ShadowNet: A Secure and Efficient System for On-device Model Inference

Zhichuang Sun
Northeastern University

Ruimin Sun
Northeastern University

Long Lu
Northeastern University

Somesh Jha
University of Wisconsin, Madison

Abstract

On-device machine learning (ML) is getting more and more popular as fast evolving AI accelerators emerge on mobile and edge devices. Consequently, thousands of proprietary ML models are being deployed on billions of untrusted devices, raising a serious security concern about the model privacy. However, how to protect the model privacy without losing access to the AI accelerators is a challenging problem. This paper presents a novel on-device model inference system called ShadowNet, which protects the model privacy with TEE while securely outsourcing the heavy linear layers of the model onto the hardware accelerators without trusting them. ShadowNet achieves it by transforming the weights of the linear layers before outsourcing them and restoring the results inside the TEE. The nonlinear layers are also kept secure inside the TEE. The transformation of the weights and the restoration of the results are designed in a way that can be implemented efficiently. We have built a ShadowNet prototype based on TensorFlow Lite and applied it on three popular CNNs including AlexNet, MiniVGG and MobileNets. Our evaluation shows that the ShadowNet transformed models have comparable performance with the original models, offering a promising solution for secure on-device model inference.

1 Introduction

On-device machine learning is gaining popularity with more and more AI accelerators on mobile and embedded devices, like NPU, GPU and Edge TPU [5]. Recent study [36] shows that thousands of mobile apps are using on-device machine learning for various purposes, including OCR, face recognition, liveness detection, ID card and bank card recognition, translation, and so on. The benefits of on-device machine learning are obvious. It avoids sending user private data onto the cloud, saves the latency of sending data back-and-forth, and does not require a network connection. Many ML apps even use on-device ML for real-time tasks like rendering a video stream lively, which is not possible with the traditional cloud-based machine learning on mobile devices.

However, with thousands of private models deployed on billions of untrusted mobile devices, the attackers are not only technically capable of but also financially motivated to steal the models [36,41]. Leakage of those proprietary models can cause severe financial loss to the businesses. For example, good models help companies maintain a competitive advantage over their competitors. Training models also involves a large amount of engineering effort in data collection and labeling, as well as parameters tuning.

What's worse, existing proprietary models are found to be not well protected. As shown by [36], 41% of the models are stored in plaintext and can be downloaded along with the app packages. For those apps that protect the models (e.g., encrypting the models), they are still suffering from unsophisticated runtime attacks that can extract the decrypted models from the memory. With 54%

ML apps using GPU for acceleration, it becomes even more complicated to protect the model inference without losing access to the GPU accelerations.

So far, there are generally two lines of research in protecting the model inference: Trusted Execution Environment (TEE)-based approaches and cryptography-based approaches. Both of them face the unique challenges of on-device model inference. First of all, the TEE on mobile devices is designed for small critical tasks, like key management; while model inference is usually very resource demanding. The TEE may not have enough resources, like secure memory, to support model inference. What's more, simply moving the model inference into the TEE will greatly increase the TCB size of the TEE, and lose access to the hardware accelerators like GPU at the same time. For cryptography-based approaches, they either use Homomorphic Encryption(HE) or Multi-Party Computation(MPC) [9, 21, 26, 34] to secure model inference. However, Homomorphic Encryption based model inference are orders of magnitude slower than the state-of-the-art model inference. MPC based approaches involve multiple participants which require network connection, thus not suitable for real-time tasks or offline usage.

To this end, we design our own scheme ShadowNet, which transforms the weights of the linear layers before outsourcing them to the untrusted world for acceleration, and restores the results inside the TEE. The nonlinear layers are also kept secure inside the TEE. The key idea of ShadowNet is based on the observation that the linear layers of CNNs usually take up 90% of the computation in the whole network. With ShadowNet, we can outsource the heavy linear layers to the untrusted world (including GPU) without leaking the model weights. Our security analysis shows that, ShadowNet does not give the attacker any advantage in learning the model weights.

We have built a prototype of ShadowNet based on TensorFlow Lite and OP-TEE OS, and applied it on several popular CNNs including AlexNet, MiniVGG and MobileNets. Our evaluation shows that the ShadowNet transformed models have comparable performance with the original models, increasing the model inference time for AlexNet by 22% at best and for MobileNets by 63% at worst. Comparing with the crypto-based approaches which usually are orders of magnitude slower, ShadowNet provides a practical solution for securing the on-device model inference. We have also explored a variant of ShadowNet scheme, called Layerwise ShadowNet, which applies the ShadowNet transformation on a few selected layers to offer a trade-off between security and performance. Layerwise ShadowNet benefits significantly from the GPU acceleration and reduces the model inference time by as much as 59ms(26%) for AlexNet. In summary, this paper makes the following contributions:

- We designed a novel on-device model inference scheme ShadowNet, which can protect the model weights privacy with TEE while leveraging the untrusted hardware accelerators.
- We built an end-to-end ShadowNet model inference system based on TensorFlow Lite to demonstrate the practicality. We also greatly optimized the performance of model inference inside the TEE with small TCB size.
- We performed a thorough security analysis on the ShadowNet scheme with both theoretical and empirical results, analyzing the security under various attack scenarios.
- We evaluated the ShadowNet scheme on three popular CNNs including AlexNet, MiniVGG, and MobileNets. Our evaluation shows that the ShadowNet models has comparable performance as the original models.

2 Background

2.1 Convolutional Neural Network

Convolutional Neural Network (CNN) is a class of deep neural networks that is commonly applied to analyzing visual images. A CNN consists of an input and an output layer, with a sequence of linear and nonlinear layers stacked in between. The linear layers include the convolutional layer and fully connected layer. In addition, MobileNets introduces two new type of linear layers: Pointwise Convolution and Depthwise Convolution. The nonlinear layers include the activation layers and pooling layers.

ShadowNet is based on the observation that most of the computation and the model weights are from the linear layers. ShadowNet applies linear transformation on the weights of the linear layers to obfuscate them, so that the heavy linear layers can be outsourced to the untrusted hardware accelerators without leaking the original weights. The nonlinear layers remain unchanged and are kept inside the TEE.

2.2 Trusted Execution Environment

A trusted execution environment (TEE) is a secure area of a main processor. It guarantees that the code and data loaded inside to be protected with respect to confidentiality and integrity. [14]

Arm TrustZone [3] is a popular TEE implementations for mobile devices. It is a hardware feature available on both Cortex-A processors [12] (for mobile and high-end IoT devices) and Cortex-M processors [13](for low-cost embedded systems). TrustZone renders a so-called "Secure World", an isolated environment with tagged caches, banked registers, and private memory for securely executing a stack of trusted software, including a tiny OS and trusted applications (TA). In parallel runs the so-called "Normal World", which contains the regular/untrusted software stack. Code in the Normal World, called client applications (CA), can invoke TAs in the Secure World. A typical use of TrustZone involves a CA requesting a sensitive service from a corresponding TA, such as signing or encrypting a piece of data. Arm TrustZone has been widely used for security critical services like key management and Digital Rights Management(DRM) on smartphones.

OP-TEE(Open Portable Trusted Execution Environment) [30] is an open-source trusted OS running inside Arm TrustZone. It supports a broad type of mobile devices from Arm Juno Board, Raspberry Pi 3, to a series of Hikey boards. It is also integrated with AOSP to run along with Android OS. OP-TEE OS usually reserves a small part of DRAM, e.g. 32MB, as secure memory so as to minimize the performance impact on the Normal World applications.

We choose Hikey960 [1] as our development board which has Arm TrustZone support. It runs Android P as the Normal World OS and OP-TEE as the Secure World OS [8]. ShadowNet runs the transformed linear layers of the model in the Normal World, uses a CA to communicate with the TA in the Secure World, which runs the other layers of the model securely.

3 Overview

3.1 Design goals

We want to design a secure on-device model inference system that meets the following goals:

- Security rooted in the hardware, so that the models will be secure even when the OS is compromised;
- Reasonable performance overhead, making sure it will not break existing real-time tasks;

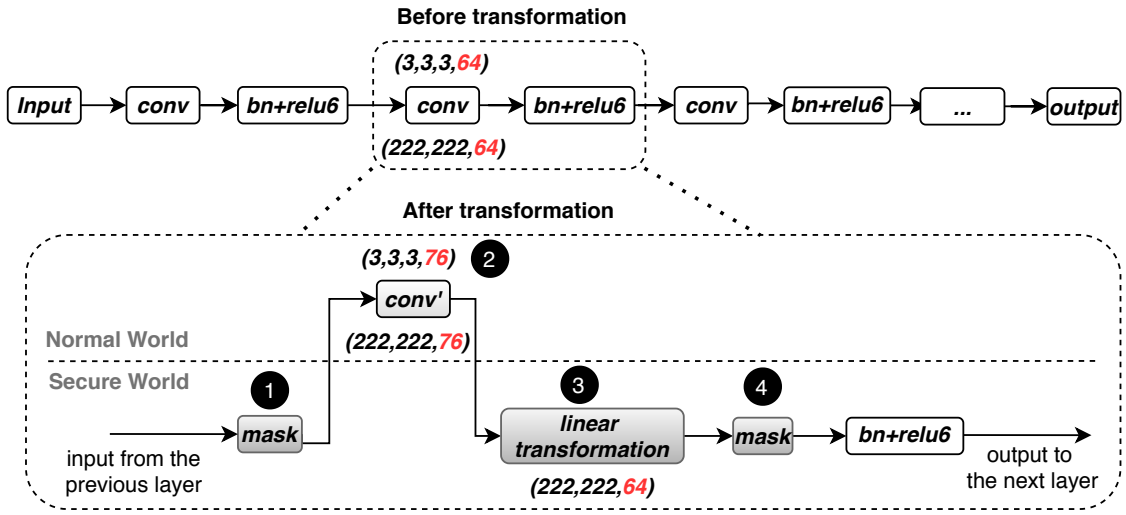


Figure 1: An overview of ShadowNet transformation on a simple CNN.

