Premium Access to Convolutional Neural Networks

Julien Bringer[1], Hervé Chabanne[2,3], and Linda Guiga[2,3](✉)

[1] Kallistech, Paris, France
[2] Idemia, Paris, France
{herve.chabanne,linda.guiga}@idemia.com
[3] Télécom Paris, Paris, France

Abstract. Neural Networks (NNs) are today used for all our daily tasks; for instance, in mobile phones. We here want to show how to restrict their access to privileged users. Our solution relies on a degraded implementation which can be corrected thanks to a PIN. We explain how to select a few parameters in an NN so as to maximize the gap in the accuracy between the premium and the degraded modes. We report experiments on an implementation of our proposal on a deep NN to prove its practicability.

Keywords: Neural Networks · Software protection · Reverse engineering

1 Introduction

Today, many applications rely on Neural Networks (NNs) to perform different classification tasks. Here, we want to investigate how to restrict their use to a set of privileged users, who have access to a premium mode. The premium mode is carried out by providing each user with an NN specially trained for their personal use.

The first idea behind our solution comes from [19]. [19] describes how, in 2016, mobile RSA's SecurID and Vasco DIGIPASS Software Tokens can be hacked despite relying on different defense mechanisms implemented to thwart reverse engineering processes. Its conclusion is that in such a hostile environment, quoting: "The best defense against the attacks shown in this paper is securing the mobile token with a PIN".

As our protection relies on low-entropy PINs, we cannot let hackers perform brute-force attacks at ease. Rather than implementing simple work/no-work modes for the application, we implement a default degraded classification task for incorrect PINs vs an optimal one for the privileged users. Switching from a degraded mode to the premium optimal one is achieved through the modification of some of the NN's parameters. We consider that the attacker has access to the NN's implementation in a degraded mode. We are well aware that an attacker might perform an exhaustive search on all PINs. However, our goal is to slow each attempt down.

A particular emphasis is given to the way we store these critical parameters in the mobile phone. Although our method could also be applied to other contexts, we believe that our method could be applied to mobile phones, as we take storage space into account.

Our second idea relies on some specific layers: the convolutional ones. These are found in Convolutional Neural Networks (CNNs), which are, for instance, the most used NNs for image processing. We select some parameters in a given convolutional layer as the ones enabling us to move from degraded modes to the optimal one.

Going a step further, we exploit the fact that we are dealing with NNs. During a training phase with a dedicated database, the parameters of an NN are optimized. One strategy we introduce uses the fact that an attacker who does not have full access to testing facilities will be unable to find optimal parameters through retraining.

To sum up, each privileged user is provided with an NN trained for him and a PIN enabling him to reach the premium mode of his NN. We end this introduction with a description of some Related Works. In Sect. 2.1, we provide some background about NNs. We describe our proposal to store optimal parameters in Sect. 3. To illustrate its practicability, we detail two examples of parameter selection to show how an unprivileged user would end up with degraded outputs and we report in Sect. 4 our experiments on an NN based on ResNet18 [9] which is typical of deep learning. Section 5 concludes.

1.1 Related Works

Android offers a multi-layer security strategy. [18] describes this security model (see also [1]). Moreover, one may add ad-hoc protections for obfuscating the code making it harder to reverse-engineer [4]. There is an on-going cat-and-mouse game between hackers and developers. However, it seems to us that developers may have a hard time whenever the full access to the code is available to hackers. For instance, for white-box cryptography, where the code for encrypting with DES or AES symmetric algorithms are given to attackers, all academic proposals have been broken so far [8]. Moreover, while complementary to our proposal, anti-reverse engineering techniques tend to inflate the size of the code a lot. For instance, for DES or AES, there may be a multiplication by 16k of the code size, from less than 1KB for an unprotected implementation to more than 16MB. As NNs – such as the one we are considering – are initially large, such an expansion cannot be handled in an embedded environment.

Some papers consider the use of hardware enclaves such as Intel SGX to protect NNs in the cloud setting [11,12,24,25]. However, reverse-engineering and model inversion attacks still remain possible on some protected systems [24,28,29]. Moreover, even though they provide a – relatively – protected environment for their users, they do not consider giving access to predictions to all users, with a degraded mode for unprivileged users.

There are also various works related to deep learning and NNs on mobile phones: for a secure hardware-based implementation of NNs on mobile devices, see [2]. For a comprehensive study on the deployment of deep learning Android apps, see [30].

Our approach is different. We want to force the hacker to measure the performances of the deployed NN without having the possibility to rely on a dedicated database. Here, we study an example of facial recognition (see [27] for a survey of this domain). We believe it is relevant, as a typical deep learning task.

