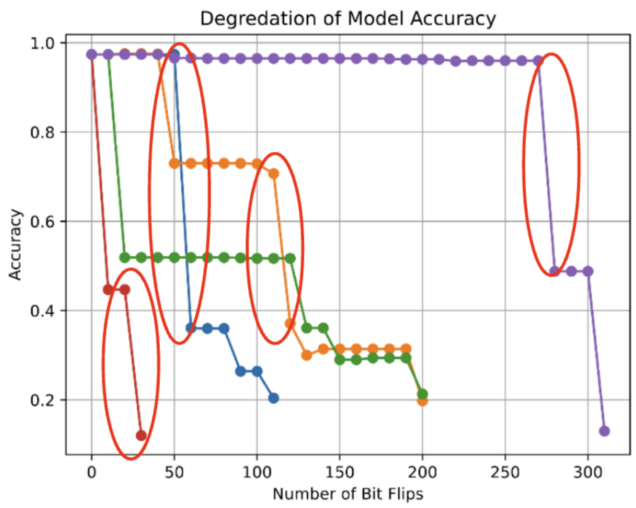


Figure 1: Model degradation

3.2 MAC Tags

MACMul computes one tag per weight matrix. The algorithm works as follows:

```
 Partition weight matrix into chunks $C_1$, $C_2$, ..., $C_n$

Compute $S_i$, the rounded sum of the weights of $C_i$ to 4 decimal places.

Digest all $S_i$ using AES-CMAC block cipher algorithm.

Return final tag from CMAC-AES
```

The summing strategy serves two purposes. First, due to encryption overheads, it is not computationally feasible to compute one ciphertext per weight. Aggregation via summing reduces the number of blocks read into AES-CMAC, reducing the encryption overhead. Secondly, rounding to 4 decimal places allows some freedom for minuscule weight changes. Such weight changes should not impact the performance of the model and are ignored. This logic is implemented in Python in the `mu_hash` method in utils/model.py. As our solution is written in Python, we use PyCryptodome for the implementation of AES-CMAC, which is a Python package of low-level cryptographic primitives.

3.3 Parameter Injection

Once the model has been lowered into Apache TVM's intermediate representation, it is defined as a set of TVM Relax functions, with one main function, the point of entry for the model. These primitive functions can be modified and re-inserted into the intermediate representation. To incorporate the MACs into the neural network, we define new TVM Relax variables for each MAC. Then, we update the signature of the main function by adding the new Relax variables to the list of parameters, while keeping the body of the function the same. This allows us to call the main function of the neural network with the MACs as arguments and gives the neural network access to the MACs.

3.4 Custom Transformations

In Apache TVM, users may define custom transformations that act on the model's intermediate representation. The custom transformations view the model as a computational graph, where nodes are operations (e.g. matmul, ReLU), and edges correspond to the output of one operation being the input to the next. Each TVM custom transformation has an associated Mutator, which is responsible for traversing the graph. Our Mutator, named MACMul, traverses the computational graph in post-order. At each node, MACMul inspects the operation name. If the operation name corresponds to a matrix multiplication and acts on a weight matrix, MACMul inserts the tag verification logic as follows: First, the mutator indexes the correct MAC for this layer. Then, MACMul defines a TVM BlockBuilder, which is a computation block that can call a sequence of primitive tensor functions. The first primitive tensor function MACMul inserts into the block builder is the same matrix multiplication operation. Then, MACMul inserts an external library function `check_tag(w, h)`, which performs the MAC verification logic given the weight matrix and ground-truth tag. Finally, MACMul replaces the original matrix multiplication operation with the new block in the computational graph. At run-time, this block computes the matrix multiplication on the weight and input and also verifies the MAC. This transformation is called within a Module Pass named MACMulPass. All of the transformation-related logic can be found in models/mlp/mlp_interactor.py.

