Extension of Model Extraction Using API Queries

Course Project for Course CS6570 - Secure Systems Engineering

Pranay Mathur*, Varun Bajpai†

*Department of Electrical Engineering, IIT Madras
ee22b132@smail.iitm.ac.in
†Department of Electrical Engineering, IIT Madras
ee22b152@smail.iitm.ac.in

***Abstract*—With the current rate of adoption of AI models in day-to-day activities, large amounts of money are spent in training and fine-tuning deployable and scalable models. Significant work has been done in analyzing the security of these models, and more specifically, security against inference-time and API-query attacks to extract model parameters. Previous works give an algorithm to extract parameters from neural networks with a single node output, trained with the ReLU activation function. In this report, we detail an extension of this attack to various other activation functions, as well to networks with multi-class outputs.**

I. Introduction

The task at hand was to go through a paper on ”Extracting Model Parameters using API queries” [1], and to extend the attack. The paper originally talks about extraction of ReLU neural networks with single class outputs. We have extended the attack to (1) other activation functions: LeakyReLU, tanh, sigmoid (2) multi-class output.

II. Analysis of the paper

At a high level, the complete attack consists of the following five steps:

1) Collect decision boundary points
2) Recover the normalized model signature
3) Recover weights layer by layer
4) Recover all the biases
5) Filter functionally inequivalent models

Consider a k-layer deep neural network $f_{\boldsymbol{\theta}}$. For an input $x \in \mathcal{X}$, $f_{\boldsymbol{\theta}}$ can be described as an affine transformation:

$$\begin{aligned} f_{\boldsymbol{\theta}}(x) &= A^{(k+1)} \cdots \left(I_P^{(1)}(A^{(1)}x + b^{(1)}) \right) \cdots + b^{(k+1)} \\ &= A^{(k+1)} I_P^{(k)} A^{(k)} \cdots I_P^{(2)} A^{(2)} I_P^{(1)} A^{(1)} x + B_P \\ &= \Gamma_P x + B_P \end{aligned}$$

For a k-deep neural network $f_\theta(x)$, the model signature denoted by $\mathcal{S}_\theta$ is the set of affine transformations

$$\mathcal{S}_\theta = \{(\Gamma_{\mathcal{P}}, B_{\mathcal{P}}) \text{ for all the } \mathcal{P}\text{'s}\}\,.$$

Here, $\mathcal{P}$ denotes the activation pattern, and describes which neurons are active (have positive output) and which are inactive (output 0) post-activation.

The attack stated in the paper incorporates the on/off nature of the ReLU activation function in the I_P matrices, which are 0/1 diagonal matrices. So any changes in the activation functions should correspond to changes in the implementation of model signatures.

III. Extension to Activation functions

A. LeakyReLU

The paper had already implemented the attack for the ReLU activation function. We extended the attack to the LeakyReLU activation.

1) Differences Between ReLU and LeakyReLU:

$$\text{ReLU}(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

$$\text{LeakyReLU}(x) = \begin{cases} \alpha x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

Here α is a small positive constant (e.g., 0.01). As we can see from the definitions, ReLU and LeakyReLU behave identically for non-negative inputs and differ only for negative inputs.

2) Approach:
The original attack relies on a diagonal matrix populated with 1s and 0s — where 1 corresponds to the slope used for positive inputs and 0 for negative inputs. For LeakyReLU, this matrix needs to be modified to consist of 1s and αs, where α is the slope for negative inputs.