This CNN is a stack of convolutional layers, and each Convolution layer (*conv*) is followed by a Batchnorm(*bn*) layer and a ReLU6(*relu6*) activation layer. The shapes of the weights are marked on the top of the box, and the shapes of the outputs are marked under the box. The red color indicates the shape change after transformation. For each Convolution layer (*conv*), the ShadowNet transformation works in four steps marked above. After the transformation, the *conv'* runs in the Normal World, the other layers run in the Secure World.

- Access to hardware accelerators, which are being developed and used for on-device ML tasks;

3.2 The threat model

We assume a strong attacker that takes control of the Normal World including the OS, observing everything exposed to the Normal World on the device (including the GPU tasks). We consider model privacy as our goal, and do not consider model inference integrity for this work. Model inference integrity can be achieved with the verification idea proposed in Slalom [37]. We also do not consider side channel attacks on the TEE. We assume that the TEE can protect the confidentiality and integrity of the program and data inside it.

3.3 Design challenges

TEE provides hardware-level security, while using the mobile TEE for secure model inference has several technical challenges. First, the mobile TEE, like Arm TrustZone, is designed for small security critical services like managing encryption keys. The memory reserved for the TEE OS is limited. For example, only 14 MB is available for Trusted Applications of OP-TEE OS on Hikey960 Dev Board, while the model size of AlexNet is 242MB. It is simply not feasible to run the high resource-demanding model inference inside the TEE. Second, the current TEE does not include GPU/NPU into the secure domain, so we will also lose access to the hardware acceleration. Third, the model inference framework will also greatly increase the TCB of TEE, risking the security of the whole system.

3.4 Our solution: ShadowNet

The idea of ShadowNet: The key observation of ShadowNet is shared with the previous research like Slalom [37] that the linear layers of CNNs occupy the majority of the model parameters and

the model inference time. For example, the linear layers of MobileNets occupy around 95% of the model parameters, 99% of the model inference time. The idea of ShadowNet is to apply linear transformation on the weights of the linear layers and outsource them onto the untrusted world, so that we can leverage the hardware acceleration without trusting it. ShadowNet restores the results inside the TEE. The other nonlinear layers are also kept secure inside the TEE.

With this design, ShadowNet's security is rooted in the TEE, meeting the first design goal; ShadowNet does not introduce any heavy cryptographic operations, and our evaluation shows that ShadowNet has reasonable overhead, meeting the second design goal; ShadowNet is still able to use the hardware acceleration meeting the third design goal. At the same time, ShadowNet solves the technical challenges of mobile TEE by keeping the TCB size small and the TEE memory usage low.

An application example: We use a simple example to show how ShadowNet works on a typical CNN, as shown in Figure 1. The example CNN is a stack of convolutional layers, and each Convolution layer (*conv*) is followed by a Batchnorm(*bn*) layer and a ReLU6(*relu6*) activation layer.

For each Convolution layer *conv*, the ShadowNet transformation works in four steps: (1) add a mask layer to the input;(2) replace the original *conv* layer with a transformed *conv'* layer;(3) add a *linear transformation* layer to restore the result of the *conv'*; (4) unmask the input. The combination of *conv'+linear transformation* is equivalent to the *conv* in the original CNN. The combination of *mask+conv'+linear transformation+mask* is also equivalent to the *conv* in the original CNN. The Batchnorm layer and ReLU6 layer remain unchanged.

The *mask* layers in step (1) and step (4) are introduced to protect the input/output privacy of the outsourced linear layers, so that the attacker can not observe the original input and output. We discuss how the mask layer is implemented in Section 4.2, and why the mask layer is needed in our security analysis in Section 6.4. Step (3) and step (4) shows the high level ideas of how ShadowNet works on a convolution layer. Notice that, the *conv'* layer has 76 kernels instead of 64 kernels. This number is related to the obfuscation ratio and it is configurable. In Section 4.1, we will explain the choice of this number and how to generate the weights for *conv'* layer and *linear transform* layer. We will also discuss how ShadowNet transforms the other type of linear layers like Pointwise Convolution, Depthwise Convolution and Dense/Fully Connected.

In summary, ShadowNet offers a novel model inference system that can protect the model weights with the TEE while leveraging the untrusted hardware for acceleration. ShadowNet achieves its goal by transforming the heavy linear layers' weights and masking their input before outsourcing them to the untrusted world, then restoring the results inside the TEE. The nonlinear layers are also kept secure inside the TEE.

4 ShadowNet scheme

In this section, we introduce the ShadowNet scheme. We first explain how to apply linear transformation on a broad class of linear layers, including Convolution, Depthwise Convolution, Pointwise Convolution and Dense/Fully Connected. Subsequently, we discuss the layer input/output privacy by introducing the mask layer. We discuss the scheme optimization at the end.

4.1 Transformation on linear layers

ShadowNet relies on linear transformation to obfuscate the weights of linear layers. We base the foundation of ShadowNet on the mathematical property of linear transformation, which guarantees that the ShadowNet scheme will not change the output of the original network. We discuss its application on a few common types of linear operations used in CNNs.

Property of linear transformation: Mathematically, linear transformation is a function f defined on vector space V and W over the same field R, $f : V \to W$. In our scenario, R is the field of real number. For any two vectors $u, v \in V$ and any scalar $c \in K$, the following two conditions are satisfied:

$$\begin{aligned} &additivity : f(u+v) = f(u) + f(v) \\ &homogeneity : f(cu) = cf(u) \end{aligned} \tag{1}$$

Convolutional Layer: The convolutional layer is the core building block of a CNN. The parameters of the convolutional layer consists of a set of learnable filters (or kernels). Each convolutional kernel is defined by the width, height and depth of the receptive field. The depth must be equal to the number of channels (depth) of the input feature map. For the *conv* layer from our example CNN in Figure 1, the input shape is (224, 224, 3) where 3 is the depth. The convolutional layer of the example CNN has 64 filters. The shape of the convolutional kernel is (3, 3, 3), corresponds to the receptive field's height, width and depth.

Mathematically speaking, the convolution operation on a given image I with filter K can be described as below, where He, Wi, Ch represents the height, width and depth of the filter K, and x, y refer to the coordinates in the two-dimensional output feature map.

$$Conv(I,K)_{x,y} = \sum_{i=1}^{He} \sum_{j=1}^{Wi} \sum_{k=1}^{Ch} K_{i,j,k} I_{x+i-1,y+j-1,k} \tag{2}$$

Before transformation, the convolutional layer can be described as follows, where X and Y denote the input and output, W is a group of filters.

$$Y = Conv(X, W^T) \tag{3}$$

The linear transformation on the convolutional layer works as follows. Assume that the weights W comprises n convolutional kernels, where $W = [w_1, \ldots, w_n]^T$. We define a linear transformation f from n-dimensional vector space V to m-dimensional vector space P, namely $f : V \to P$. The vector after transformation is T in space P, where $T = [t_1, \ldots, t_m]^T$.

$$\begin{aligned} &(t_1, t_2, \cdots, t_m) = (w_1, w_2, \cdots, w_n) \begin{bmatrix} \lambda_{11}, \lambda_{12}, \cdots, \lambda_{1m} \\ \lambda_{21}, \lambda_{22}, \cdots, \lambda_{2m} \\ \vdots \\ \lambda_{n1}, \lambda_{n2}, \cdots, \lambda_{nm} \end{bmatrix} \text{ or} \\ &T^T = W^T \cdot \Lambda \end{aligned} \tag{4}$$

Note, w_i and t_i are three-dimensional matrix of the same shape. During the above transformation, we actually perform elementwise linear transformation on the matrix w_i and t_i.