3.5 Balancing Security and Performance

MAC verification on neural networks can be costly, yet fast inference time is crucial for many applications. To manage the performance cost, we provide two tunable parameters that define a balance between performance and security that aligns with their use case. These are discussed next.

3.5.1 Probability Schedule

Prior works suggest that certain layers of a neural network may be more vulnerable than others [3]. Accordingly, MACMul allows the user to define a probability schedule. The probability schedule is a list of probabilities, one for each layer, which dictates the probability that the tag verification logic is executed at that layer for a single inference performed by the model. As long as the probabilities are non-zero, any tampering will eventually be detected, but tampering at vulnerable layers will be identified more quickly if they are assigned a higher probability. This reduces the performance overhead on average, as non-vulnerable layers will not be verified frequently.

3.5.2 Number of Chunks

The tag algorithm described in 3.2 can be computationally expensive, and increasingly so as the number of chunks the weight matrix is partitioned into grows. More chunks mean more runs of the AES cryptography algorithm, which has a computational and memory overhead.

We observed that the inference time increases linearly with the total number of chunks. When using MACMul, the user can define a maximum acceptable inference time. Based on the linear relationship between the number of chunks and runtime, MACMul will find the number of chunks that respects this upper bound while still maintaining security. Then, it will divide the allocation of these chunks between each layer proportional to that layer's parameter count.

3.6 Summary

Given a model and tunable parameters provided by the user, MACMul computes the tags, probability schedule, and number of chunks. It injects all of these as parameters to the intermediate representation of the model. Then, the custom transformation described in 3.4 is run on the model. At inference time, if an integrity violation is detected, an assertion error is thrown by the check_tag operation. We leave it to the machine learning engineer to handle this assertion error appropriately.

4 Experiments and Results

We present 2 experiments on a 3-layer MLP with Linear layers followed by a ReLU non-linearity, which has the following weight matrix shapes: 784x128, 128x128, 128x10. These experiments provide insights into the security and performance of MACMul. All experiments were run on the University of Toronto comps server, which runs Ubuntu 22.04.5 LTS and uses an x86_64 architecture. All computation was performed on the CPU. All experiments were performed on models trained on the MNIST Dataset. The code and results for all of the experiments can be found in the experiments directory of the MACMul GitHub repository.