Previous works have studied the notion of privileged information [16,26]. However, they focus on the training phase of NNs, and study how a Student network could achieve a better accuracy when provided with privileged information while training along a Teacher network. Thus, they differ from our work in the sense that this paper focuses on the end-user and the inference phase.

2 Background

2.1 Convolutional Neural Networks

Today, Convolutional Neural Networks (CNNs) are used for making predictions in various fields of application ranging from image processing [3], to classification [21,22] and segmentation [20].

They are composed of several layers:

- Convolutional layers compute a convolution between one – or several – filter F and the input, as follows:

$$O_{i,j} = \sum_{k=1}^{h} \sum_{l=1}^{w} X_{i+k,j+l} \cdot F_{k,l}$$

  where O is the output of the convolution. A convolution can be seen in Fig. 1.

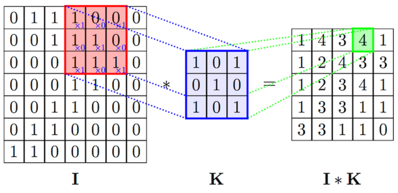

Fig. 1. Convolution between an input I and a filter K

  The elements of the filter are the weights of the layer and will be designated either by 'weights' or by 'parameters' in the rest of the paper.
- Other layer types include fully connected layers – through weights – to all the elements from the previous layer.

- A nonlinear function is applied at the end of each layer. The most popular one is the ReLU (Rectified Linear Unit) function defined as the max of a value and zero. It is used to activate – or deactivate – elements of the layer.
- Finally, pooling layers are usually present between other layers in order to reduce the dimensionality of the input.

The input of each layer consists of different channels. For instance, in image processing, the input of the model is usually divided in three channels corresponding to the RGB colors.

The weights – and other parameters – of a CNN are trained over several epochs – i.e. runs on a training data set – so as to reach a value guaranteeing the best possible prediction accuracy. Given their large number of parameters, and the necessity of high accuracy nowadays, some NNs take days – or even months – to train.

Several techniques are added over the years to make training more efficient. One of these consists in adding a Batch Normalization layer to improve the training phase. In 2015, the authors of [14] discovered this type of layer whose purpose is to make training faster, more efficient and more stable. The layer normalizes its input. Thus, given an element $x_{i,j}$ in a batch B of its input, the layer computes:

$$\tilde{x}_{i,j} = \gamma \frac{x_{i,j} - \mu_B}{\mathbb{V}_B} + \beta \tag{1}$$

where γ and β are parameters optimized during the training phase, and $\mathbb{V}_B$ and μ_B are the considered batch's variance and expected value respectively.

2.2 ResNet18

At first glance, one could imagine that the more layers an NN contains, the better its accuracy will be, once fully trained. However, a known problem occurs for deep neural networks during training. In 2016, the authors of [9] discovered ResNet as a way to better train deeper NNs, without having to deal with it. This is achieved thanks to residual blocks corresponding to "identity shortcuts", described in Fig. 2. More generally, the architecture of Residual Neural Networks introduces these skipping connections. The authors argue that thanks to the identity mapping, the training should be similar be it with or without the shortcut layer. This is why large ResNet architectures, containing sometimes up to 1,001 layers [10], are efficiently trained with a high accuracy.

The model we consider here is based on a particular instance of ResNet: ResNet18. The latter is composed of 17 convolutional layers, a fully connected layer, a max pooling layer and a final global average pooling layer.

2.3 Selecting Optimal Parameters

Our selection strategy aims at choosing only a few parameters – called optimal – which have an impact on the accuracy of the NN.

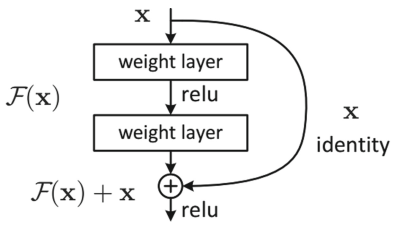
x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Fig. 2. Residual block in ResNet

Section 4 is devoted to detailing how this works on a particular instance of an NN. We will report how the fulfillment of the first criteria is achieved in this case. We also address the second criteria to penalize an attacker who has a limited access to the training database (see Sect. 4.2).

Minimizing the Number of Parameters. The authors of [23] show that some neurons have a higher impact on the model's prediction than others. Indeed, [23] defines a neuron i's sensitivity given an input x as follows:

$$\Delta(i, x) = argmin_{\delta}\{|\delta| | f(x) \neq \tilde{f}^i_{\delta}(x)\}$$

where f is the original model and $\tilde{f}$ is a modified model where noise δ was added to the output of neuron i.

It corresponds to the minimal noise one needs to add to neuron i for the classification to change. The authors of [23] observe that a large number of neurons have a high sensitivity (small Δ).

