Blind Data Adversarial Bit-flip Attack against Deep Neural Networks

Behnam Ghavami, Mani Sadati, Mohammad Shahidzadeh, Zhenman Fang, Lesley Shannon
Simon Fraser University, Burnaby, BC, Canada
Shahid Bahonar University of Kerman, Iran
Emails: {behnam_ghavami, zhenman, lesley_shannon}@sfu.ca

***Abstract*—Because of their high accuracy, deep neural networks (DNNs) have achieved amazing success in security-critical systems such as medical devices. It has recently been demonstrated that Adversarial Bit Flip Attacks (BFAs) against DNN hardware by flipping a very small number of bits can result in catastrophic accuracy loss. The reliance on test data, however, is a significant drawback of previous state-of-the-art bit-flip attack methods. This is frequently not possible with applications containing sensitive or proprietary data. In this paper, we propose Blind Data Adversarial Bit-flip Attack (BDFA), a novel technique to enable BFA against DNN hardware without any access to the training or testing data. This is achieved by optimizing for a synthetic dataset, which is engineered to match the statistics of batch normalization across different layers of the network and the targeted label. Experimental results show that BDFA could decrease the accuracy of ResNet50 significantly from 75.96% to 13.94% with only 4 bits flips.**

I. Introduction

In recent years, Deep Neural Networks (DNNs) have achieved tremendous results on different computer vision and speech recognition tasks such as image classification[1], [2], object detection[3], and segmentation[4]. As DNNs become more popular and applicable in real-world scenarios, their security and safety issues are also becoming crucial. As a result, it is critical to investigate the vulnerability of DNN-based systems against various attacks.

Several serious security concerns have recently revealed in various DNN-related applications. The most popular security concern with DNNs is perturbed inputs, also known as ”adversarial example” [5], [6], [7], [8], [9], [10], [11]. By introducing minor changes to the original inputs, this type of attack can cause a DNN to produce misclassification outputs.

Recently, a new class of attacks has raised further security concerns on DNNs known as adversarial parameter attacks via the development of fault injection attacks on the storage of DNN parameters [12], [13], [14], [15], [16], [17]. This type of attacks try to perturb DNN's parameters in the memory via intentional bit-flipping and cause the DNN to malfunction. These malicious bit-flips have been realized in DNN accelerators via well know RowHammer attack on the DRAM containing the model parameters [15]. RowHammer attack[18] has been shown to maliciously flip the memory bits in DRAM in a software manner without being granted any data write privileges.

In the security analysis of DNN models, adversarial parameter perturbation is an active area of research. However, deploying adversarial perturbations on DNNs in real-world applications is difficult because existing work on bit-flip attacks [12], [13], [14], [15], [16], [17] focuses primarily on ”whitebox settings” in which the attacker has complete access to the target model and test/training data. The white box assumption allows an attacker to gain access to a collection of network architecture, parameter values, and test data. Although side-channel model extraction techniques [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29] can be used to extract information about network architecture and parameters; however, still, the main difficulty in employing prior adversarial bit flip attacks (BFAs) is that the perturbing entity, i.e., the adversary, must have access to network test/validation data. This is because in BFA, as will be introduced in section III, the adversary should determine the vulnerable bits deploying gradient computation and hence, needs to rank the sensitivity of every attackable bit over all the DNN's parameters. On the other hand, training/test data is frequently unavailable in many real-world scenarios due to privacy concerns, such as medical and confidential applications (this is known as the ”blind-data” setting.). Health information is an example of a use case that cannot be uploaded due to various privacy concerns or regulatory restrictions. As a result, traditional BFA methods might not be practically deployed in such DNN-based

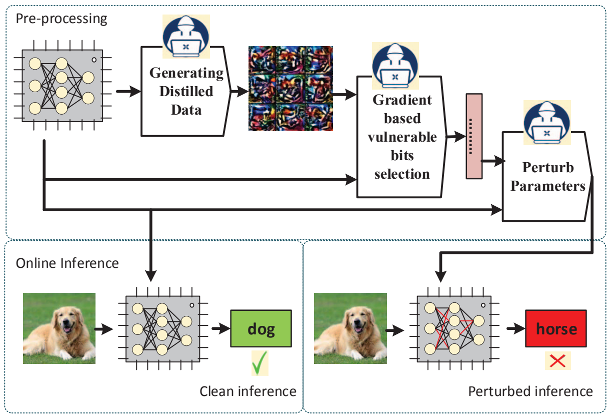

Fig. 1: A general view of proposed blind data bit-flip attack flow. Synthetic data are generated and used to enable conventional gradient-based bit-flip attack.

edge applications.