The dimension of T is m, which is configurable and should be equal or bigger than n. A bigger m hides the real dimension of W, but increases the computation time. The obfuscation ratio r is defined as:

$$r = m/n \tag{5}$$

Let Λ denote the transformation matrix. The inverse of the transformation matrix is $B = \Lambda^{-1}$,

which can be derived from Λ.

$$(w_1, w_2, \cdots, w_n) = (t_1, t_2, \cdots, t_m) \begin{bmatrix} \beta_{11}, \beta_{12}, \cdots, \beta_{1n} \\ \beta_{21}, \beta_{22}, \cdots, \beta_{2n} \\ \vdots \\ \beta_{m1}, \beta_{m2}, \cdots, \beta_{mn} \end{bmatrix} \text{ or}$$
$$W^T = T^T \cdot B \tag{6}$$

Combining weights transformation Equation 4 and the property of linear transformation, we have:

$$\begin{aligned} Conv(X, W^T) &= Conv(X, (T^T \cdot B)) \\ &= Conv(X, T^T) \cdot B \end{aligned} \tag{7}$$

So we can compute $Conv(X, T^T)$ on the untrusted GPU, and restore the output Y with the following formula inside the TEE:

$$Y = Conv(X, T^T) \cdot B \tag{8}$$

Pointwise Convolutional Layer: The pointwise convolutional layer is a type of convolution whose kernel height and width are both 1. The scheme for the standard convolutional layer can be directly applied on the pointwise convolutional layer.

Depthwise Convolutional Layer: Depthwise convolution is a type of convolution where we apply a single convolutional filter for each input channel. The number of input channels and the number of filters are the same. For a given image I and filter F, the depthwise convolution $DWConv$ on input channel c can be described as below, where He, Wi represents the height and width of the filter F, and x, y refer to the coordinates in the two-dimensional output feature map. Note that the filter F is a two-dimensional matrix.

$$DWConv(I^{(c)}, F)_{x,y} = \sum_{i=1}^{He} \sum_{j=1}^{Wi} F_{i,j} I^{(c)}_{x+i-1, y+j-1} \tag{9}$$

The original depthwise convolutional layer is described as follows, where x_i represents ith channel of input X.

$$\begin{aligned} Y &= DWConv(X, W) \\ &= [DWConv(x_1, w_1), \cdots, DWConv(x_n, w_n)] \end{aligned} \tag{10}$$

We apply linear transformation on both the input and the filters. In detail, we shuffle the sequence of input channels, and shuffle the sequence of the filters in the same way. We scale each input channel with a random scalar, and scale each filter accordingly. In this way, we can easily restore the result of the transformed depthwise convolutional layer inside the TEE.

Now we discuss how the transformation works on both the input channels and the filters. Assume the input has n channels, so the depthwise convolutional layer has n filters, one per channel. Let w_i represent the ith filter of the weights W, where $W = [w_1, w_2, \ldots, w_n]^T$. We use a group of random scalars $(\lambda_1, \ldots, \lambda_n)$ for 1th to nth input channel respectively.

We pick a random permutation $\pi \in S_n$ (S_n is the group of permutations of $[1,\ldots,n]$). Let P_π be a $n \times n$ permutation matrix corresponding to π ($P_\pi(i,j) = 1$ if $\pi(i) = j$, and 0 otherwise). We scale and shuffle the sequence of the filters in W with Λ, defined as follows.

$$\Lambda = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix} P_\pi \tag{11}$$

We apply the same shuffling sequence for the input channels. Assume the input transformation matrix is A. We define A as:

$$A = \begin{bmatrix} \frac{1}{\lambda_1} & & \\ & \ddots & \\ & & \frac{1}{\lambda_n} \end{bmatrix} P_\pi \tag{12}$$

A and Λ can be described as follows, where I is the identity matrix.

$$A \cdot \Lambda^T = I \tag{13}$$

The transformation on the input X and weights W are described as follows, where $T = [t_1,\ldots,t_n]^T$ is the transformed weights,

$$\begin{aligned} T^T &= W^T \cdot \Lambda \\ X' &= X \cdot A \end{aligned} \tag{14}$$

Let $Y' = DWConv(X', T)$. It is easy to see that:

$$Y' = [DWConv(x_1, w_1), \cdots, DWConv(x_n, w_n)]P_\pi \tag{15}$$

Let P_π^{-1} be the inverse of P_π, we can restore the result with the following equation:

$$Y = Y' \cdot P_\pi^{-1} \tag{16}$$

Notice that, both the transformation of the input and the restoration of the result are performed inside the TEE, while the depthwise convolution on the obfuscated weights can be outsourced to the untrusted GPU.

Dense/Fully Connected Layer: The dense or fully connected layer is a common layer used in neural network. It is also commonly used in CNN based networks. The dense layer connects every input node to every output node. It can be implemented as a pointwise convolutional layer. For example, a dense layer connect n input to m output can be viewed as a pointwise convolutional layer that has m kernels of size (1, 1, n). We can apply the same linear transformation scheme which we apply on the standard convolutional layer.

4.2 Layer Input/Output Privacy

We introduce the mask layer to protect the input and output privacy of the linear layers. Otherwise, by observing enough pairs of input X and output Y, the attacker can infer the weights W of linear layers with Equation 3. We analyze the necessity of mask layer with concrete attack in Section 6.4.

The mask layer: A mask layer is either a random pad added to the input, or a calculated pad to unmask the output. For each linear layer outsourced to the Normal World, we always mask the input before sending it out, and unmask the output inside the TEE. In a typical CNN with multiple linear layers, each layer's output will also be the following layer's input. We call it the layer input/output privacy.

Regenerating the random pad of the mask layer for every model inference is essential for security, but it is avoided for performance reasons. We use pregenerated random pad and one-time use random scalar to reduce the overhead. We leave the security analysis of random scalar for Section 6.4. Let R represent the pregenerated random pad, E the dynamic random pad, and s the one-time use random scalar, we have:

$$E = sR \tag{17}$$

Assume the original input of convolutional layers inside the TEE is X, and masked output is X', then

$$X' = X + E \tag{18}$$

X' will be outsourced to the Normal World for the computation of the following linear layer. For the above two formulas, R, E, X, and X' are multi-dimensional matrices while s is a scalar.

4.3 Scheme optimization

We transform the standard convolutional layer from $Y = Conv(X,W)$ to $Y' = Conv(X,T)$ and restore the result inside the TEE with $Y = Y' \cdot B$ as described in Equation 8. It is a dot product of the convolutional layer's output and the transformation matrix B, and can be treated as a Pointwise Convolutional layer as B can be viewed as of n convolutional kernels of size (1, 1, m). This is helpful for us formulate the security analysis of the scheme in Section 6.4.

$$Y = PWConv(Conv(X,T),B) \tag{19}$$

We use sparse matrices to simplify the generation of T and B. In practice, we first expand $W = (w_1, w_2, \ldots, w_n)$ to $W' = (w_1, w_2, \ldots, w_n, f_1, \ldots, f_{m-n})$ by generating $(m-n)$ random filters $F = (f_1, f_2, \ldots, f_{m-n})$. As T has m filters $(t_1, t_2, \ldots, t_m)$, now W' has the same number of filters as T. Then we can easily transform W' to T with sparse matrices.

We briefly describe the transformation algorithm here. A formal description is attached in Appendix D. It has two steps. Step (1), for each w_i in W, we scale it with a random scalar d, then add it by a filter f, randomly picked from F. The result is stored at a random location j inside T, namely, $t_j = dw_i + f$. Now, n slots of T are filled with transformed weights from W, the other $(m-n)$ slots are still empty. Step (2), we shuffle the $(m-n)$ filters of F into all other empty slots in T. Assume that f is shuffled to the kth filter of T, namely, $t_k = f$. We rewrite t_j as follows. Notice that, w_i, t_k and t_j are filters of the same shape, and d is a random scalar.

$$t_j = dw_i + t_k \tag{20}$$

We restore the original result $Y = Conv(X,W)$ inside the TEE as follows. Assume that $G = Conv(X,T)$. G has m feature maps, let $G = (g_1, g_2, \ldots, g_m)$. Y has n feature maps, let $Y = (y_1, y_2, \ldots, y_n)$. With Equation 20, we have:

$$\begin{aligned} g_j &= Conv(X, t_j) \\ &= Conv(X, (dw_i + t_k)) \\ &= d \cdot Conv(X, w_i) + Conv(X, t_k) \\ &= dy_i + g_k \end{aligned} \tag{21}$$

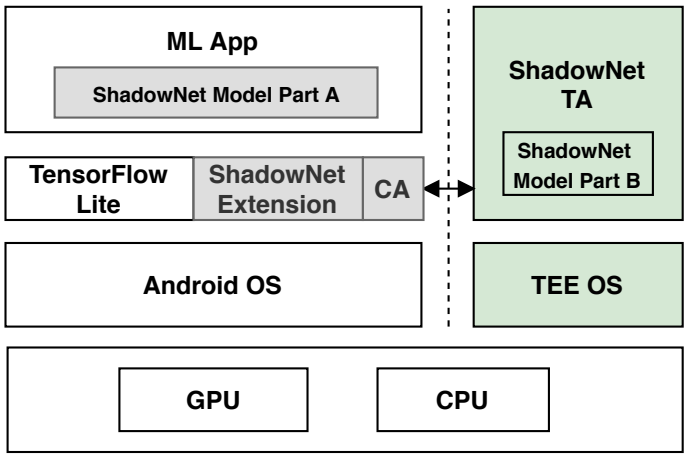


Figure 2: The system architecture of Shadownet on mobile platforms. Color grey shows the part modified by ShadowNet in the Normal World. Color green shows the part in the Secure World. The ShadowNet model part A and B refer to linear and nonlinear part respectively.