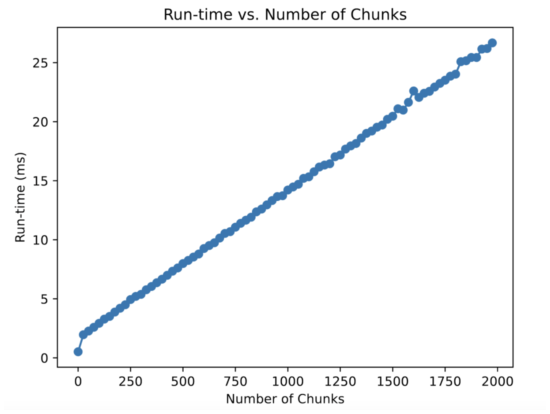


Figure 2: Runtime vs Number of Chunks

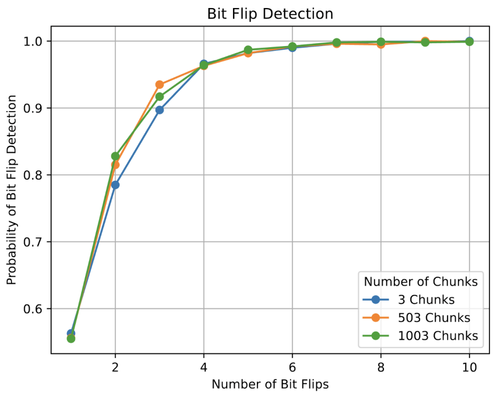
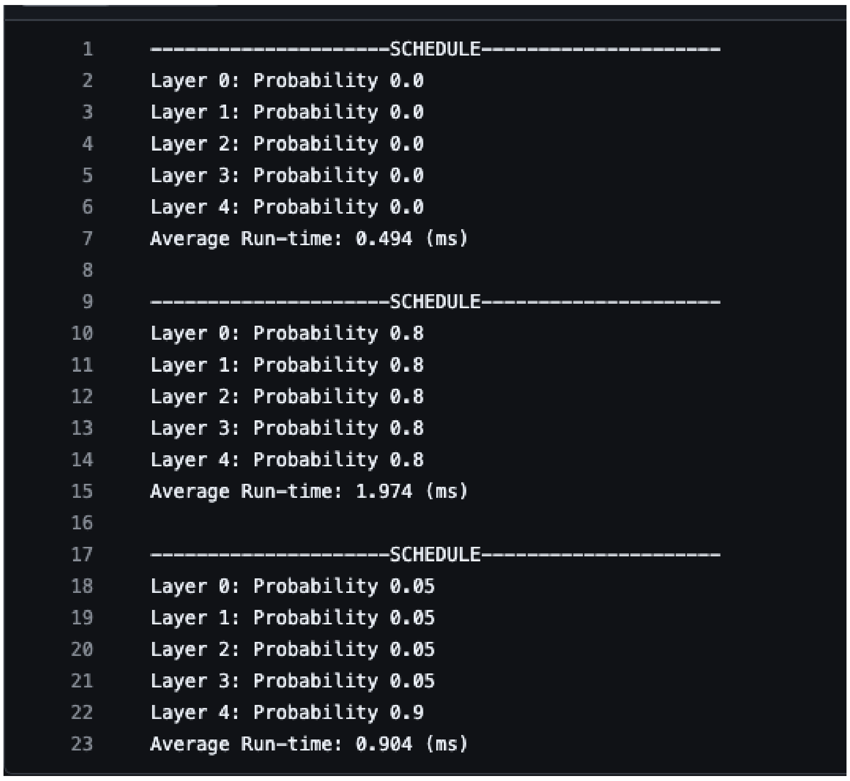


Figure 3: Bit Flip Detection

Figure 1 compares the runtime for 1 inference as the number of chunks varies between 0 and 2000. The runtime increases sharply from 0 to 5 number of chunks and then grows linearly. The sharp increase is due to the check_tag operation loading the weights a second time (in addition to the matrix multiplication), which is a cost we incur regardless of the number of chunks. From there, the runtime grows linearly as more chunks lead to more blocks digested by the AES-CMAC cipher. Figure 2 inspects the probability of detecting N ($0 \leq N \leq 10$) bit flips for models secured by 3, 503, and 1003 chunks (the probability schedule for this experiment is fixed at 1. for all layers). MACMul reliably detects ≥ 6 bit flips. The undetected bit flips are ignored by the rounding mechanism, which means they must not change the magnitude of the weights by much. If one of the first 5 bit flips is critical, it will still be detected by MACMul as this bit flip will have significantly changed the magnitude or the sign of the weight. Furthermore, we see that under the random bit-flip attack model, the number of chunks does not provide a greater security guarantee, however, we believe more chunks would provide better security against a powerful attacker (explained in the Future Works section).



```
--------------------SCHEDULE--------------------
Layer 0: Probability 0.0
Layer 1: Probability 0.0
Layer 2: Probability 0.0
Layer 3: Probability 0.0
Layer 4: Probability 0.0
Average Run-time: 0.494 (ms)

--------------------SCHEDULE--------------------
Layer 0: Probability 0.8
Layer 1: Probability 0.8
Layer 2: Probability 0.8
Layer 3: Probability 0.8
Layer 4: Probability 0.8
Average Run-time: 1.974 (ms)

--------------------SCHEDULE--------------------
Layer 0: Probability 0.05
Layer 1: Probability 0.05
Layer 2: Probability 0.05
Layer 3: Probability 0.05
Layer 4: Probability 0.9
Average Run-time: 0.904 (ms)
```

Figure 4: Probability Schedules

This experiment computes the average run-time of one inference on a 5-layer MLP for 3 different probability schedules. The runtimes are averaged across 5000 inferences. The baseline (tags are not checked) takes 0.494ms. When each layer is checked with probability 0.8, the average runtime increases ∼4.0x to 1.974ms. When the first 4 layers are only checked with probability 0.05, and the final layer is checked with probability 0.9, the average runtime increases ∼1.8x to 0.904ms. Note that these runtimes are an average over many inferences. The probability schedule does not change the worst-case inference time, where all layers are checked, but rather, provides a performance gain for some inferences.