In this paper, we introduce a new and effective blind data bit-flip attack that causing the DNN to malfunction completely just via bit flipping a few of DNN parameters. In this regard, we generate synthetic data similar to the training dataset using the knowledge of the DNN architecture and deep learning datasets in general (Figure 1). The distilled data is only obtained by inspecting the trained model and would be used via traditional gradient-based BFA technique to find the most vulnerable bits. This work is a step closer to achieving a fully black-box parameter attack. Experimental results show that the introduced blind data Bit-Flip attack (BDFA) can perform similar results to the conventional data-dependent Bit-Flip Attack (BFA) [14]. For example, on ResNet50 architecture that trained on the CIFAR100 dataset, BDFA can crush the DNN model down to the accuracy of 11% by just flipping 8 bits in memory.

In summary the main contributions of this paper are as below:

- We are the first to develop an adversarial parameter bit-flip attack that does not require any data to attack the DNN model.
- We demonstrate that traditional data distillation techniques that only consider the statistics of the batch normalisation layers and do not specify any specific output label would not work efficiently via DNN attack purpose. In this regard, we present a novel data distillation comprised of three distinct sections: batch normalisation loss, DNN output loss, and input normalisation, in order to deploy traditional gradient-based bit-flip attacks via blind-data setting.
- Our bit-flip attack results are comparable to, and even outperform, white-box DNN attacks in some conditions.

The rest of this paper is as follows: Section II gives an overview of prior DNN bit-flip attacks. Section III contains the proposed blind data bit-flip attack. Section IV presents the experimental results. Finally, Section V concludes the paper.

II. Related Work

Fault injection attacks have been used to change the key DNN parameters, like weights and biases, in which stored in memory, reducing overall prediction accuracy to that of random guesses. Liu et al. [12] began by investigating memory error injection on DNN hardware in order to achieve misclassification. Breier et al. [13] experimentally showed what types of memory fault attacks are achievable in practice. They injected faults into the activation function of the DNN to missclassify a predefined input. Rakin et al. [14] demonstrated how to identify memory error patterns that can significantly reduce DNN accuracy. Yao et al. [15] attempted to attack the DNN hardware, which stores network weights in DRAM, using the well-known row hammer attack [18]. Zhao et al. [16] launched a DNN classifier bit flipping attack to secretly misclassify some predefined inputs. Rakin et al. [17] also introduced an aggressive bit flip attack against the DNN model. Its goal was to identify the weights that are strongly associated with target output misclassification. Ghavami et al. [30] recently presented an stealthy attack on DNNs to circumvent the algorithmic defenses: via smart bit flipping in DNN weights, they reserved the classification accuracy for clean inputs but misclassify crafted inputs even with algorithmic countermeasures.

Limitations of previous works: All previous bit-flip attacks required access to the original testing dataset in order to locate the most vulnerable bits of DNN, which may not be applicable in all application scenarios.

III. Proposed Attack

In this section, we present the Blind Data Bit-Flip Attack (BDFA) to maliciously cause a DNN system to malfunction through flipping extremely small amount of the most vulnerable weight bits. To identify the most vulnerable bits, there is a need to compute the gradient of each bit with respect to the DNN's loss function and some input data. Since we assume that the training/testing dataset is not accessible, we generate a batch of synthetic test data using the trained network architecture itself. Our main idea is inspired by prior work in data free DNN quntization [31], [32].

A. Bit-Flip Attack [14]

Conventional BFA [14] aims to degrade the overall accuracy of a DNN. BFA uses the gradient ranking and a progressive bit search to find the most vulnerable bits of a network.