Notice that y_i, g_j and g_k are two-dimensional output feature maps, and d is the random scalar. The operations on the feature maps are elementwise operations. We restore y_i as follows.

$$y_i = (g_j - g_k)/d. \tag{22}$$

For each w_i, we record j, k and d. These information can be stored in three arrays of length n. With the recorded information and G, we can easily restore Y inside the TEE.

5 Implementation

In this section, we start with an overview of the ShadowNet prototype, then we discuss three key components of the ShadowNet prototype: ShadowNet model conversion, TensorFlow Lite extension, and ShadowNet CA/TA.

5.1 Overview of the ShadowNet prototype

We have implemented the ShadowNet prototype as an end-to-end on-device model inference system, as shown in Figure 2. It is made up of the ML mobile app, the modified TensorFlow Lite runtime library with ShadowNet Extension, ShadowNet CA and TA. The ShadowNet model has two parts, A and B, referring to the linear and nonlinear part respectively.

During model inference, TensorFlow Lite handles the linear layers in the Normal World. For nonlinear layers, ShadowNet Extension takes the parameters and call the CA to start a secure session with the TA. The TA handles the nonlinear layers inside the Secure World and then pass the results back.

5.2 Model conversion

Given a description for the original TensorFlow model, which can be a Python script with Keras API, we follow a set of rules to transform the description into a full ShadowNet model description. Based on it, we (1) derive the model description with only linear layers and generate ShadowNet Model Part A. (2) extract the weights of the nonlinear layers and generate ShadowNet Model Part B.

New ShadowNet layers: Before we discuss model conversion in detail, we first introduce the new layer types created by ShadowNet. They are LinearTransform, ShuffleChannel, PushMask, PopMask and TeeShadow. LinearTransform applies linear transformation on the input. They will be used in a model conversion example shown in Figure 3. ShuffleChannel shuffles the sequence of the channels and also scales each channel in the input. PushMask and PopMask can both be implemented as AddMask, which adds a random pad on its input. We use the name *PushMask* and *PopMask* to show that the mask and unmask layers are used in pairs. TeeShadow is a special layer used between linear layers in the ShadowNet model(linear part) as a placeholder for those nonlinear layers. When we perform model inference on the linear part of a ShadowNet model, we will switch to the Secure World once we see a TeeShadow layer. We then perform the model inference for the nonlinear layers in the Secure World and return to the TeeShadow layer with the results.

Model conversion rules: Model conversion has two steps. First, we transform a given model description into a full ShadowNet model description; Second, we derive the linear part of the ShadowNet model from the output of the first step. There are four types of linear layers in CNNs: Convolution, Depthwise Convolution, Pointwise Convolution and Fully Connected/Dense. Among them, Pointwise Convolution and Fully Connected/Dense can be treated as Convolution during conversion(explained in Section 4.1). The rules for the first step of the model conversion are:

- Nonlinear layers remain unchanged;
- For any Convolution layer (Pointwise Convolution and Fully Connected/Dense layer included), replace it with four layers: PushMask layer, obfuscated Convolution layer, LinearTransform layer and PopMask layer.
- For any Depthwise Convolution layer, replace it with five layers: PushMask layer, ShuffleChannel layer, obfuscated Depthwise Convolution layer, ShuffleChannel layer and PopMask layer.

When we derive the linear part of the ShadowNet model, we treat all the newly added ShadowNet layers, including LinearTransform, ShuffleChannel, PushMask and PopMask, as nonlinear layers. The rule is very simple: Linear layers remain unchanged; we replace the continuous nonlinear layers with a TeeShadow layer. We use a small part of MobileNets model to show how model conversion works, as shown in Figure 3.

5.3 Adding ShadowNet support in TensorFlow (Lite)

TensorFlow is a Machine Learning framework where developers can define, train and run a model. TensorFlow Lite is based on TensorFlow, and only contains the model inference framework for mobile and embedded devices. It does not support defining or training a new model. To use TensorFlow Lite, you need to convert an existing model to TensorFlow Lite format. ShadowNet needs TensorFlow support to define the ShadowNet model with new layer types. ShadowNet also needs TensorFlow Lite support to run on mobile devices.

ShadowNet introduces several *operations* that are not supported in TensorFlow (Lite) framework. They are ***LinearTransform***, ***ShuffleChannel***, ***AddMask***, and ***TeeShadow***. Both PushMask and PopMask are implemented as AddMask. To add ShadowNet support, we extend TensorFlow (Lite) as follows:

- Add ShadowNet related operations as CustomOps for TensorFlow;
- Add Keras ***Layer*** support for the above operations to provide Keras API;
- Add TensorFlow Lite implementation of the CustomOps to support model inference on mobile devices;

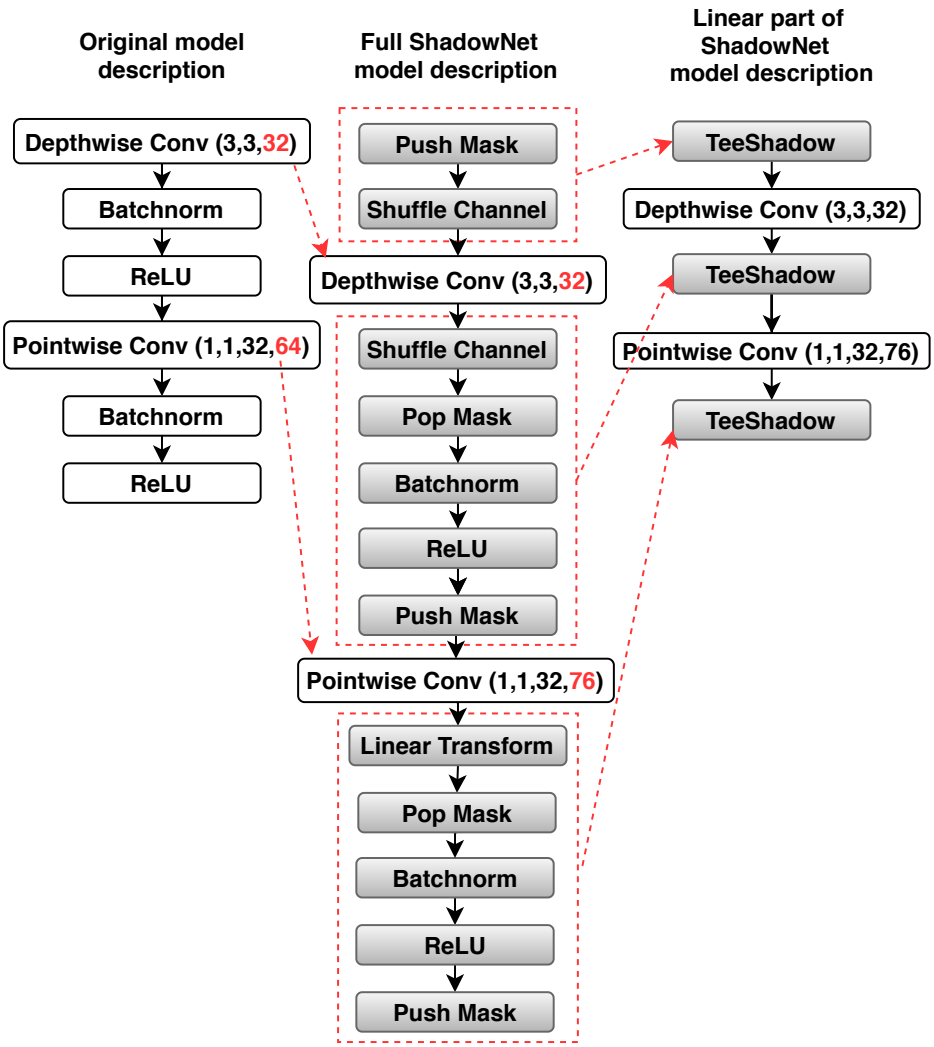


Figure 3: An example of ShadowNet model conversion (obfuscation ratio is 1.2). During the first step, the linear layers are transformed. The Pointwise Convolution layer's weights shape changed from (1, 1, 32, 64) to (1, 1, 32, 76), where $76 = 64 \times 1.2$. During the second step, linear layers remain unchanged, and continuous nonlinear layers are replaced with a placeholder TeeShadow.

- Modify the TensorFlow Lite model converter, which converts TensorFlow model to TensorFlow Lite model, to support the convert of ShadowNet models.