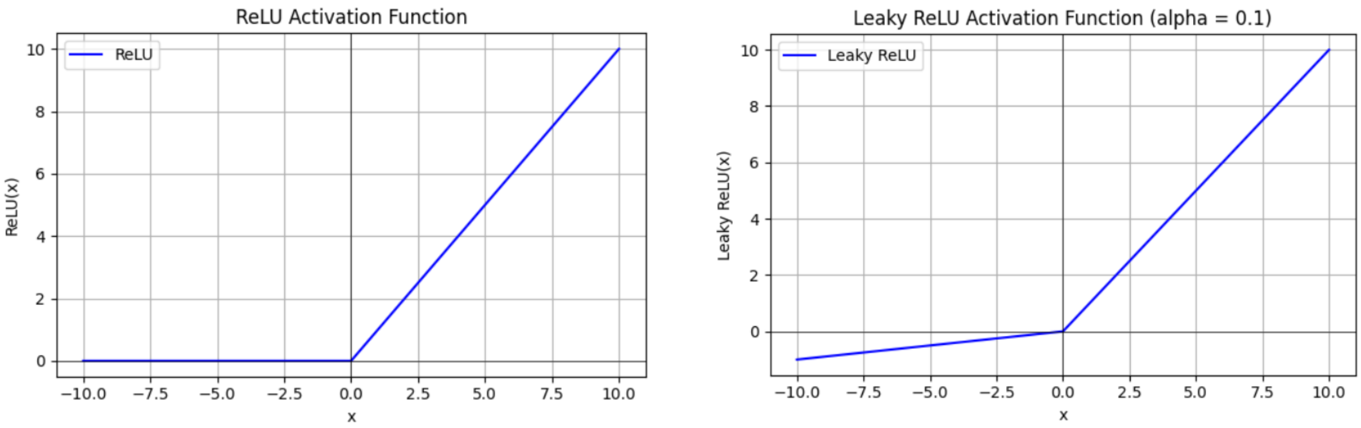
For this particular activation function, we can retain the original definition of the activity of a neuron, which states that a neuron is active if it's pre-activation output is positive, and inactive otherwise.

We define $I_{\mathcal{P}}^{(i)'} \in \mathbb{R}^{d_i \times d_i}$ to be a α-1 diagonal matrix, with α at inactive neurons, thus modeling the transformation again as:

$$\begin{aligned} f_{\boldsymbol{\theta}}(x) &= A^{(k+1)} \cdots \left(I_P^{(1)'}(A^{(1)}x + b^{(1)}) \right) \cdots + b^{(k+1)} \\ &= A^{(k+1)} I_P^{(k)'} A^{(k)} \cdots I_P^{(2)'} A^{(2)} I_P^{(1)'} A^{(1)} x + B_P \\ &= \Gamma^{'}_P x + B_P \end{aligned}$$

Here, due to the changes in definition of $I_{\mathcal{P}}^{(i)}$, the $\Gamma^{'}_P x$ vector changes. While extracting the weights for an intermediate layer, the matrices in the expressions also change:

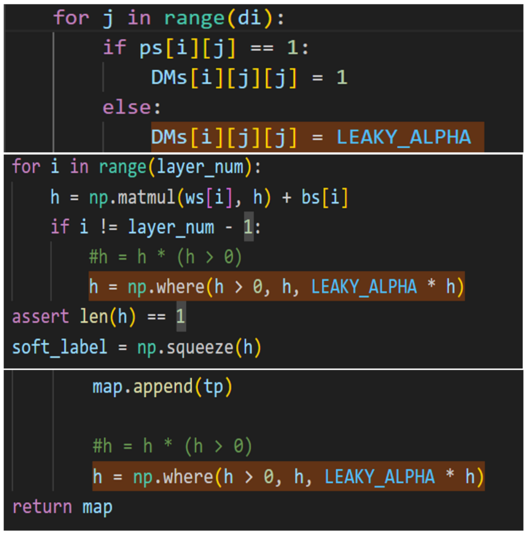
$$f_\theta(x) = \Gamma^{'}_{\mathcal{P}} \cdot x + B_{\mathcal{P}} = \mathcal{G}_j^{(i)'} A_j^{(i)'} \cdot C^{(i-1)'} \cdot x + B_{\mathcal{P}},$$



(a) ReLU

(b) Leaky ReLU

Fig. 1: Graphs for ReLU and LeakyReLU



```
    for j in range(di):
        if ps[i][j] == 1:
            DMs[i][j][j] = 1
        else:
            DMs[i][j][j] = LEAKY_ALPHA
```

```
for i in range(layer_num):
    h = np.matmul(ws[i], h) + bs[i]
    if i != layer_num - 1:
        #h = h * (h > 0)
        h = np.where(h > 0, h, LEAKY_ALPHA * h)
assert len(h) == 1
soft_label = np.squeeze(h)
```

```
        map.append(tp)

        #h = h * (h > 0)
        h = np.where(h > 0, h, LEAKY_ALPHA * h)
return map
```

Fig. 2: Changes for LeakyReLU

However, the algorithm which extracts specific sort of boundary points will work for these definition modifications. Although the values change, the structure of the equations and thus the overall attack remains the same. Since this attack relies on first extraction of Γ_P, and then the biases for each layer via a system of equations, the values of the matrices do not matter as long as I am able to express the transform in terms of these matrices. Thus, just by making this modification to the overall attack is sufficient to extend the above attack to networks trained on the LeakyReLU activation.