This result shows that it is possible to select few parameters to protect, and still prevent the attacker from getting a good accuracy.

When the Attacker Does Not Have Access to the Database. [5] operates a distinction between static and dynamic parameters. We will also make such a distinction, but our definition of static parameters is slightly different from theirs. Let us define the following:

1. We say that a parameter w remains unchanged from one training epoch to the next if
$|w_{current\_layer} - w_{previous\_layer}| < r \cdot w_{previous\_layer}$ where $r = 10^{-2}$
2. We denote *static parameters* the parameters that have not changed over the last epoch.
3. *Dynamic parameters* are the non-static parameters.

The choice of r in Point 1. comes from the fact that a slight change in a parameter does not lead to a noticeable drop in the accuracy. What interests us when studying the parameter fluctuation is the way the modifications influence the accuracy. Thus, r is tuned so that the resulting evolution curves for the number of static parameters is representative of the evolution of the accuracy. I.e. r is chosen so that when the accuracy changes less, the number of static parameters increases drastically. After trying several values for r, we selected $r = 10^{-2}$, as it enabled us to differentiate between static and dynamic parameters as per the previous explanation.

Static parameters are easier to obtain by the attacker through a shorter training. Moreover, dynamic parameters are the ones that change the accuracy over the last few epochs and bring it to its optimal value. For those two reasons, protecting the dynamic parameters seems to be a viable strategy in order to limit the number of parameters to protect.

2.4 Per User Training

For a same NN architecture, training with different initialization parameters results in different weights for all layers (see [7]). As our privileged users benefit from dedicated training, they do share the same NN architecture, but with different parameters. For our proposal, this means that we have to modify the optimal parameters for each of the privileged users' NN. To the best of our knowledge, given a trained NN, there is no way to deduce the parameters computed through another training with a different initialization of the same NN.

2.5 Finite Fields

Let $\mathbb{F}$ denote a finite field with 2^l elements such that $2^l - 1$ is a prime. For instance, $\mathbb{F} = \mathbb{F}_{2^{521}}$.

Lemma 1. *1. The non-zero elements of $\mathbb{F}$ form a multiplicative group.*
2. This group is cyclic.
3. In this group, all elements are generators except the unity.

Point 2. of the previous lemma means that all non-zero elements can be expressed as powers of a single element called a generator.

For a proof, see [17].

Let us further note that since l is taken to be a prime, all elements of $\mathbb{F}$ are invertible modulo $p = 2^l - 1$.

3 Protecting Optimal Parameters

Here, we suppose we have a set of n optimal parameters $\{o_1, \ldots, o_n\}$ that we want to keep secret. These secrets have to be protected by a PIN, in such a way that for all PIN values, the protection returns legitimate values. An attacker knows the way the parameters are stored and can try all PIN values.

Let $\mathbb{F}$ denote a finite field with 2^l elements such that $2^l - 1$ is a prime. For instance, $\mathbb{F} = \mathbb{F}_{2^{521}}$.

We want to keep l small. This means that we want a small n too. An example of doing that is given in the next section.

Denote $O = *|o_1|\dots|o_n \in \mathbb{F}$ where $|$ stands for the concatenation and $*$ is a bitstring with no particular value which is introduced to fit the length of the finite field $\mathbb{F}$ elements.

Given a PIN value, we then compute

$$g = O^{1/\mathrm{PIN}} \in \mathbb{F} \tag{2}$$

Let us note that g always satisfies (2) according to Lemma 1.

We store the function $f : \pi \mapsto g^{\pi} \in \mathbb{F}$. We have: $f(\pi) = O$ if and only if $\pi = \mathrm{PIN}$.

Thanks to Point 3. of Lemma 1, g is a generator of the multiplicative group of $\mathbb{F}$ and $f(\text{all the values between 1 and } 2^l - 1) = \mathbb{F} \setminus \{0\}$, which implies that an attacker who tries all possible values of PIN will get all elements of $\mathbb{F}$. This way, she cannot identify which one has been chosen for O.

Note that the implementation of function f does not have to be secured.

4 Example of Application: Facial Recognition

In this section, we consider an adapted version of the ResNet18 [9] model architecture to the task of facial recognition. We think that this is a relevant example, as it demonstrates the feasibility of our concept on an NN structure that is used in different applications, including in a mobile environment. Moreover, relying on facial recognition facilitates experiments on large data sets and comparisons with large scale benchmarks. Our architecture extends ResNet18 and relies on 14 million parameters across 76 layers. Our goal is to extract at most around a hundred parameters.

For facial recognition, the performances are assessed thanks to the accuracy of the recognition. On the one hand, false positives might happen, allowing unauthorized individuals to be recognized. On the other hand, false negatives might be a nuisance to genuine users. More precisely, the error is measured as follows: given a maximal False Acceptance Rate (FAR) – i.e. the probability of a malicious individual being authenticated, assess the False Rejection Rate (FRR) – i.e. the probability of a genuine user being rejected.