Adding ShadowNet support in TensorFlow (Lite) is non-trivial. There are several challenges. First, ***LinearTransform*** and ***ShuffleChannel*** involve computation on sparse matrix used by us, for which no existing compute library can be reused in TensorFlow (Lite). Second, ***TeeShadow*** operation has different output shapes even on the same input shape, so the output shape inference required by TensorFlow (Lite) for resource allocation needs to be handled correctly. Third, the implementation of TeeShadow for TensorFlow Lite involves interaction with the Trusted Application(TA) in the Secure World, requires integrating TEE client library with the TensorFlow Lite library. Fourth, converting a ShadowNet model from TensorFlow format to TensorFlow Lite format is not supported with the latest TensorFlow Lite model converter. We need to add support for the ShadowNet models.

The extension is based on TensorFlow 2.2, which is the latest version during our implementation. To support CustomOp for TensorFlow, we add 563 LOC in Python, 924 LOC in C++; to support TensorFlow Lite, we add 774 LOC in C++.

5.4 ShadowNet CA and TA

ShadowNet CA and TA work in pair to complete the model inference of the nonlinear layers. During initialization, ShadowNet CA starts a secure session with the TA, and loads the ShadowNet Model Part B into the TA. During the model inference, ShadowNet CA passes the parameters from the TeeShadow operation to the TA and fetch results from it. ShadowNet TA runs the model inference for the nonlinear layers shadowed by TeeShadow operation as shown in Figure 3.

There are several challenges in implementing ShadowNet CA and TA. We will share some of them as well as our experience in building efficient ShadowNet CA and TA below.

Optimizing TA memory management: The TEE OS only has 14MB memory available for TA in our prototype system. Without careful memory management, the TA would exhaust the memory and crash. We share two examples on how we optimize the memory usage in the TA: a) We use static memory allocation in the TA to avoid fragmentation. Dynamically allocating memory with *malloc* will cause fragmentation of the TA memory. For a given model, the memory needed for each layer is predictable and can be allocated statically. b) During model inference inside TA, allocating buffer for each layer's output is also unnecessary. The previous layer's output will only be needed in the current layer, but not the next layer. We pre-allocate two big buffer and rotate them as output buffer to save memory.

Optimizing TA's performance: Writing TA for OP-TEE has many restrictions. For example, OP-TEE only supports C, we lose access to the popular compute libraries like Eigen [7] and Arm Compute Lib [2] in C++. What's more, OP-TEE lacks math library. We need to find efficient implementation of functions such as *sqrt*, *exp*, or *tanh* for the activation layers.

Table 1: Optimizations of the MobileNets TA

Optimizations	Exec. Time (ms)
Baseline (Static mem. alloc)	1500
(1) Neon sqrt	300
(2) Cache friendly	245
(3) Optimize loop sequence	205
(4) Preload weights	100
(5) Neon for Batchnorm, AddMask	90
(6) Neon for ReLU6	81

Note: The optimizations are applied in sequence. For example, the 81 ms is the execution time when the optimization (1) to (6) are all applied.

Initially, we port the nonlinear layers from the Darknet [33], a deep learning framework written in C for desktop. The TA is very slow on our Arm64 Dev Board. We start a series of optimizations to make it as fast as TensorFlow Lite. Table 1 shows our optimizations on the execution time of the MobileNets TA. These optimizations include (1) using Arm Neon to optimize the *sqrt* implementation in the Batchnorm layer; (2) swapping the inner and outer loops to make data access cache-friendly; (3) moving repetitive computation out of the loops and precompute it; (4) preloading all weights to avoid repetitive weights loading (assuming that we have enough TA memory); (5) using Neon multiply+add instructions to optimize the Batchnorm and AddMask layers; and (6) using Neon minimum/maxmum instructions to optimize the activation layers like the ReLU6 layer. We attach the details for the *sqrt* optimization in Appendix A, which is an example of how we profile and optimize the TAs.

6 Evaluation and analysis

We evaluate ShadowNet on three popular machine learning models: MobileNets, AlexNet, and MiniVGG. We perform the evaluation on the Hikey960 board equipped with the Kirin 960 SoC, which features 4 Cortex A73 + 4 Cortex A53 Big.Little CPU architecture, ARM Mali G71 MP8, and 3GB LPDDR4 SDRAM. We run Android P in the Normal World and OP-TEE OS 3.4.2 in the Secure World. The system reserves 16 MB RAM for TEE OS, of which 14 MB can be used by TA.

Our evaluation and analysis focus on four questions:

- Correctness: Does the ShadowNet transformed model produce the same result as the original one?
- Efficiency: How much overhead is introduced by the ShadowNet scheme?
- Ease of Use: How easy is it to apply ShadowNet scheme on a new model?
- Security: How secure are ShadowNet models against thefts?

6.1 Correctness

Theoretically, the ShadowNet transformed model should produce the same result as the original model. Due to the finite precision on computers, numerical errors exist in the transformed model. We measure the errors introduced by the ShadowNet transformation including the linear transformation and the mask layers, and evaluate whether it affects the correctness of the model inference results(or the CNN classification results).

We attach a detailed evaluation of correctness for ShadowNet in Appendix B. Our evaluation shows that the numerical errors introduced by linear transformation is negligible. However, the errors introduced by the mask layers can be intolerably high when the mask value are not of the same order of magnitude as the input. We introduce *Adaptive Mask Layer*, which scales the mask layer dynamically according to the input. *Adaptive Mask Layer* is compatible with the ShadowNet design, only affecting the generation of the random scalars. With the *Adaptive Mask Layer*, the numerical errors become negligible. In summary, the ShadowNet transformed models produce the same result as the original models.

6.2 Efficiency

We evaluate the efficiency of ShadowNet on MobileNets, AlexNet, and MiniVGG. We first evaluate the ShadowNet model inference time, which means the time between feeding an image to a network and getting the classification result. Then, we measure the performance impact of the TEE communication and GPU acceleration. We also evaluate an interesting variant of ShadowNet scheme, the Layerwise ShadowNet, for which we apply ShadowNet scheme on a few selected linear layers.

Experimental highlights: Our evaluation shows that (1) ShadowNet has reasonable overhead, increasing model inference time by 50 ms (22%) for AlexNet, 12 ms (44%) for MiniVGG and 98 ms (63%) for MobileNets in CPU mode. The user experience of real-time object detection is not affected in a noticeable way. (2) The TEE communication has minor impact (adding 3 ms to 11 ms) on the overall model inference time. (3) ShadowNet is so far not GPU friendly as it forces the model inference to be split between CPU and GPU, and the mobile GPU acceleration has large space for optimization. (4) Layerwise ShadowNet is a promising variant of ShadowNet scheme that is GPU friendly, as it reduces model inference time by as much as 59ms (26%) for AlexNet model in GPU mode.

Methodology: We use the TF Lite Android Image Classifier Demo App [10] developed by Google to evaluate the end-to-end model inference time. We evaluate ShadowNet under different settings: (1) the original model(used as the baseline); (2) the ShadowNet transformed model without TEE; (3) the ShadowNet transformed model with TEE; and (4) the Layerwise ShadowNet with half of the layers unmodified and the other half layers transformed by ShadowNet. ShadowNet without TEE means we run everything in the Normal World and do not involve TEE communication at all. For the above settings, we measure the model inference time on the same input in both CPU mode and GPU mode. The obfuscation ratio is set at 1.2, we will discuss the choice in Section 7.2. Table 2 shows the model inference time under different settings.

Table 2: ShadowNet model inference time(ms) under different settings.

	Original Model		ShadowNet without TEE		ShadowNet With TEE		Layerwise (HH)* Without TEE	
	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
MiniVGG	27	24	36	39	39	44	29	29
AlexNet	224	107	265	259	274	282	229	164
MobileNets*	155	85	242	251	253	287	188	159

Note: a. For ShadowNet MobileNets in TEE mode, the nonlinear part of weights size is larger than the TEE memory size. We have to swapping weights in and out of TEE during model inference, the actual model inference time is 434 (ms) and 446 (ms) for CPU and GPU mode respectively. The model inference time *253* and *287* in the table are based on the assumption that the TEE memory is big enough to so that we do not reload weights during model inference. For AlexNet and MiniVGG, their nonlinear weights can be preloaded into TEE. b. "Layerwise (HH)" means the Layerwise ShadowNet scheme of which the first half layers remain unchanged, and we only apply ShadowNet transformation on the other half layers of the model.

Performance without TEE: Comparing with the original model, ShadowNet without TEE in CPU mode increases the model inference time by 9ms(33%),41ms(18%) and 87ms(56%) for MiniVGG, AlexNet and MobileNets, respectively.

The impact on three models are disproportionate: less impact on AlexNet and more impact on MobileNets. The reason is that ShadowNet has fixed impact on linear layers, but different impact on nonlinear layers. For example, ShadowNet always increases computation for linear layers by 20% when the obfuscation ratio is 1.2. And the impact on nonlinear layers depends on the size of the mask layers. AlexNet has relatively smaller mask layers and MobileNets has bigger mask layers.