1) Problem Formulation: We denote W as a vector of length n containing all the q-bit quantized and attackable parameters in DNN and B as the vector of binary values, representing all of the bits in W [14]. BFA tries to find a set of parameters B', which has the closest hamming distance D to B and causes the network to malfunction. In other words, it tries to maximize the loss between real parameters B and perturbed parameters B' where the distance between them is smaller than a constant C which is the maximum number of bit flips that can be performed [14]:

$$\begin{aligned} \max_{B'} \; & \mathcal{L}(H(B';X),Y) - \mathcal{L}(H(B;X),Y) \\ s.t. \;\; & D(B',B) < C \end{aligned} \tag{1}$$

where X is the input batch and Y is the target output of that batch, and L is loss function on D through adjusting DNN parameters.

2) Vulnerable bit finding: Using a batch of test data, BFA finds the most vulnerable bits. It first ranks the bits by their gradient, and in the next step tries to find the most vulnerable bits. It ranks all the network's bits B by absolute value of their gradient with respect to the loss L. For this purpose, it first computes the loss function L by providing input data X and output targets Y computed in the section B. Then, it calculates the gradients with respect to the loss through back propagation [14]:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N} f(H(X_i;B),Y_i) \tag{2}$$

$$\overrightarrow{\nabla}\mathcal{L} = \{\frac{\partial \mathcal{L}}{\partial b_0}, ..., \frac{\partial \mathcal{L}}{\partial b_N}\} \tag{3}$$

For finding the most vulnerable bits, BFA uses progress bit search that has two main steps: A) Inner-layer search: In this step, it finds the most vulnerable bits with the use of gradient ranking in every layer and computes the model loss after flipping them. B) Cross-layer search: In this step, it chooses the layer that increased the loss more than other layers and flips the selected bit in that layer. It performs this process several times until it causes the network to malfunction with a small number of bit flips.

B. Generating Attack Oriented Synthetic Data

As shown in the previous section, BFA needs a batch of input data and the corresponding labels in the form of $\{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$ for computing the loss function, which is not accessible in all scenarios. To address this issue, a very naive approach would be to generate random input data from a Gaussian distribution and feed it into the model. This method, however, is incapable of capturing the correct statistics of the training/testing data used for computing the gradients.

In order to generate proper input data to perform attack, we use multiple similarity measures to reconstruct statistically similar data samples to dataset D by starting from randomly generated samples.

1) Input similarity: Starting from random data, in order to make them statistically similar to training data, they should have a close mean and variance to the data samples in D. We set the mean and variance of the initial random data to 0 and 1, respectively. This is because almost all of the deep learning systems use a normalized input to get a more standard and accurate model.

2) Batch normalization layer statistics: Another statistical information comes from the batch normalization layers. Each batch normalization layer contains statistical channel-wise information (mean and variance) of its input neurons during the training process. As a result, by making the statistics of hidden neurons close to the pre-stored statistics of training data, we can have more similarities between the generated data and the training samples. The formulated batch-norm similarity can be shown as [31]:

$$\tilde{\mu}_l(c) = \mathbf{E}_x[sum(\frac{1}{h*w}.featuremap_l^c(x)] \tag{4}$$

$$\tilde{\sigma}_l^2(c) = \mathbf{E}_x[sum(\frac{1}{h*w}.(featuremap_l^c(x) - \tilde{\mu}(c))^2] \tag{5}$$

where $featuremap_l^c(x)$ represents the values of cth input channel in the lth bach normalization layer given input data x. Note that both $\tilde{\mu}_l$ and $\tilde{\sigma}_l^2$ are vectors with length C_l which is the number of input channels in lth batch normalization layer. We calculate $\tilde{\sigma}_l$, the standard deviation of each channel by taking the square root of the elements in the vector $\tilde{\sigma}_l^2$.