Based on these results, assuming a random bit-flip attack model, we recommend 1 chunk per layer with high verification probabilities assigned to vulnerable layers, and lower probabilities assigned to non-vulnerable layers. If the attacker is powerful and understands the MACMul tag generation algorithm, the number of chunks should be increased.

5 Limitations and Future Works

5.1 Double Loading Weights

Since MACMul adds the check_tag operator to each layer of the computational graph of the neural network, each weight matrix is loaded a second time. This roughly doubles the runtime regardless of the number of chunks. PTGuard embeds the MAC within the PTE itself, allowing the memory controller to check the MAC at the same time the PTE is accessed and avoiding the double-loading problem [1].

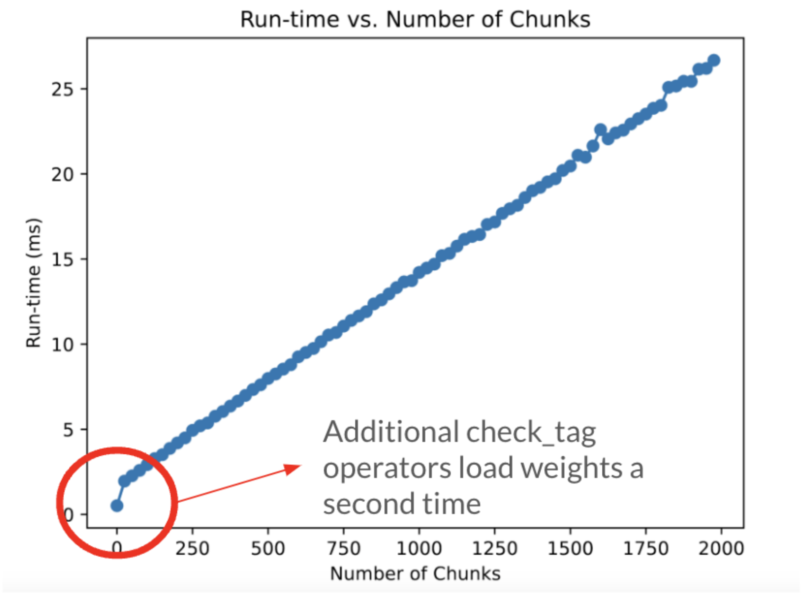


Figure 5: Performance Hit Incurred by Double Loading

Future works should explore a solution that does not incur this extra loading cost. Fortunately, the custom transformation described in 3.4 allows for a MACMul implementation that performs the matrix multiplication and tag verification at the same time. Specifically, since the custom transformation identifies and replaces the relevant matrix multiplication operations with a new Relax Block, a future iteration would only need to change the definition of the new Relax Block. In fact, this is why the custom transformation was written in this way. We envision a solution with a new Relax Block that simply has a modified matrix multiplication operation that computes and checks the tags as the weight matrix is being accessed.