The accuracy of our ResNet18-based model on the Labeled Faces in the Wild (LFW) database [13] in our proprietary setting is as follows (3):

- For $FAR = 10^{-4}$, $FRR = 0.24$ %
- For $FAR = 10^{-5}$, $FRR = 0.70$ %

In our case, we reach the best accuracy after 13 training epochs.

4.1 Optimal Parameters for ResNet18

Given the large number of parameters in our NN model, carefully selecting the parameters to protect allows us to limit the size of $\mathbb{F}$ (see Sect. 3). In the following section, we describe two main parameter selection strategies. In the first, we protect parameters from batch normalization layers, either by protecting all the parameters of one layer, or by using the method described in Sect. 2.3. In the second, we describe a strategy to sample elements from a convolutional layer, as a way to limit the number of optimal parameters.

4.2 Batch Normalization

Generating Suboptimal Parameters. When selecting a set of parameters to protect in an NN, the first, most intuitive, strategy would be to protect a layer with few parameters. As explained in Sect. 2, batch normalization layers aim at normalizing the input. For this reason, each of the layer's parameters affect one whole input channel. The said parameters are therefore scarce and impactful. Thus, batch normalization layers are one obvious choice of layer to protect. More specifically, we will focus here on the γ parameters mentioned in (1). We randomize the parameters of a batch normalization layer in the middle of the architecture (38th layer out of 76). The layer contains 128 γ parameters.

Even though the attacker does not have access to trained weights, observing the other batch normalization layers might enable them to spot erroneous settings if the random parameters selected do not reflect the usual distribution of γ parameters. To prevent this, we compute the distribution of the chosen layer's γ parameters and generate values following the same distribution. Figure 3 shows that the γ parameters we generated have, indeed, a distribution similar to that of the original batch normalization γ parameters.

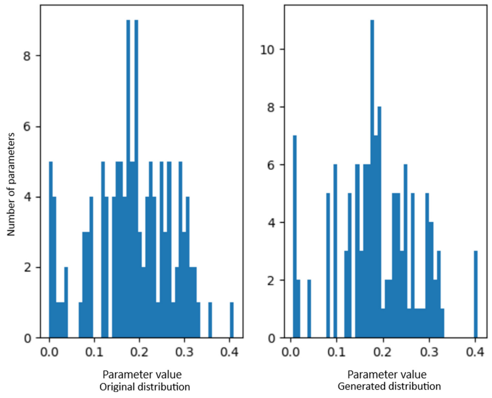

Fig. 3. Distribution of the γ parameters in a batch normalization layer from the ResNet18 network, with the original distribution on the left and the generated distribution on the right

Once we have established the way suboptimal parameters have been generated, we can observe the associated drop in the accuracy and evaluate the security of our process for this strategy.

When we change the selected layer's γ parameters to random ones following the distribution of batch normalization layers, we get the following accuracy:

– For $FAR = 10^{-4}$, $FRR = 0.32$ %
– For $FAR = 10^{-5}$, $FRR = 0.96$ %

Thus, the false rejection rate for $FAR = 10^{-4}$ has increased by 33% and the rate for $FAR = 10^{-5}$ has increased by 37.1% compared to the original model (see (3) for reference). This corresponds to the accuracy of the model after only 8 training epochs. Thus, modifying only one small layer over the 76 ones already results in a critical drop in the accuracy. Protecting the 128 γ parameters of the batch normalization layer would therefore be enough to distinguish between premium and degraded accesses.

The following section describes a second strategy.

Static VS Dynamic. Depending on the layers, the proportion of static parameters – as defined in Sect. 2.3 – varies a lot. While convolutional layers contain mainly dynamic parameters as shown in Fig. 4, the γ parameters in batch normalization layers tend to be mostly static, as can be seen in Fig. 5. Figure 6 shows the distribution of the number of epochs for which the γ parameters have been static. We can see that most γ parameters do not change over the last epoch at least. This explains our definition of static parameters: we seek to select a minimal number of parameters.

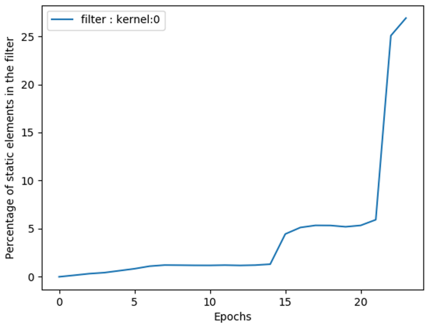

Fig. 4. Percentage of static parameters in a convolutional filter with relation to the epoch.