The generated batch should have a close μ and σ to the $\tilde{\mu}$ and $\tilde{\sigma}$ that were computed in the training process. In order to estimate the similarity of generated data and the training data, we use the mean squared error as the loss function. By minimizing this loss function, we decrease the Euclidean distance of statistical information in batch-norm layers between the training data and the synthetic data [31]:

$$\min_X \mathcal{L}oss_{BN}(X) = \sum_{i=0}^{L} ||\tilde{\mu}_i - \mu_i||_2^2 + ||\tilde{\sigma}_i - \sigma_i||_2^2 \tag{6}$$

3) Label similarity: To get the gradient of each bit in every parameter, we need to compute the loss function. Based on Equation 2, in order to compute the loss function, each input data should have a ground-truth label. So, we need to assign a label to each generated data sample. Also, every parameter in the network is trained to minimize the loss function over training data. Therefore, to have similar distilled data and training data, we train distilled data in a way that the model's loss function is minimized with respect to the given input and the ground-truth label. However, instead of adjusting DNN parameters, we adjust the input data to minimize the loss function. Since the generated data are random at the beginning and do not have any labels, we can randomly assign labels to each input and train them to minimise the model loss:

$$\min_{X^r} \mathcal{L}oss_{DNN}(X) = \frac{1}{N}\sum_{i=1}^{N} f(Y_i^{'}, Y_i) \tag{7}$$

4) Combining together: In the first step of generating the synthetic inputs, we set the mean and variance of the random training batch to 0 and 1. Then we define the distillation loss as a combination of $\mathcal{L}oss_{DNN}$ and $\mathcal{L}oss_{BN}$. In other words, in the training process we try to minimize the following loss function:

$$\min_{X^r} \mathcal{L}(X) = \alpha \;.\; \mathcal{L}oss_{BN}(X) + \;\beta \;.\; \mathcal{L}oss_{DNN}(X) \tag{8}$$

α and β are hyper parameters for the loss function to balance the effect of each part of the loss function.

The pseudo-code provided for the task is presented in Algorithm 1. In this algorithm, given model M and knowing that the data shape is $N*C*H*W$, we want to generate a batch of data X (line 2). Note that N is the batch size, and each input data has a shape of $C*H*W$. The algorithm begins by generating a random batch of data from the normal distribution with $\mu = 0$ and $\sigma = 1$. Line 3 and 4 store computed mean and standard deviation of each BN layer, which were calculated and saved in those layers, during the training process. In line 5, as mentioned in section B.3, we randomly assign each data in the batch to a ground-truth label, and we will use these labels to train our distilled data according to equation 6. Line 6 to 12 is the main loop for the generation of data. Like every other deep learning training process, we start each iteration with a forward propagation and compute μ and σ and the outputs of DNN, using our data X (line 7). In lines 8 and 9, we compute the two parts of the final loss function according to equations 5 and 6. Then we calculate the total loss by combining these two parts and adding the hyper-parameters to balance the loss function. Finally, in line 11, we do the back-propagation and update data X. By doing this for enough iterations (e.g., 500), we can produce our distilled data and use it to attack the DNN model.