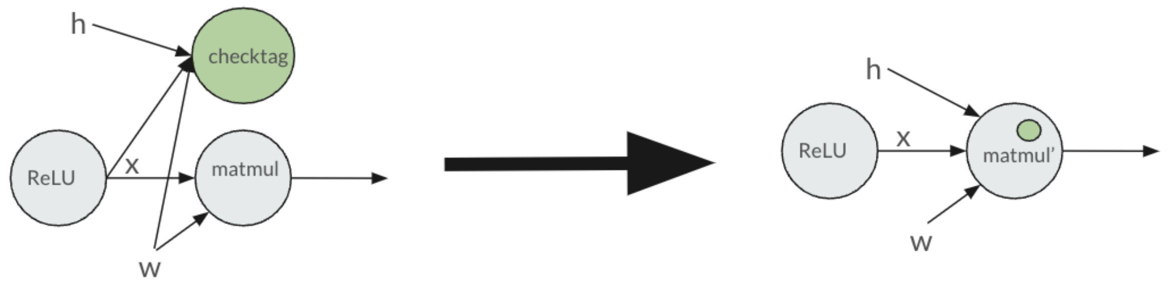


Figure 6: **Left**: Current Computational Graph of the neural network. Note that the check_tag and matmul are separate operations. **Right**: A future iteration of MACMul, where the transformation produces a new matrix multiplication (matmul'), which checks the tags as the weights are being loaded.

5.2 Summing Vulnerability

Consider the weight matrix below. Assume that the attacker flips the sign on the first and last weight in the first row, which corresponds to the first chunk. In this case, the sum of the chunk will remain the same, but the model's behavior will be different, effectively bypassing MACMul. As the chunks become smaller, this attack is more difficult, since the attacker is restricted in the number of sign flips that bypass MACMul. This is why more chunks provide greater security against a powerful adversary.

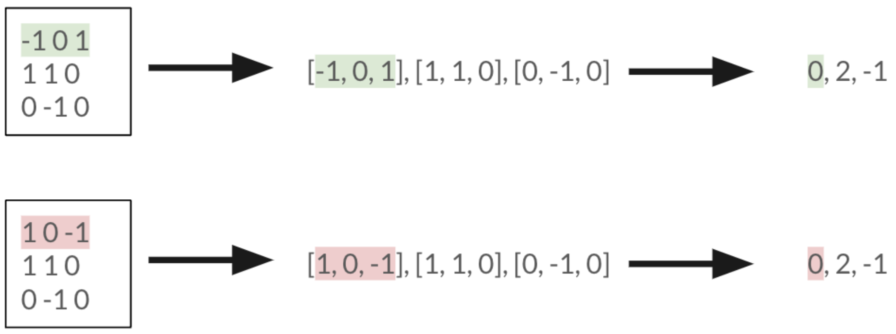


Figure 7: Summing Vulnerability

Obfuscation can mitigate the summing vulnerability further. Here, the weights are chunked non-sequentially based on a secret key that dictates this order. The key is replaced each epoch. This gives the attacker less time to find 2 bits that can exploit the summing vulnerability.

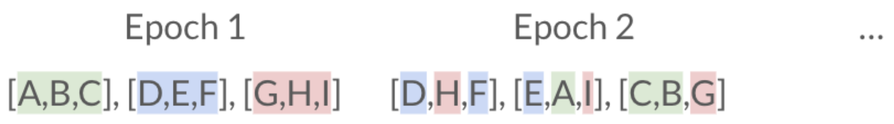


Figure 8: Obfuscation with Per-Epoch Keys

Finally, we argue that this attack model is unrealistic in practice. It requires an attacker who can flip specific bits in specific weights with extremely high precision while not flipping any other bits. It also assumes the attacker knows the model's parameters, which is typically not the case for frontier models that have been fine-tuned for a downstream task.

5.3 Extension to More Operations and Frameworks

MACMul is designed to handle MLPs defined in PyTorch with linear layers and non-linear activation functions (e.g. ReLU). Updating the custom transformation described in 3.4 to handle more operations will make this solution more robust and protect more architectures. Future works should rely more heavily on Apache TVM's ability to transform models defined in any ML framework into the intermediate representation.