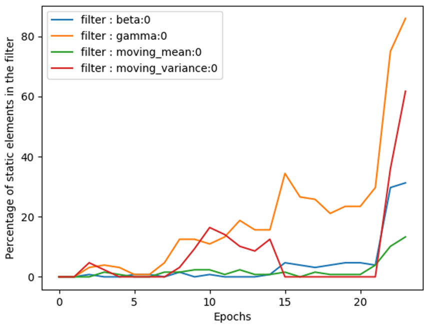

Fig. 5. Percentage of static parameters for each parameter type in a batch normalization layer with relation to the epoch.

Considering this, the advantages of batch normalization layers are threefold:

– They contain few parameters.
– As stated before, γ parameters in batch normalization layers influence several input elements and can have a noticeable effect on the following layers.

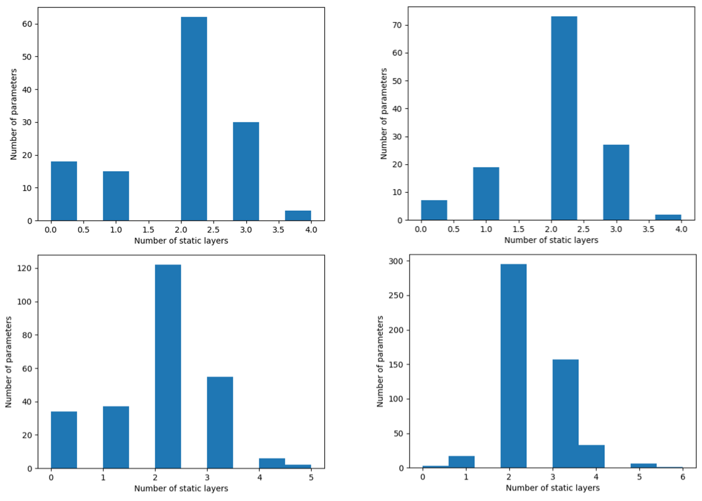

Fig. 6. Number of epochs for which the γ parameters have been static, in four different batch normalization layers

– The large proportion of static parameters means we can protect a few parameters from various batch normalization layers.

Our new strategy is therefore to protect the dynamic parameters from several batch normalization layers.

Since only a few parameters per layer are modified, it is no longer necessary to copy the layer's distribution: making sure the random elements generated are in the range $[0, 0.4]$ is enough to fool a potential attacker who cannot train the model.

Selecting the static parameters from the four batch normalization layers whose histograms are displayed in Fig. 6 results in protecting 62 parameters (18 in the first layer, 7 in the second, 34 in the fourth and 3 in the last). We replace the selected γ parameters by uniformly generated ones in the range $[0, 0.4]$. This leads to the following accuracy:

– For $FAR = 10^{-4}$, $FRR = 0.28$ %
– For $FAR = 10^{-5}$, $FRR = 0.86$ %

Even though the drop in the accuracy is less drastic than in the previous experiment on a batch normalizations layer, the FRR for $FAR = 10^{-5}$ still corresponds to the accuracy at the end of the 8th training epoch.

Thus, this new method enables us to halve the number of parameters to protect while significantly dropping the accuracy (the accuracies obtained are summarized in Table 1).

The question that remains is whether defining static parameters as parameters that have not changed over the last 2 (or more) epochs would lead to an improved security.

Taking now into account the last 2 epochs, and considering 3 batch normalization layers, we have to protect 124 γ parameters. This leads to an accuracy of:

- For $FAR = 10^{-4}$, $FRR = 0.34$ %
- For $FAR = 10^{-5}$, $FRR = 0.99$ %

Since this new accuracy corresponds to the accuracy at the beginning of the 8th training epoch, we consider that the increased drop in the accuracy does not outweigh the increase in the number of parameters to protect. This confirms our choice of one epoch for the definition of static parameters.

4.3 Convolutional Layer

In this section, we explain how to further drop the accuracy of the degraded modes, while keeping around the same number of protected parameters.

Figure 1 shows how a convolutional layer computes the next layer's neurons. A convolutional filter is usually much smaller than the layer's input. Indeed, filters are usually 3×3 or 5×5 windows. On the other hand, when dealing with the Labeled Faces in the Wild (LFW) [13] data set, the model's input is commonly 250×250 images. Thus, each of the few parameters in a given filter impacts a large number of parameters. With the values considered, one filter value modification changes the value of $248 \times 248 = 61,504$ neurons from the following layer.

Therefore, even though convolutional layers have more parameters than batch normalization ones, we can still further limit the number of selected optimal parameters in the convolutional case.