3) *Differences in Code:*

To implement the same, we had to modify the given code accordingly. Wherever a forward pass occurred in the code, we ensured that 0s were replaced with α. There were two primary locations where the forward pass occurs: in the functions `f_cheat()` and `get_maps()`. In both functions, we modified the lines given in the Fig. 2. The changes in figure two are in the `compare_model_signatures()`, `f_cheat()` and `get_map()` functions respectively.

Additionally, while verifying the correct model signature, the comparison matrix also had to be updated to contain 1s and α. This change ensures that the model correctly reflects the LeakyReLU behavior.

B. *Tanh*

1) *Differences Between ReLU and Tanh:*

$$\text{ReLU}(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

Tanh(x) is defined as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

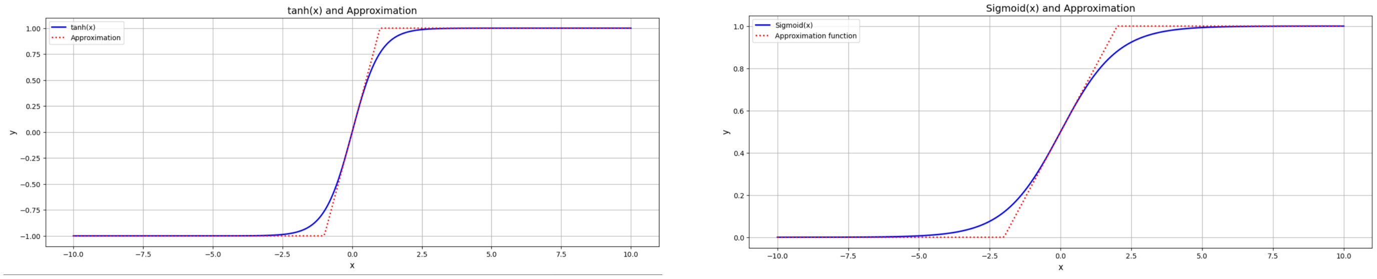
In the attack, the paper uses the fact that ReLU is piecewise linear. Extending to tanh, we can make an approximation of the tanh function:

$$\tanh(x) \approx \begin{cases} -1 & \text{if } x < -1 \\ x & \text{if } |x| \leq 1 \\ 1 & \text{if } x > 1 \end{cases}$$

2) *Approach:*

Extending the attack to tanh is not as trivial as the extension to LeakyReLU. To facilitate this extension, we first modify the definition of an active neuron. We will consider a neuron to be active if it's pre-activation value lies between -1 and 1, and consider it to be inactive otherwise.

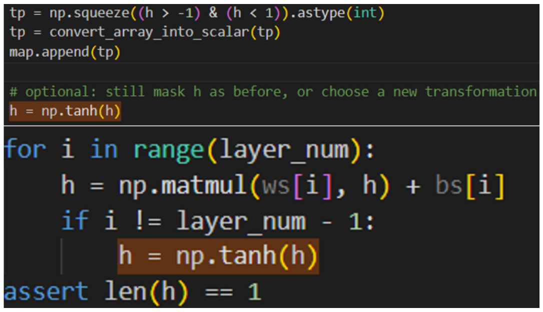
We shall, as done in the original paper, define $I_{\mathcal{P}}^{(i)} \in \mathbb{R}^{d_i \times d_i}$ to be 0-1 diagonal matrices, with values being 1 for active neurons, and 0 otherwise. However, to account for the DC value that we need to add for inactive neurons, we introduce a bias vector during activation as $V_{\mathcal{P}}^{(i)} \in \mathbb{R}^{d_i}$. The values of