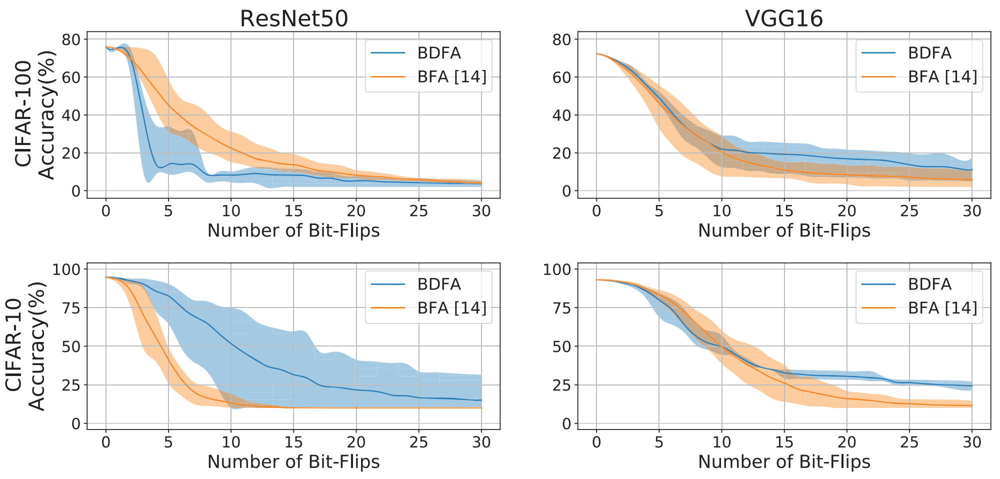

Fig. 2: The Top one accuracy of BDFA and BFA after performing 0 to 30 bit-flips. We repeated the experiment 5 times. The line indicates the average over these 5 tests. The shadow around each line indicates the error band(the minimum and maximum of these 5 tests).

Algorithm 1 Synthetic Data Generation

```
Input: A Deep learning model M with L layers of BN
       shape=(N=batch size,C=3,H=32,W=32)
Output: Generated data X
1:  procedure GENERATE DATA($M$,$Shape$)
2:      $\overrightarrow{X}$ ← RandomNormalizedData(Shape)
3:      $\tilde{\mu}_{\forall j \in 1,2,..,L}$ ← computed mean in the training process
4:      $\tilde{\sigma}_{\forall j \in 1,2,..,L}$ ← computed std in the training process
5:      $y$ ← GenerateRandomLabel(N)
6:      for $iteration = 1, 2, \ldots$ do
7:          $\mu_j$,$\sigma_j$,$y'$ ← ForwardProp($\overrightarrow{X}$,M)    ▷ $\forall j \in 1, 2, .., L$
8:          $loss_{BN}$ ← BNLOSS($\mu$,$\sigma$,$\tilde{\mu}$,$\tilde{\sigma}$)    ▷ equation 5
9:          $loss_{DNN}$ ← DNNLOSS($y$,$y'$)    ▷ equation 6
10:         loss ← $\alpha.loss_{BN} + \beta.loss_{DNN}$
11:         $\overrightarrow{X}$ ← updated $\overrightarrow{X}$ by BackProp(loss)
12:     end for
13:     return ($\overrightarrow{X}$, $y$)    ▷ $\overrightarrow{X}$ is the generated artificial input
        batch and y is the target output
14: end procedure
```

IV. Experimental Results

A. Experimental Setup

1) Datasets: We used CIFAR-10, and CIFAR-100 [33], popular datasets for image classification. We used these datasets to train our models. Both CIFAR-10 and CIFAR-100 contain 60000 RGB images with a size of $32 * 32$.

2) DNN Architectures: We chose VGG16 and ResNet50, which are two of the conventional CNN architectures. Both of these architectures use batch normalization layers to achieve better performance. VGG16 and ResNet50 have, respectively, 13 and 53 batch normalization layers. We implement these architectures in the Pytorch framework [34] and use 8-bit quantization [35] for network parameters.

3) Attack Assumptions: In our experiments, we assume that we have full access to the network's parameters and architecture. However, contrary to previous papers, we do not assume having a batch of input data; Instead, we use generated distilled data as the inputs for DNN.

B. Attack results and comparing to other methods

In this section, we demonstrate our results on different networks and datasets and compare our results to previous work done by Rakin et al. [14]. Table I shows the baseline accuracy of both networks on CIFAR-10 and CIFAR-100. In the experiments, we use a batch size of 128, the best batch size for Bit-Flip Attack [14], to have a fair comparison. However, we are able to generate data as much as we want.

BDFA was able to successfully attack all of the models and datasets that were given. Interestingly, BDFA forces a single output label to be applied to almost all the input images (as shown in Fig 3). It means that after a few initial bit-flips, BDFA tries to maximize the output of one class in order to always make it the selected output.

Network	CIFAR-10 Acc(%)	CIFAR-100 Acc(%)
Resnet50	94.63	75.96
VGG16	93.05	72.34

TABLE I: The baseline accuracy of 8 bit quantized Resnet50 and VGG16 on cifar10 and cifar100 datasets.