Another element we need to take into account, however, is that the number of filters in a convolutional layer is usually high. For instance, if there are 3 input channels and 64 output channels, the layer stores $64 \times 3 = 192$ filters. Observing any drop in the accuracy requires a change in several such filters. For instance, feeding degraded values to all the parameters of only two filters among the 192 results in almost no drop in the accuracy. Given the explanation in the previous paragraph, the approach we consider is to randomly select one element among each set of *input_channels_number* filters. Thus, in the previous example, each output channel requires three filters. For each output channel, we randomly select one parameter among the three filters as an optimal parameter.

Furthermore, the depth of the selected convolutional layer matters. Indeed, if the said layer is among the first architecture layers, we can take advantage of the chain reaction. In a convolutional layer, each input neuron impacts several neurons in the following layer due to the way convolutions are computed. Each degraded filter parameter in the considered layer will change the value of a large number of neurons from the following layer, which, in turn, will impact several neurons in the layer after that, and so on. Given that our model is a convolutional neural network, most layers are convolutional ones. This explains why limiting ourselves to few parameters in one convolutional layer at the very beginning of the architecture can lead to a large drop in the accuracy.

Selecting a layer early in the architecture yields three other advantages:

1. the number of input channels is lower in the first layers (and only 3 in the first convolutional layer).
2. the number of output channels is lower in the first layers.
3. the input and output sizes are larger.

Points (1) and (2) ensure a minimal overall number of filter parameters for the considered convolutional layer. Point (3) results in a higher impact for every degraded filter parameter.

Finally, let us note that, given the fact we only consider one parameter per filter, we do not need to take into account the filter's parameters distribution: if the degraded parameters are in the range of possible values, the attacker cannot detect the degradation.

To summarize, the strategy to select the optimal parameters is as follows:

– Consider the model's first convolutional layer.
– For each output channel, select one element among the three filters for that channel.

In order to check that the first convolution has a higher impact on the predictions than batch normalization layers, we compare the sensitivity (as explained in Sect. 2.3) of the two strategies on a ResNet18 architecture trained on the CIFAR10 dataset [15]. Thus, we select one image (the second image from the CIFAR10 testing set for instance), and plot, on the one hand, the minimum δ one needs to add to all the γ parameters of each batch normalization layer in order to change the model's prediction (Fig. 7), and, on the other hand, the minimum δ to add to 64 parameters from the first convolutional layer, randomly selected according to our strategy (Fig. 8). Figure 8 shows that the peak sensitivity (over the various sets of parameters) for the convolutional layer considered is slightly lower than 0.1. On the other hand, changing all parameters from the various batch normalization layers shows that, for most layers, the sensitivity is higher than 1 (see Fig. 7). Since we are interested in a low sensitivity – meaning that a slight change in the protected parameters would lead to a significant change in the accuracy –, Fig. 7 and 8 confirm that the first convolutional parameters, selected according to our strategy, are more sensitive to small noise than batch normalization ones.

For our model, this strategy results in 64 selected parameters. As before, we can encode each parameter on 8 bits, thus leading to all the parameters being encoded on 512 bits overall. When we change 64 parameters from the model's first convolutional layer – selected as described previously – to random ones, we get the following accuracy:

– For $FAR = 10^{-4}$, $FRR = 0.34$ %
– For $FAR = 10^{-5}$, $FRR = 0.97$ %

Thus, with only half the parameters, we reach almost the same accuracy as in the batch normalization layer's case.

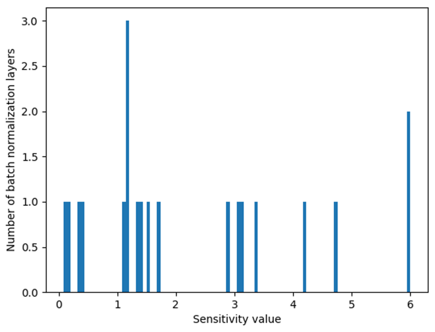

Fig. 7. Distribution of the sensitivity with respect to the second image of the CIFAR testing dataset over the batch normalization layers. For each batch normalization layer, the sensitivity Δ corresponds to the minimum value δ such that adding δ to all the γ parameters of the layer results in a change in the prediction. All Δ values greater than 6 are assimilated to 6.

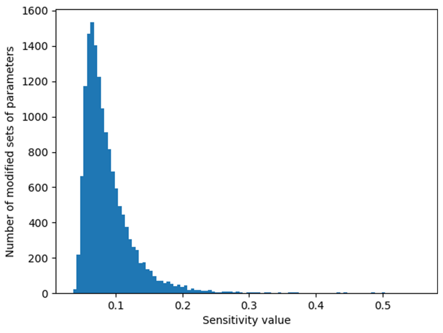

Fig. 8. Distribution of the sensitivity with relation to the second image of the CIFAR testing dataset over the first convolutional layer. 64 parameters are selected at random among the layer's parameters, as explained in Sect. 4.3. For each selected set of parameters, the sensitivity Δ corresponds to the minimal value δ such that adding δ to the selected parameters results in a change in the prediction.