(a) tanh(x) with approximation

(b) sigmoid with approximation

Fig. 3: Graphs for tanh and sigmoid



```
tp = np.squeeze((h > -1) & (h < 1)).astype(int)
tp = convert_array_into_scalar(tp)
map.append(tp)

# optional: still mask h as before, or choose a new transformation
h = np.tanh(h)
```

```
for i in range(layer_num):
    h = np.matmul(ws[i], h) + bs[i]
    if i != layer_num - 1:
        h = np.tanh(h)
assert len(h) == 1
```

Fig. 4: Changes for tanh

the bias vector would be -1 for neural values < -1, and 1 for neural values > 1. The expression will now look as follows:

$$
\begin{aligned}
f_{\boldsymbol{\theta}}(x) &= A^{(k+1)} \cdots \left(I_P^{(1)} (A^{(1)} x + b^{(1)}) + V_P^{(1)} \right) \cdots + b^{(k+1)} \\
&= \Gamma'_P x + B'_P
\end{aligned}
$$

Expanding this, we notice that both $\Gamma'_P x$ and $B'_P x$ have different equations than the ones originally for ReLU. Similar modifications will be seen for intermediate states:

$$
f_\theta(x) = \Gamma'_{\mathcal{P}} \cdot x + B'_{\mathcal{P}} = \mathcal{G}_j^{(i)'} A_j^{(i)'} \cdot C^{(i-1)'} \cdot x + B'_{\mathcal{P}},
$$

However, the algorithm for extracting the weights and biases in a layer based on detecting decision boundary points and identifying different activation patterns will still work, due to the fact that even after modifications, we can still express our network in terms of similar matrices.

3) *Differences in Code:*

There were two primary locations where the forward pass occurs: in the functions `f_cheat()` and `get_maps()`. In both functions, we modified the lines as given in Fig. 4 which have the changes in `get_map()` and `f_cheat()` respectively.

We did not need to change the matrix in the `compare_model_signatures()` function.

C. *Sigmoid*

1) *Differences Between ReLU and Sigmoid:*

$$
\text{ReLU}(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}
$$

$$
\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}
$$

Similar to tanh, in sigmoid, we again made well established approximations to make the sigmoid function piecewise linear.

$$
\text{sigmoid}(x) \approx \begin{cases} 0 & \text{if } x < -2 \\ 0.25x + 0.5 & \text{if } -2 \leq x \leq 2 \\ 1 & \text{if } x > 2 \end{cases}
$$

2) *Approach:*

Extending the attack to sigmoid requires us to make the most modifications. Firstly, we modify the definition of an active neuron. We will consider a neuron to be active if it's pre-activation value lies between -2 and 2, and consider it to be inactive otherwise.

We shall now define $I_{\mathcal{P}}^{(i)'} \in \mathbb{R}^{d_i \times d_i}$ to be 0-0.25 diagonal matrices, with values being 0.25 for active neurons, and 0 otherwise. This is evident as:

$$
\frac{d}{dx}\sigma(x) = \begin{cases} 0.25, & \text{if } |x| < 2 \\ 0, & \text{otherwise} \end{cases}
$$

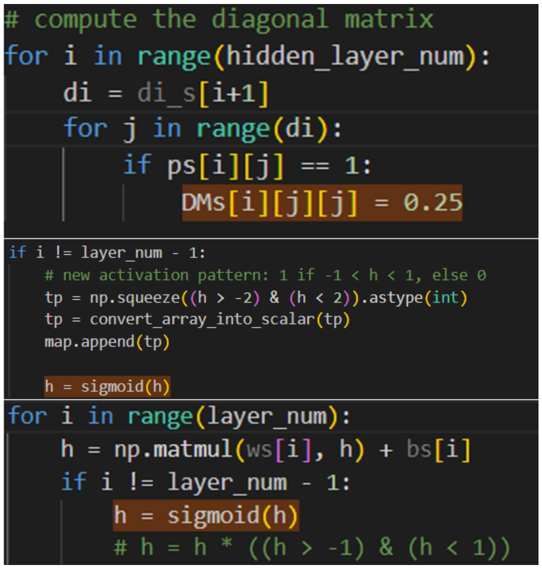
Again, to account for the DC value that we need to add for inactive neurons, we introduce a bias vector during activation as $\mathcal{V}_{\mathcal{P}}^{(i)} \in \mathbb{R}^{d_i}$. The values of the bias vector would be 0 for neural values < -2, and 1 for neural values > 2. The expression will now look as follows:

$$
\begin{aligned}
f_{\boldsymbol{\theta}}(x) &= A^{(k+1)} \cdots \left(I_P^{(1)'} (A^{(1)} x + b^{(1)}) + V_P^{(1)} \right) \cdots + b^{(k+1)} \\
&= \Gamma'_P x + B'_P
\end{aligned}
$$