CIFAR100: The top row of Figure 2 shows the obtained results from attacking ResNet50 and VGG16 on the CIFAR-100 dataset. In ResNet50, the model accuracy decreases significantly from 75.96% to 13.94% with only 4 bits flips, and in 4 of the 5 tests performed, it reaches less than 9%. Also, based

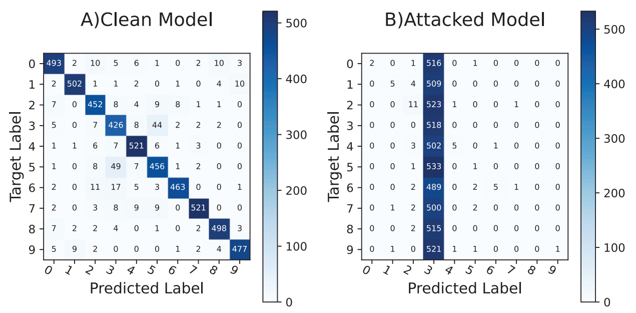

Fig. 3: Comparison between the confusion matrix of a clean ResNet50 model trained on CIFAR-10 with the same model after flipping 10 bits of its parameters.

on Table II, by continuing bit-flips up to 30 bits, it reaches an average of 3.6%. For the VGG16 network, the model's accuracy after 9 bit-flips drops from 72.34% to less than 30% in all 5 tests, and the average accuracy in different runs after 30 bits-flips reaches 11.05%, showing the attacks were quick and successful.

Network	**CIFAR-100**		**CIFAR-10**	
	BDFA(%)	**BFA(%)**	**BDFA(%)**	**BFA(%)**
Resnet50	3.6 ± 1.6	3.94 ± 1.5	15.08 ± 16.3	10.1 ± 0.1
VGG16	11.05 ± 6.3	5.8 ± 4.9	24.3 ± 2.9	11.5 ± 2.9

TABLE II: Top one accuracy comparison of BDFA and BFA after performing 30 bit-flips. These results are the average over 5 tests plus maximum error obtained by the average.

CIFAR10: The bottom row of Figure 2 shows the accuracy drop comparison between BDFA and BFA [14] on VGG16 and Resnet50 in which trained on CIFAR-10. As shown in Table II, the accuracy of VGG16 and ResNet50 decreases to nearly 24.3% and 15.08% with only 30 bit-flips. This shows that BDFA is able to decrease the model accuracy significantly by only flipping 30 bits out of more than 500 million ResNet50 parameters (4 billion bits).

Comparision to BFA: As shown in Figure 2, both BDFA and BFA work well for finding the first few vulnerable bits and causing DNN to malfunction. In Resnet50 trained on CIFAR100, we achieve better performance than BFA, and with only 4 bit-flips, BDFA can decrease the model accuracy to 5-20%. Therefore, it shows that the artificial data has better statistical similarities than one batch of training data. Although using distilled data can drastically decrease the accuracy of the model to 20-10% with just 10-20 bit-flips, it can not completely destroy the function of DNN and decrease it to 0%. It is also worth noting that, unlike the BFA, the BDFA has no access to any kind of test data.

Effectiveness Of Distilled Data: Here, we discuss the importance of generating distilled data as described in section III. Fig 4 depicts the attack effectiveness for different loss functions in case of VGG16 in CIFAR-10. The random Gaussian data (blue) and the Batch-Norm loss (orange), do not have ground-truth labels. Therefore, we assign their labels to a random class. In the case of Gaussian data, the inputs