Let us note that even though the absolute increase in the accuracy does not seem critical, it still corresponds to an accuracy obtained after only 8 epochs in our case (instead of the full 13 epochs training).

4.4 Exhaustive Search FAR and FRR

We place ourselves in the attacker's shoes. We consider that we have full access to the implementation of a degraded NN and, according to Sect. 3, we know how to compute parameters given a certain PIN. Thus, we generate random PINs and compute the accuracy – FRR for FAR – associated with the deduced parameters instead of the optimal ones.

The minimal accuracy the attacker gets is the following for degraded parameters from the first convolutional layer (selected as in Sect. 4.3):

– For $FAR = 10^{-4}$, $FRR = 0.30\%$, representing a 25% relative increase compared to the original model.
– For $FAR = 10^{-5}$, $FRR = 0.94\%$, representing a 34% relative increase compared to the original model.

On average, the attacker gets:

– For $FAR = 10^{-4}$, $FRR = 0.87\%$, representing a 263% relative increase compared to the original model.

Table 1. Accuracy of the original model and of the model where some parameters have been replaced by random ones

	Number of protected parameters	FRR ($FAR = 10^{-4}$)	FRR ($FAR = 10^{-5}$)
Original model	0	0.24 %	0.70 %
Modification of one batch normalization (beginning of Sect. 4.2)	128	0.32 %	0.96 %
Modification of dynamic parameters (end of Sect. 4.2)	62	0.28 %	0.86 %
Modification of convolutional parameters (Sect. 4.3)	64	0.34 %	0.97 %

– For $FAR = 10^{-5}$, $FRR = 2.58\%$, representing a 269% relative increase compared to the original model.

For the second strategy on the batch normalization layers (Sect. 4.2), the attacker gets, on average, the following accuracies:

– For $FAR = 10^{-4}$, $FRR = 0.28\%$, representing a 16% relative increase compared to the original model.
– For $FAR = 10^{-5}$, $FRR = 0.82\%$, representing a 17% relative increase compared to the original model.

To gauge the accuracy of our system, we use again the LFW database and a proprietary setup. Each try takes around 15 min.

5 Conclusion

We introduce a premium mode for NN applications, for instance in mobile phones. Our defense strategy is threefold:

– we rely on a PIN only known by privileged users;
– the functionality of the NN is degraded by default;
– the attacker does not have access to a training dataset and has a limited testing facility and is therefore forced to blindly guess the correct PIN.

Each privileged user benefits from a dedicated training of the NN and is given a PIN which enables him to switch from a degraded mode to the premium one.

These protections can also be enforced by classical anti-reversing engineering techniques as well as OS and software security features.

We explain how for a facial recognition NN with more than 14 million parameters, we determine 64 sensitive optimal values for our proposal, showing its practicability. We followed two approaches. The first consisted in looking for parameters affecting the accuracy, based on [23]. The second can be applied when the attacker does not have a training database, based on [6]. Further research could focus on how to systematically select a small set of parameters with a high impact on the NN's accuracy. As a first step, we would like to determine a correlation between accuracy and the number of protected parameters.

Acknowledgments. The authors want to thank Vincent Despiegel and his team for their support.