Performance with TEE: ShadowNet with TEE is the secure setting that protects the models with TEE. Comparing with the original model in CPU mode, ShadowNet with TEE incurs 12ms (44%), 50ms (22%) and 98ms (63%) overhead for MiniVGG, AlexNet and MobileNets, respectively. Comparing with crypto-based approaches which are orders of magnitude slower [21], the performance of ShadowNet is quite good. As for the app user, adding 12ms to 98ms to model inference time causes no significant latency when we run the Demo app to detect objects in real-time mode. For example, the app processes six images per second with the original MobileNets model and 4 images per second with ShadowNet transformed MobileNets model.

Comparing with ShadowNet without TEE in CPU mode, ShadowNet with TEE only incurs 3ms (11%), 9ms (4%) and 11ms (7%) extra overhead for MiniVGG, AlexNet and MobileNets, respectively. This shows that the extra overhead introduced by TEE is negligible comparing with the ShadowNet transformation. Based on this observation, we measure the Layerwise ShadowNet scheme without involving TEE, avoiding the engineering effort for implementing the customized

CA/TA pairs.

Performance impact on GPU: As for the GPU acceleration, we find that ShadowNet scheme almost nullifies the benefits of GPU acceleration. Without ShadowNet, GPU achieves around 1x speedup for AlexNet and MobileNets. After introducing ShadowNet, GPU mode even slows down the model inference comparing with the CPU mode. It adds 5ms, 6ms and 34ms for MiniVGG, AlexNet and MobileNets, respectively.

This happens for several reasons. First, ShadowNet requires splitting the model inference between CPU (TEE mode) and GPU. The interleaving between CPU and GPU will cause extra overhead for repeatedly setting up the GPU jobs. The GPU/CPU switching overhead has large space for optimization. On the other hand, the GPU speedup on our evaluation board is not significant (around 1x), far less than the GPU speedup in the cloud. Mobile GPU acceleration for the on-device machine learning is still an active developing area [6]. ShadowNet will be greatly benefited from the future GPU optimization.

Performance of the Layerwise ShadowNet: We introduce Layerwise ShadowNet as an trade-off option for better performance. The Half-Half mode leaves the first half of the linear layers unprotected and apply ShadowNet scheme on the bottom half of the linear layers. The results are shown in Table 2. As we can see, this Half-Half Layerwise ShadowNet increases 2ms (7%), 6ms (3%) and 33ms (21%) on the model inference time.

For the GPU mode, the layerwise scheme has even better performance. Comparing with the original model running in CPU mode, GPU mode reduces the model inference time by 59ms (26%) for AlexNet, and only increases 2ms (7%) and 4ms (3%) for MiniVGG and MobileNets. We get better GPU performance as those unmodified layers can run on GPU without interruption. This shows if we can selectively apply ShadowNet scheme on a few layers that have more privacy value, ShadowNet scheme will have much better performance without sacrificing the security. For example, research on transferable learning [42] shows that the bottom layers contain more specific features to the training data set, which have more privacy value than those generic features in the top layers.

6.3 Ease of use

In this section, we measure the efforts for developers to apply ShadowNet scheme on new models. In our prototype, we have not automated the CA/TA pair generation. We use code size to measure the development efforts. We also measure the model size change as it affects the deployment of the model.

Experimental highlights: Our evaluation shows that (1) the effort of generating the ShadowNet CA/TA pair is manageable with the help of a template; the CA size ranges from 600 to 900 LOC while the TA size ranges from 1200 to 2000 LOC. (2) ShadowNet transformation usually increases the model size by 20% to 30%. One exception is the MobileNets, of which the model size increases by 40MB(235%) due to the large size of random pad in the mask layers.

Deriving the ShadowNet CA/TA pair: Our customized CA/TA pair is small and only contains the code needed for the given network. The CA/TA pair is written in C. Table 3 shows the code size for each network. CA's code size ranges from 600 to 900 LOC, and TA's ranges from 1200 to 2000 LOC. Different CA/TA pairs share many code in common. A new CA/TA pair is usually generated from a template with some changes on the network structure related code. We can automate the generation of CA/TA pair based on the network structure descriptions in the future.

ShadowNet's impact on model size: The ShadowNet scheme increases the model size. We measure the original and converted model sizes at an obfuscation ratio of 1.2. Table 4 shows the results. ShadowNet has fixed impact on the weights size of the linear layers. For example, it increases the weights size of the linear layers by 20% given the obfuscation ratio of 1.2. The extra weights

Table 3: The code size of CA/TA in ShadowNet.

Code Size (LOC)	AlexNet	MiniVGG	MobileNets
CA	612	635	840
TA	1281	1357	2032

introduced by the mask layers are determined by the output size of each linear layer. As we can see, for AlexNet and MinivGG, ShadowNet increases the model sizes by 20% to 30%. For MobileNets, ShadowNet increases the model size by 235%, as MobileNets has larger mask layers.

Table 4: ShadowNet model size change for different networks.

Model Size(MB)		AlexNet	MiniVGG	MobileNets
Original Model	*Linear*	242	20	17
	Nonlinear	0	0.015	0.175
ShadowNet Model	*Linear*	291	24	20
	Nonlinear	5	2	37

6.4 Security analysis

In this section, we divide the security analysis of the ShadowNet scheme into three sub-questions. We first analyze the necessity of the mask layer. Then we analyze the security of the mask layer. Finally, we analyze the security of the ShadowNet transformation.

The necessity of the mask layer: We consider the security of ShadowNet without the mask layer. For a single Convolution layer, we use $Y = Conv(X,W)$ to represent it, where W is the weights that the attacker wants to learn. Even though ShadowNet transforms W, as long as the input X and output Y can be observed, it is easy to build linear equations with W as unknown parameters and solve the W.

Although ShadowNet keeps the batch normalization and activation layers inside TEE, the attacker can still learn information about the weights W by solving linear equations. Activation layers like ReLU can be easily reversed so they can be ignored. Batch normalization layers are linear layers with unknown parameters. With the observed input and output, the attacker can learn the relationship between the parameters of batch normalization layers and W, thus greatly reducing the unknown parameters of the network. We attach an attack example in Appendix C.

The security of the mask layer: Now we discuss the security of ShadowNet with the mask layer. In our design, we use a pregenerated random pad and a one-time use random scalar s to form the mask layer's random pad. The attacker's goal can be formalized as Figure 4. For convolutional layer $Conv_i$, assume the input X_i has $|X_i|$ elements, masked output Y has $|Y_i|$ elements, the weights W_i has $|W_i|$ elements. Assume that the attacker wants to build linear equations to solve $Conv_i$'s weights W. The attacker can set the mask and unmask on the output as $|Y_i|$ unknown variables, and weights W_i as $|W_i|$ unknown variables.

By watching one round of model inference, the attacker can build $|Y_i|$ linear equations, and there are $2|Y_i| + |W_i| + 2$ unknown variables, namely the mask/unmask parameters, the weights parameters and the two random scalars for the mask and unmask layers. By watching k rounds of model inference, the attacker can watch k input/output pairs and build $k|Y_i|$ linear equations. As we only regenerate the random scalar to scale the same mask for each model inference, there will only be $2|Y_i| + |W_i| + 2k$ unknown variables. When $k|Y_i| > |W_i| + 2|Y_i| + 2k$, the attacker will be able to solve the equations and get weights W.

Attacker's view of ShadowNet model inference

Input: x

View:

$Conv'_i$, $1 \leqslant i \leqslant L$
$I_{i-1} - u_{i-1} + m_{i-1}$

$I_0 = x, m_0 = input\_mask$,
$u_0 = 0, u_1 = input\_unmask$,
$y = F(x)$, *output of the model*.

Goal of Attacker:

Find weights of $Conv_i$, $1 \leqslant i \leqslant L$.

Figure 4: **Formalization of the attacker's goal** $y = F(x)$ refers to the CNN with L convolutional layers. Given input x, the model's output is y. I_{i-1} refers to the input from previous layer $i-1$, m_{i-1} and u_{i-1} refers to the mask/unmask at layer $i-1$. For each $Conv'_i$ layer, the attacker can observe the masked input $I_{i-1} - u_{i-1} + m_{i-1}$, and the weights of $Conv'$. Attacker's goal is to find the original weights of $Conv_i$.

For the convolutional layer in Figure 5, $|W_i|$ is $3 \times 3 \times 3 \times 64$, namely 1,728; $|Y_i|$ is $222 \times 222 \times 64$, namely 3,154,176. So the attacker needs to watch at least three model inference to solve the equations. The cost is to solve $|W_i| + 2|Y_i| + 6$, namely 6,310,086 linear equations. Solving millions of linear equations with float parameters while meeting the model weights precision is a hard problem in practice. It shows that the random pads in the mask layer needs to be updated periodically to prevent such brute force attacks.