Again, both $\Gamma'_P x$ and $B'_P x$ have different equations than the ones originally for ReLU. Similar modifications will be seen for intermediate states:

$$
f_\theta(x) = \Gamma'_{\mathcal{P}} \cdot x + B'_{\mathcal{P}} = \mathcal{G}_j^{(i)'} A_j^{(i)'} \cdot C^{(i-1)'} \cdot x + B'_{\mathcal{P}},
$$

However, as argued above, as we are able to express the neural transformation in the same form as done in the paper, the exact same algorithm for extraction of parameters layer-wise will for for us too.



```
# compute the diagonal matrix
for i in range(hidden_layer_num):
    di = di_s[i+1]
    for j in range(di):
        if ps[i][j] == 1:
            DMs[i][j][j] = 0.25
```

```
if i != layer_num - 1:
    # new activation pattern: 1 if -1 < h < 1, else 0
    tp = np.squeeze((h > -2) & (h < 2)).astype(int)
    tp = convert_array_into_scalar(tp)
    map.append(tp)

    h = sigmoid(h)
```

```
for i in range(layer_num):
    h = np.matmul(ws[i], h) + bs[i]
    if i != layer_num - 1:
        h = sigmoid(h)
        # h = h * ((h > -1) & (h < 1))
```

Fig. 5: Changes for sigmoid

3) Differences in Code:

We had to make changes in 3 functions, namely `f_cheat()`, `get_maps()` and `compare_model_signatures()`. The lines which are shown in Fig. 5 were changed. This shows the changes in the `compare_model_signatures()`, `get_map()`, and `f_cheat()` functions respectively.

IV. Extension to multi-class classifiers

The attack in the paper worked only for one neuron in the output layer. In this section we have extended this to multi-neurons in the output layer and thus multiple classes.

A. Approach to extension

The attack works on the *relative* values of the neurons rather than their absolute values. For example, between two neurons in a layer, if one neuron has a relatively larger value, that neuron outputs 1. Thus, the attack reduces to extracting the relative weights between pairs of neurons.

Let us consider two neurons in the output layer with weights connected to them denoted as w_1 and w_2. If we can extract the difference $w_1 - w_2$, we can simply initialize w_1 to some value and obtain w_2 accordingly.

We can model the neural network such that the weights represent the difference between the actual weights. This allows us to use the existing attack to extract the parameters of this *pseudo-model* of the neural network. Once we have the parameters, we can initialize w_1 and recover w_2 with respect to w_1.

This approach can be extended to three or more neurons by following the same procedure as for two neurons.

B. Mathematical Formulation

We can formalize the above as follows for a 1-layer deep neural network. Let the model be expressed as:

$$A^{(2)}\left(\sigma(A^{(1)}\bar{x} + b^{(1)})\right) + b^{(2)} = \bar{y}$$

For a k-class classifier, we have $\bar{y} \in \mathbb{R}^k$. Let us define y_j as output of the j^{th} node, $A_j^{(2)}$ as the j^{th} row of the weight matrix, and $b_j^{(2)}$ as the j^{th} element of the bias vector.

$$y_j = A_j^{(2)}\left(\sigma(A^{(1)}\bar{x} + b^{(1)})\right) + b_j^{(2)}$$

Let us take y_1 as the reference. For all $j = 2, 3, \ldots, k$, let us consider $k-1$ models as follows:

$$y_1 - y_j = (A_1^{(2)} - A_j^{(2)})\left(\sigma(A^{(1)}x + b^{(1)})\right) + (b_1^{(2)} - b_j^{(2)})$$

This is a single output network. For examples belonging to class 1, the we have $y_1 > y_j$, resulting in a positive output. Similarly, for examples belonging to class j, we have $y_1 < y_j$, resulting in a negative output. This results in a network of the desired format, and we can apply the extraction outlined in the original paper to extract the parameters.