5.4 Dynamic Probability Schedule

Vig et. al proposed a dynamic integrity tree to identify and protect vulnerable rows as they are being targeted [11]. We envision a future iteration of MACMul that uses a dynamic probability schedule. That is, the probability of tag verification of each layer is adjusted as weights are being targeted. Layers with weights that are being targeted would be temporarily assigned a higher tag verification probability, whereas other layers would maintain a low probability. This would ensure that inference time is largely unaffected unless a Rowhammer attack is detected. However, to detect bit flips in any layer eventually, the probability for all layers should never reach 0.

6 Conclusion

In this paper, we proposed MACMul, an integrity protection defense that uses Apache TVM and AES-CMAC to detect weight tampering, with tunable parameters to balance performance and security. We analyze security and performance guarantees using a series of experiments, which we use to provide insights for using MACMul. Finally, we explore limitations and discuss the next steps for a second iteration of MACMul. We hope that this system provides a strong framework for future integrity protection systems with minimal performance overhead that maintains a high degree of integrity in neural networks.

References

[1] Saxena, A., Saileshwar, G., Juffinger, J., Kogler, A., Gruss, D., Qureshi, M. (2023). PT-Guard: Integrity-protected page tables to defend against breakthrough rowhammer attacks. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Porto, Portugal, 2023 (pp. 95-108). IEEE. https://doi.org/10.1109/DSN58367.2023.00022

[2] Li, S., Wang, X., Xue, M., Zhu, H., Zhang, Z., Gao, Y., Wu, W., Shen, X. (2024). Yes, *One-Bit-Flip* Matters! Universal *DNN* Model Inference Depletion with Runtime Code Fault Injection. In *33rd USENIX Security Symposium (USENIX Security 24)*, 1315-1330. USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/li-shaofeng

[3] Javaheripi, M., Koushanfar, F. (2021). HASHTAG: Hash signatures for online detection of fault-injection attacks on deep neural networks. In *2021 IEEE/ACM International Conference on Computer Aided Design (ICCAD)* (pp. 1–9). IEEE Press. https://doi.org/10.1109/ICCAD51958.2021.9643556

[4] Machine Learning Compiler. Retrieved November 25, 2024, from https://mlc.ai/

[5] Jena, B. K. (2024, July 16). AES encryption: Secure data with advanced encryption standard. Simplilearn. Retrieved November 25, 2024, from https://www.simplilearn.com/tutorials/cryptography-tutorial/aes-encryption

[6] TechTarget. (n.d.). block cipher. TechTarget. Retrieved November 25, 2024, from https://www.techtarget.com/searchsecurity/definition/block-cipher

[7] Fortinet. (n.d.). What Is a Message Authentication Code?. Fortinet. Retrieved November 25, 2024, from https://www.fortinet.com/resources/cyberglossary/message-authentication-code

[8] Song, J. H., Poovendran, R., Lee, J., Iwata, T. (2006). *The AES-CMAC algorithm* (RFC 4493). Internet Engineering Task Force. Retrieved November 25, 2024, from https://www.ietf.org/rfc/rfc4493.txt

[9] Wang, J., Zhang, Z., Wang, M., Qiu, H., Zhang, T., Li, Q., Li, Z., Wei, T., Zhang, C. (2023). Aegis: mitigating targeted bit-flip attacks against deep neural networks. In *Proceedings of the 32nd USENIX Conference on Security Symposium* (SEC '23), 131, 18 pages. USENIX Association. Anaheim, CA, USA.

[10] Li, J., Zhang, Z., Wang, M., Qiu, H., Zhang, T., Li, Q., Li, Z., Wei, T., Zhang, C. (2020). Defending Bit-Flip Attack through DNN Weight Reconstruction. In *Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC)*, 1-6. San Francisco, CA, USA. IEEE. ht

[11] Vig, S., Bhattacharya, S., Mukhopadhyay, D., Lam, S.-K. (2018). Rapid detection of rowhammer attacks using dynamic skewed hash tree. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '18)* (Article No. 7, pp. 1–8). Association for Computing Machinery. https://doi.org/10.1145/3214292.3214299