The security of the ShadowNet transformation: Now we consider another type of attack which is to train an equivalent CNN with the transformed weights. There are already lots of research on stealing model parameters via prediction APIs [24, 32, 38]. In those attacks, the attacker trains an equivalent model with no access to the model except the querying APIs(a black-box model). For ShadowNet models, the attacker has extra access to the transformed weights. Our question is, will the transformed weights help the attacker gain an advantage in training an equivalent model?

To answer this question, we assume that the attacker reuses the transformed weights to build an equivalent CNN. Let us use the example CNN from Figure 1, the minimum equivalent CNN reusing the transformed weights is shown in Figure 5. We use *pwconv* to represent the *linear transformation* layer inside the TEE. As mentioned before in Section 4.3 the linear transformation layer is essentially a pointwise convolutional layer. Besides, the mask layers are not needed to construct the equivalent CNN.

We simplify the analysis by assuming that the difficulty to train a CNN is in proportion to the number of learnable parameters. Thus we can compare the number of learnable parameters to tell whether it is easier to train an equivalent CNN or not. In our example, the block in the original CNN has $3*3*3*64+4*64$, namely 1,984 parameters, while the block in the attacker's equivalent CNN has $76*64+4*64$, namely 5,120 parameters to be trained. $4*64$ is the number of learnable parameters in the Batchnorm (*bn*) layer. 76 is the number of filters in the *conv'* layer. It is configurable with respect to the obfuscation ratio, which is set to 1.2 in our example($76 = 64*1.2$). What's more, even if we set the obfuscation ratio to 1, the minimum allowed, there is still more learnable parameters in the attacker's CNN ($64*64+4*64 = 4,480$). We can tell that the attacker gains no advantage by training an equivalent CNN with the transformed weights.

The analysis here is not a strict proof for all ShadowNet transformations. It provides a way to

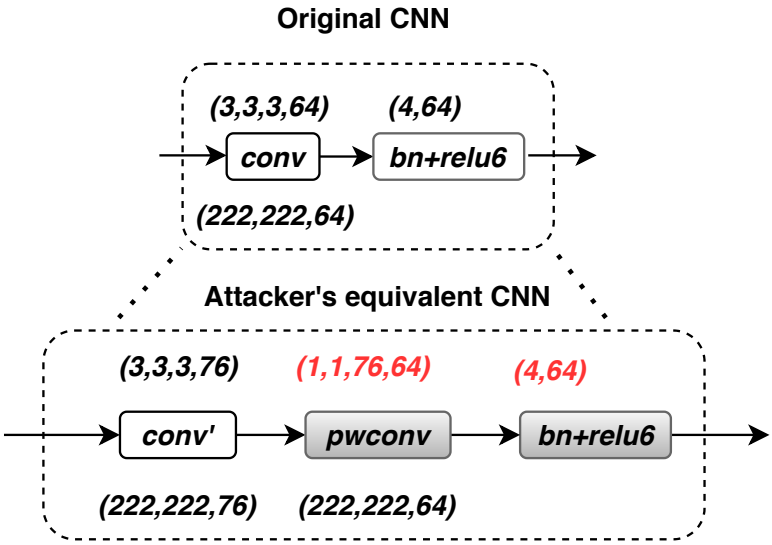


Figure 5: The equivalent CNN architecture needs to be trained by attacker. The red color marks the weights shape of the layers with unknown parameters that the attacker has to train. The weights shape is marked on top of the box and the output shape is marked under the box.

analyze the attacker's effort in training an equivalent CNN with the transformed weights.

7 Discussion

7.1 Scalability of ShadowNet

Support for CNNs and RNNs: ShadowNet can be applied on various types of CNNs. We have applied ShadowNet on AlexNet, MiniVGG, and MobileNets. Following the same transformation rules, we can apply ShadowNet on Inception, DenseNet, ResNet and so on. The CA/TA might be a little different. For example, for networks with shortcuts, like DenseNet and ResNet, the TA will need more TEE memory to keep the previous layers' output as input for the following layers.

ShadowNet can also be used on RNNs, like LSTMs. For LSTMs, we can apply linear transformation for the fully connected layers while keeping the pointwise operations and activation layers inside the TEE. ShadowNet only support linear layers with fixed weights. If the weights are changing all the time, then the transformation can not be done offline. For this reason, we need to move the pointwise operations on cell states into TEE as it changes over time.

Support for cloud platform: ShadowNet can be applied for secure model inference in the cloud just like for on-device model inference. The ShadowNet CA/TA needs to be changed to support cloud TEE like SGX. Comparing with other design that puts model inference inside SGX, ShadowNet will make more efficient use of SGX memory and will be benefited from the co-located GPUs for acceleration.

Quantized models: So far, ShadowNet does not support quantized models. Quantized models usually use 8-bit integer for weight parameters. The computation on the ShadowNet transformed weights overflows easily. The correctness of the transformed model can not be guaranteed.

Model training process: ShadowNet does not support training process, as the ShadowNet transformation assumes fixed weight value, while the weights during training change frequently.

7.2 The choice of the obfuscation ratio

The choice of obfuscation ratio has predictable impact on the performance and the weights size of the outsourced linear layers as explained in Section 6. We have also shown in our security analysis that the network is still secure even with the minimal obfuscation ratio, namely 1.0.

It is possible to better protect the models with flexible use of the obfuscation ratio. For example, the obfuscation ratio for each outsourced linear layer can also be different. If we add some randomness to the obfuscation ratio for each outsourced linear layer and pad the input accordingly, we can hide the weights shape of the original linear layers, further protecting the hyperparameters of the original model. In our evaluation, we choose 1.2 as the obfuscation ratio universally to make it easier to understand how the scheme works. It also indicates we can get better performance results if we choose a lower obfuscation ratio.

7.3 Extend TEE memory

For Hikey960 board with OP-TEE, 16 MB RAM is reserved for TEE OS, another 16 MB is reserved for the SDP(Secure Data Path), the actual memory available to TA is only 14MB, which is too restrictive. We suggest allocating 64MB(currently 32MB) for TEE, so that there will be 48MB memory available for TA which will be large enough for MobileNets. TEE memory size is a hardware configuration supported by Arm TrustZone. The reconfiguration is possible with changes to the low-level system including the Linux Kernel, Arm Trusted Firmware and OP-TEE OS. We will leave it as an option for the future.

8 Related work

Existing research on secure machine learning covers both the end devices and the cloud. Offline Model Guard (OMG) [16] provides a secure model inference framework for mobile device based on SANCTUARY [17], a user space enclave based on Arm TrustZone. OMG allows the model inference framework runs fully inside SANCTUARY enclave to protect model privacy. MLCapsule [23] also deploy the model on the client side to protect the user input from being sent to the untrusted cloud end. At the same time, it runs the model inference inside SGX to prevent the model from being leaked to the client. GPU is not guarded by SANCTUARY and SGX, so both OMG and MLCapsule do not support secure GPU acceleration. DarkneTZ [31] is a secure machine learning framework built on top of Arm TrustZone. It allows a few selected layers to be running inside TEE to protect part of the model. By running heavy linear layers inside TEE, it also poses resource challenges(like memory)on TEE. Comparing with OMG, MLCapsule and DarknetTZ, ShadowNet has a small TCB inside TEE and allows secure outsource of linear layers onto GPU. Graviton [40] proposes TEE extension for GPU hardware, thus allowing GPU tasks like machine learning to be running securely on GPU. It is a promising feature but requires hardware changes on GPU. Secloak [29] partitions GPU into secure world with Arm TrustZone to run GPU tasks securely at high performance penalty.

Research on securing machine learning on the cloud end is an active area. TensorSCONE [27] propose a secure machine learning framework running in the untrusted cloud. TensorSCONE integrates TensorFlow with the secure Linux container technology SCONE [15] guarded by SGX. TF Trusted [11] leverages custom operations to send gRPC messages into the Intel SGX device via Google Asylo [4] where the model is then run by Tensorflow Lite. Running model inference inside TEE faces performance challenges due to limited memory and lack of GPU acceleration. Occlumency [28] provides a suite of heuristic techniques based on Caffe and improves inference speed by 3.6 times. Despite promising, these works do not support GPU acceleration.

YerbaBuena [22] partitions the model into frontnets(like the first layer) and backnets,and execute the frontnets inside SGX to protect the user input from being leaked to the untrusted cloud while running backnets unprotected to leverage hardware acceleration. Slalom [37] splits DNN into linear and nonlinear layers, and outsources linear layers to GPU for acceleration with masked input. It verifies the linear layer's results and computes the nonlinear layers inside SGX. Slalom protects the user input privacy but not the model weights. SecureNets [19] transforms both input and linear layer's weights into matrix, and applies matrix transformation proposed in [35] to hide the non-zero elements, then sends them to the untrusted cloud for acceleration. It is not clear whether SecureNets supports depthwise convolution and convolution with stride. ShadowNet does not require transforming input and weights into matrix, and is compatible with existing linear operations.