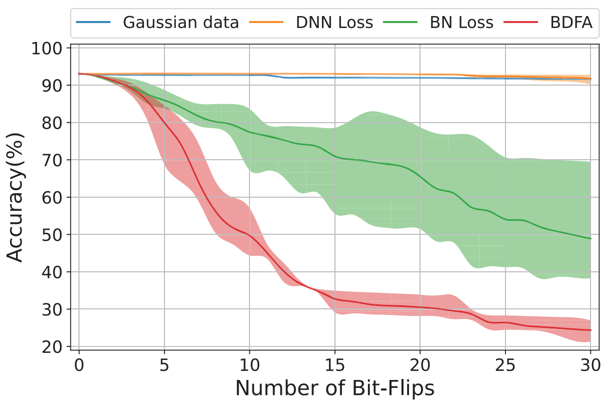

Fig. 4: Comparison between different part of loss function for generating distilled data. The blue plot indicates that random gaussian inputs were used for the attacking purpose, whereas the orange (DNN loss), green (BNN loss), and red (BDFA) plots indicate the loss function used for generating the input data based on equations 7, 6, and 8, respectively.

are just generated from a uniform random distribution. Since it does not use any statistical similarities for neurons, it does not affect DNN, and we can not find the vulnerable bits with random data. The furthest we can get by using this method is to reach 1.2% accuracy drop. The orange line is showing the results for using data generated by only the second part of the loss function (equation 5). It is still unable to crush DNN and only can cause 1.2% accuracy drop after 30 bit-flips, same as random data. The green line shows the result of generating data based on the first part of the loss function (equation 6). It implies that batch-norm layer similarities have a notable impact on the generated data. However, this method is not solely effective and only decrease the accuracy from 93.05% to 70-39% (average accuracy drop is 43%). The red line shows our approach to generating distilled data by combining the two mentioned loss functions (equation 7). As it can be seen, thr proposed loss function is able to cause a drastic drop in accuracy, around 70% with 30 bit-flips and causing DNN to fully malfunction. It implies that both parts of the loss functions in equation 7 are necessary and useful to have a successful attack.

V. Conclusion

This paper presents a blind data bit-flip attack (DBFA) on deep neural networks, which exploits synthetic data for attack usage. We show that BDFA can decrease model accuracy dramatically to the random point using the distilled data which is only obtained by inspecting the pee-trained model. Experimental results show that the BDFA can perform similar results to traditional Bit-Flip Attack (BFA) which based on the training/testing data.

As exploiting the DNN architecture and parameters using side-channel attacks has shown to be possible, BDFA could be the first attempt to perform adversarial parameter attacks on DNNs in a black-box setting by eliminating the need for any input data. This poses an important threat to any safety-critical application that uses deep learning models.

VI. Acknowledgements

We acknowledge the support from Government of Canada Technology Demonstration Program and MDA Systems Ltd; NSERC Discovery Grant RGPIN341516, RGPIN-2019-04613, DGECR-2019-00120, Alliance Grant ALLRP-552042-2020, COHESA (NETGP485577-15), CWSE PDF (470957); CFI John R. Evans Leaders Fund; Simon Fraser University New Faculty Start-up Grant.

References

[1] Y. Sun, B. Xue, M. Zhang, and G. G. Yen, "Evolving deep convolutional neural networks for image classification," *IEEE Transactions on Evolutionary Computation*, vol. 24, no. 2, pp. 394–407, 2019.

[2] A. Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.

[3] S. S. A. Zaidi, M. S. Ansari, A. Aslam, N. Kanwal, M. Asghar, and B. Lee, "A survey of modern deep learning based object detection models," *Digital Signal Processing*, p. 103514, 2022.

[4] S. Minaee, Y. Y. Boykov, F. Porikli, A. J. Plaza, N. Kehtarnavaz, and D. Terzopoulos, "Image segmentation using deep learning: A survey," *IEEE transactions on pattern analysis and machine intelligence*, 2021.

[5] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," *arXiv preprint arXiv:1312.6199*, 2013.

[6] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[7] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "Deepfool: a simple and accurate method to fool deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2574–2582.

[8] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *2017 ieee symposium on security and privacy (sp)*. IEEE, 2017, pp. 39–57.

[9] S.-M. Moosavi-Dezfooli, A. Fawzi, O. Fawzi, and P. Frossard, "Universal adversarial perturbations," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1765–1773.

[10] J. Hayes and G. Danezis, "Learning universal adversarial perturbations with generative models," in *2018 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2018, pp. 43–49.