References

1. Android enterprise security white paper (2020)
2. Bayerl, S.P., et al.: Offline model guard: secure and private ML on mobile devices. In: 23. Design, Automation and Test in Europe Conference (DATE 2020) (2020). http://tubiblio.ulb.tu-darmstadt.de/117658/
3. Browne, M., Ghidary, S.S.: Convolutional neural networks for image processing: an application in robot vision. In: Gedeon, T.T.D., Fung, L.C.C. (eds.) AI 2003. LNCS (LNAI), vol. 2903, pp. 641–652. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-24581-0_55
4. Collberg, C.: Engineering code obfuscation. In: Advances in Cryptology - Eurocrypt 2016. Lecture Notes in Computer Science, vol. 9666, p. 1 (2016). https://www.iacr.org/cryptodb/data/paper.php?pubkey=28982. abstract of invited talk
5. Denil, M., Shakibi, B., Dinh, L., Ranzato, M., de Freitas, N.: Predicting parameters in deep learning. In: Neural Information Processing Systems (2013)
6. Denil, M., Shakibi, B., Dinh, L., Ranzato, M., de Freitas, N.: Predicting parameters in deep learning. In: Burges, C.J.C., Bottou, L., Ghahramani, Z., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5–8, 2013, pp. 2148–2156. Lake Tahoe, Nevada, United States (2013). http://papers.nips.cc/paper/5025-predicting-parameters-in-deep-learning
7. Frankle, J., Schwab, D.J., Morcos, A.S.: The early phase of neural network training. CoRR abs/2002.10365 (2020)
8. Gilbert, H.: On white-box cryptography (2016). invited talk
9. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27–30, 2016. pp. 770–778. IEEE Computer Society (2016). https://doi.org/10.1109/CVPR.2016.90
10. He, K., Zhang, X., Ren, S., Sun, J.: Identity mappings in deep residual networks. In: Leibe, B., Matas, J., Sebe, N., Welling, M. (eds.) ECCV 2016. LNCS, vol. 9908, pp. 630–645. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46493-0_38
11. Hua, W., Umar, M., Zhang, Z., Suh, G.E.: Guardnn: Secure DNN accelerator for privacy-preserving deep learning. CoRR abs/2008.11632 (2020). https://arxiv.org/abs/2008.11632
12. Hua, W., Umar, M., Zhang, Z., Suh, G.E.: Mgx: Near-zero overhead memory protection with an application to secure DNN acceleration. CoRR abs/2004.09679 (2020). https://arxiv.org/abs/2004.09679
13. Huang, G.B., Mattar, M., Berg, T., Learned-Miller, E.: Labeled faces in the wild: a database forstudying face recognition in unconstrained environments. In: Workshop on Faces in 'Real-Life' Images: Detection, Alignment, and Recognition (2008). https://hal.inria.fr/inria-00321923

14. Ioffe, S., Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: Bach, F.R., Blei, D.M. (eds.) Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6–11 July 2015. JMLR Workshop and Conference Proceedings, vol. 37, pp. 448–456. JMLR.org (2015). http://proceedings.mlr.press/v37/ioffe15.html
15. Krizhevsky, A.: Learning multiple layers of features from tiny images. Technical reports (2009)
16. Lambert, J., Sener, O., Savarese, S.: Deep learning under privileged information using heteroscedastic dropout. In: 2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18–22, 2018. pp. 8886–8895. IEEE Computer Society (2018). https://doi.org/10.1109/CVPR.2018.00926, http://openaccess.thecvf.com/content_cvpr_2018/html/Lambert_Deep_Learning_Under_CVPR_2018_paper.html
17. Lidl, R., Niederreiter, H.: Finite Fields. Cambridge University Press (1997)
18. Mayrhofer, R., Stoep, J.V., Brubaker, C., Kralevich, N.: The android platform security model. CoRR abs/1904.05572 (2019)
19. Mueller, B.: Hacking soft tokens - advanced reverse engineering on android (2016)
20. Shelhamer, E., Long, J., Darrell, T.: Fully convolutional networks for semantic segmentation. CoRR abs/1605.06211 (2016). http://arxiv.org/abs/1605.06211
21. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition (2015). http://arxiv.org/abs/1409.1556
22. Sultana, F., Sufian, A., Dutta, P.: Advancements in image classification using convolutional neural network. CoRR abs/1905.03288 (2019). http://arxiv.org/abs/1905.03288
23. Suri, A., Evans, D.: One neuron to fool them all. CoRR abs/2003.09372 (2020). https://arxiv.org/abs/2003.09372
24. Tople, S., Grover, K., Shinde, S., Bhagwan, R., Ramjee, R.: Privado: practical and secure DNN inference. CoRR abs/1810.00602 (2018). http://arxiv.org/abs/1810.00602
25. VanNostrand, P.M., Kyriazis, I., Cheng, M., Guo, T., Walls, R.J.: Confidential deep learning: Executing proprietary models on untrusted devices. CoRR abs/1908.10730 (2019). http://arxiv.org/abs/1908.10730
26. Vapnik, V., Vashist, A.: A new learning paradigm: learning using privileged information. Neural Networks **22**(5–6), 544–557 (2009). https://doi.org/10.1016/j.neunet.2009.06.042
27. Wang, M., Deng, W.: Deep face recognition: a survey. CoRR abs/1804.06655 (2018)
28. Wei, L., Liu, Y., Luo, B., Li, Y., Xu, Q.: I know what you see: Power side-channel attack on convolutional neural network accelerators. CoRR abs/1803.05847 (2018). http://arxiv.org/abs/1803.05847
29. Xiang, Y., et al.: Open DNN box by power side-channel attack. CoRR abs/1907.10406 (2019). http://arxiv.org/abs/1907.10406
30. Xu, M., Liu, J., Liu, Y., Lin, F.X., Liu, Y., Liu, X.: A first look at deep learning apps on smartphones. In: WWW. pp. 2125–2136. ACM (2019)