Once we have extracted the parameters from the k-1 models, an initialization of $A_1^{(2)}$ and $b_1^{(2)}$ with random values will give us values for all other weight vectors and biases. Thus, we are able to successfully extract $A^{(2)}$ and $b^{(2)}$, completing the attack.

V. Implementation and Related Issues

- The code provided in the repository was not able to extract the parameters of models which we initially trained, although they had the same number of weights and biases as the models that were present in the repository.
- To the best of our knowledge, this is due to some preprocessing step which is neither known nor documented in the paper.
- We normalised the inputs and then divided them by 100. After this pre-processing of the images, the model extraction works around 50% of the time. This was the best performance of the provided code on our custom trained ReLU models with 1 hidden layer, so we took this as a baseline. Further attacks were compared against this baseline, and were considered ”successful” if they were able to work around 50% of the time.
- The randomness of success comes from the randomness in selection of starting point and direction to find the boundary points.
- While we were trying to extend to other activation functions, even when enough boundary points were found, the code would filter out the proposed models initially majorly stating that there was a mismatch in the signs.
- To overcome this problem we tried the max voting technique. Initially, the model would look at every decision boundary point one by one, and extract the weight signs and then the parameters for each. We modified this to making the model first go through all boundary points, obtain weight signs on the basis of each of them, and then use a maximum voting scheme to finalize the sign of each weight. Once signs were finalized, model parameters were extracted normally.

TABLE I: Comparison of Attack Performances for Various Activation Functions

Activation Function	**Precision**	**L1 Error**	**MS**	**PMR**
ReLU	10^{-12}	10^{-2}	10^{-4}	1
Leaky ReLU	10^{-12}	10^{-2}	10^{-6}	0.993534
Tanh	10^{-12}	10^{-2}	10^{-4}	0.998092
Sigmoid	10^{-12}	10^{-2}	10^{-4}	0.987639

- This made the code slower and did not really solve the problem, so we decided to revert back to the original method of determining the weight signs.
- We then started tweaking the hyper-parameters. There were 3 major hyper-parameters for the whole attack, namely Precision, L1-error and ms (step size).
- Precision is the preset precision of the moving stride, and relates to how precise the binary search is and how close the result is to the boundary point.
- L1_error is used to filter out unequal gamma's.
- MS is the step-size while calculating weight signs.
- For the LeakyReLU extraction, not much tweaking of hyerparameters was reqired, and the original values gave good results.
- However, for tanh and sigmoid, we had to modify ms to 10^{-4} to get good results.
- Their code for extraction of custom ReLU networks with 2 hidden layers was only working with their models and not ours, so we weren't able to show outputs for the extension of the attack to other activation functions for deeper networks. However, we are confident that since the manipulations and related arguments are logically and mathematically sound, our extensions will work for the same.
- Additionally, increasing the size of hidden state to even 4 resulted in the original attack taking a lot of time, so we have not performed extensive studies for our attacks on the same.
- For extending to multi class output, as argued above, we have theoretically extended the attack to an arbitrary k-class classifier.
- We have successfully extracted a model with 2 classes in its output layer.
- The hyperparameters used for the same were precision = 10^{-16}, l1_error = 10^{-6}, ms = 10^{-4}
- However, the code in the repository was unable to extract subtracted weights for a 3-class classifier. Having robust attack algorithm in the base paper would help us greatly in showing results for larger networks and extended attacks.
- If we try to extract a different model which was trained taking the exact same parameters and hyper-parameters, there are times when the original code from the repository is not able to extract the different model. This inability to extract a different model trained on the same basis has extended to our attack for the different activation functions as well.
- Our implementation is in this GitHub repository.

VI. Acknowledgements

We would like to thank our professor Chester Rebeiro for this opportunity to take up this project. We would also like to thank our TA Reetwik Das for guiding us through the process. We would also like to acknowledge our usage of AI tools such as ChatGPT to help us understand some parts of the research paper and the code.

References

[1] Y. Chen, X. Dong, J. Guo, Y. Shen, A. Wang, and X. Wang, "Hard-label cryptanalytic extraction of neural network models," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur. (ASIACRYPT)*, 2024, pp. 1–20.