[11] A. Chaubey, N. Agrawal, K. Barnwal, K. K. Guliani, and P. Mehta, "Universal adversarial perturbations: A survey," *arXiv preprint arXiv:2005.08087*, 2020.

[12] Y. Liu, L. Wei, B. Luo, and Q. Xu, "Fault injection attack on deep neural network," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 131–138.

[13] J. Breier, X. Hou, D. Jap, L. Ma, S. Bhasin, and Y. Liu, "Practical fault attack on deep neural networks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2204–2206.

[14] A. S. Rakin, Z. He, and D. Fan, "Bit-flip attack: Crushing neural network with progressive bit search," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, pp. 1211–1220.

[15] F. Yao, A. S. Rakin, and D. Fan, "Deephammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips," *arXiv preprint arXiv:2003.13746*, 2020.

[16] P. Zhao, S. Wang, C. Gongye, Y. Wang, Y. Fei, and X. Lin, "Fault sneaking attack: A stealthy framework for misleading deep neural networks," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[17] A. S. Rakin, Z. He, J. Li, F. Yao, C. Chakrabarti, and D. Fan, "T-bfa: Targeted bit-flip adversarial weight attack," *arXiv preprint arXiv:2007.12336*, 2020.

[18] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

[19] J. Breier, D. Jap, X. Hou, S. Bhasin, and Y. Liu, "Sniff: reverse engineering of neural networks with fault attacks," *IEEE Transactions on Reliability*, 2021.

[20] H. Chabanne, J.-L. Danger, L. Guiga, and U. Kühne, "Side channel attacks for architecture extraction of neural networks," *CAAI Transactions on Intelligence Technology*, vol. 6, no. 1, pp. 3–16, 2021.

[21] L. Batina, S. Bhasin, D. Jap, and S. Picek, "Sca strikes back: Reverse engineering neural network architectures using side channels," *IEEE Design & Test*, 2021.

[22] W. Hua, Z. Zhang, and G. E. Suh, "Reverse engineering convolutional neural networks through side-channel information leaks," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.

[23] S. J. Oh, B. Schiele, and M. Fritz, "Towards reverse-engineering black-box neural networks," in *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*. Springer, 2019, pp. 121–144.

[24] C. Gongye, Y. Fei, and T. Wahl, "Reverse-engineering deep neural networks using floating-point timing side-channels," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[25] L. Batina, S. Bhasin, D. Jap, and S. Picek, "{CSI}{NN}: Reverse engineering of neural network architectures through electromagnetic side channel," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 515–532.

[26] J. Wei, Y. Zhang, Z. Zhou, Z. Li, and M. A. Al Faruque, "Leaky dnn: Stealing deep-learning model secret with gpu context-switching side-channel," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020, pp. 125–137.

[27] N. K. Jha, S. Mittal, B. Kumar, and G. Mattela, "Deeppeep: Exploiting design ramifications to decipher the architecture of compact dnns," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 17, no. 1, pp. 1–25, 2020.

[28] Y. Zhu, Y. Cheng, H. Zhou, and Y. Lu, "Hermes attack: Steal {DNN} models with lossless inference accuracy," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[29] A. S. Rakin, M. H. I. Chowdhuryy, F. Yao, and D. Fan, "Deepsteal: Advanced model extractions leveraging efficient weight stealing in memories," *arXiv preprint arXiv:2111.04625*, 2021.

[30] B. Ghavami, S. Movi, Z. Fang, and L. Shannon, "Stealthy attack on algorithmic-protected dnns via smart bit flipping," *arXiv preprint arXiv:2112.13162*, 2021.

[31] Y. Cai, Z. Yao, Z. Dong, A. Gholami, M. W. Mahoney, and K. Keutzer, "Zeroq: A novel zero shot quantization framework," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 13 169–13 178.

[32] Y. Choi, J. Choi, M. El-Khamy, and J. Lee, "Data-free network quantization with adversarial knowledge distillation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, 2020, pp. 710–711.

[33] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research)," 2009. [Online]. Available: http://www.cs.toronto.edu/ kriz/cifar.html

[34] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," 2019.

[35] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," 2018.