There are also secure ML with cryptographic approach. CryptoNets [21] applies Homomorphic Encryption(HE) on neural networks and runs model inference on encrypted data to protect user input privacy. Jiang et. al [25] presents a solution to encrypt a matrix homomorphically and perform arithmetic operations on encrypted matrices. It protects both user data and model. TF Encrypted [9] enables training and prediction over encrypted data via secure multi-party computation and homomorphic encryption. SafetyNets [20] designs a Interactive Protocol(IP) that allows clients to verify the correctness of a class of DNNs running on the untrusted cloud by asking for a short mathematical proof.

To ensure the integrity of model weights, Uchida et. al [39] and Zhang et. al [43] embed watermarks into deep neural model parameters, while training the models. DeepAttest [18] encodes fingerprint in DNN weights to prevent weight modification. These works are incapable of preventing weight leakage.

9 Conclusion

In this paper we address a challenging problem for on-device machine learning. We propose ShadowNet, a secure on-device model inference system that can protect the model privacy with TEE without losing access to the untrusted hardware acceleration. We implement the ShadowNet prototype based on TensorFlow Lite and OP-TEE. Our evaluation on three popular CNNs including AlexNet, MiniVGG and MobileNets shows that the ShadowNet models have comparable performance with the original models, offering a promising solution for secure on-device model inference.

References

[1] HiKey 960 . https://www.96boards.org/product/hikey960/.

[2] Arm Compute Library. https://www.arm.com/why-arm/technologies/compute-library.

[3] Arm TrustZone. https://developer.arm.com/ip-products/security-ip/trustzone.

[4] Asylo, An open and flexible framework for enclave applications. https://asylo.dev/.

[5] Coral: An ecosystem for local AI. https://coral.ai/about-coral/.

[6] Coral: An ecosystem for local AI. https://blog.tensorflow.org/2019/01/tensorflow-lite-now-faster-with-mobile.html.

[7] Eigen. http://eigen.tuxfamily.org/index.php?title=Main_Page.

[8] OP-TEE AOSP support. https://optee.readthedocs.io/en/latest/building/aosp/aosp.html.

[9] TF Encrypted. https://github.com/tf-encrypted/tf-encrypted.

[10] TF Lite Android Image Classifier App Example. https://github.com/tensorflow/tensorflow/tree/r2.2/tensorflow/lite/java/demo.

[11] TF Trusted. https://github.com/dropoutlabs/tf-trusted.

[12] TrustZone for Cortex-A. https://www.arm.com/why-arm/technologies/trustzone-for-cortex-a.

[13] TrustZone for Cortex-M. https://www.arm.com/why-arm/technologies/trustzone-for-cortex-m.

[14] Wiki: Trusted Execution Environment. https://en.wikipedia.org/wiki/Trusted_execution_environment.

[15] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. {SCONE}: Secure linux containers with intel {SGX}. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 689–703, 2016.

[16] Sebastian P Bayerl, Tommaso Frassetto, Patrick Jauernig, Korbinian Riedhammer, Ahmad-Reza Sadeghi, Thomas Schneider, Emmanuel Stapf, and Christian Weinert. Offline model guard: Secure and private ml on mobile devices. *DATE 2020*, 2020.

[17] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Sanctuary: Arming trustzone with user-space enclaves. In *NDSS*, 2019.

[18] Huili Chen, Cheng Fu, Bita Darvish Rouhani, Jishen Zhao, and Farinaz Koushanfar. Deepattest: an end-to-end attestation framework for deep neural networks. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 487–498, 2019.

[19] Xuhui Chen, Jinlong Ji, Lixing Yu, Changqing Luo, and Pan Li. Securenets: Secure inference of deep neural networks on an untrusted cloud. In *Asian Conference on Machine Learning*, pages 646–661, 2018.

[20] Zahra Ghodsi, Tianyu Gu, and Siddharth Garg. Safetynets: Verifiable execution of deep neural networks on an untrusted cloud. In *Advances in Neural Information Processing Systems*, pages 4672–4681, 2017.

[21] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210, 2016.

[22] Zhongshu Gu, Heqing Huang, Jialong Zhang, Dong Su, Ankita Lamba, Dimitrios Pendarakis, and Ian Molloy. Yerbabuena: Securing deep learning inference data via enclave-based ternary model partitioning. *arXiv preprint arXiv:1807.00969*, 2018.

[23] Lucjan Hanzlik, Yang Zhang, Kathrin Grosse, Ahmed Salem, Max Augustin, Michael Backes, and Mario Fritz. Mlcapsule: Guarded offline deployment of machine learning as a service. *arXiv preprint arXiv:1808.00590*, 2018.

[24] Matthew Jagielski, Nicholas Carlini, David Berthelot, Alex Kurakin, and Nicolas Papernot. High-fidelity extraction of neural network models. *arXiv preprint arXiv:1909.01838*, 2019.

[25] Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. Secure outsourced matrix computation and application to neural networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1209–1222, 2018.

[26] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. {GAZELLE}: A low latency framework for secure neural network inference. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1651–1669, 2018.

[27] Roland Kunkel, Do Le Quoc, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. Tensorscone: A secure tensorflow framework using intel sgx. *arXiv preprint arXiv:1902.04413*, 2019.

[28] Taegyeong Lee, Zhiqi Lin, Saumay Pushp, Caihua Li, Yunxin Liu, Youngki Lee, Fengyuan Xu, Chenren Xu, Lintao Zhang, and Junehwa Song. Occlumency: Privacy-preserving remote deep-learning inference using sgx. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–17, 2019.

[29] Matthew Lentz, Rijurekha Sen, Peter Druschel, and Bobby Bhattacharjee. Secloak: Arm trustzone-based mobile peripheral control. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pages 1–13, 2018.

[30] Linaro. Open Portable Trusted Execution Environment. https://www.op-tee.org/.

[31] Fan Mo, Ali Shahin Shamsabadi, Kleomenis Katevas, Soteris Demetriou, Ilias Leontiadis, Andrea Cavallaro, and Hamed Haddadi. Darknetz: Towards model privacy at the edge using trusted execution environments. In *ACM MobiSys 2020*.

[32] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pages 506–519, 2017.

[33] Joseph Redmon. Darknet: Open source neural networks in c. http://pjreddie.com/darknet/, 2013–2016.

[34] M Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 707–721, 2018.

[35] Sergio Salinas, Changqing Luo, Weixian Liao, and Pan Li. Efficient secure outsourcing of large-scale quadratic programs. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 281–292, 2016.

[36] Zhichuang Sun, Ruimin Sun, and Long Lu. Mind your weight (s): A large-scale study on insufficient machine learning model protection in mobile apps. *arXiv preprint arXiv:2002.07687*, 2020.

[37] Florian Tramer and Dan Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. *arXiv preprint arXiv:1806.03287*, 2018.

[38] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 601–618, 2016.

[39] Yusuke Uchida, Yuki Nagai, Shigeyuki Sakazawa, and Shin'ichi Satoh. Embedding watermarks into deep neural networks. In *Proceedings of the 2017 ACM on International Conference on Multimedia Retrieval*, pages 269–277, 2017.

[40] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on gpus. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 681–696, 2018.

[41] Mengwei Xu, Jiawei Liu, Yuanqiang Liu, Felix Xiaozhu Lin, Yunxin Liu, and Xuanzhe Liu. A first look at deep learning apps on smartphones. In *The World Wide Web Conference*, pages 2125–2136, 2019.

[42] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in neural information processing systems*, pages 3320–3328, 2014.

[43] Jialong Zhang, Zhongshu Gu, Jiyong Jang, Hui Wu, Marc Ph Stoecklin, Heqing Huang, and Ian Molloy. Protecting intellectual property of deep neural networks with watermarking. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 159–172, 2018.

Appendix A Optimizing the sqrt function

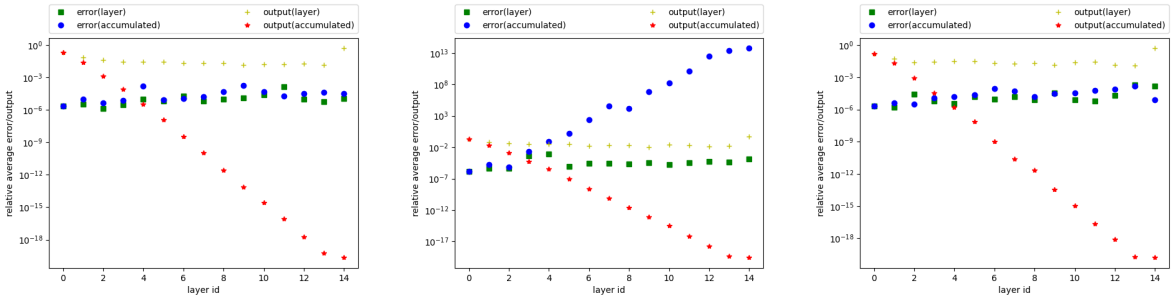
Table 5: Performance of different *sqrt* implementation.

Sqrt Impl.	Time(ms)	S/H	Algorithm	CFLAG
GNU libc	3.53	S	IEEE754	Default
Newlib	13.78	S	IEEE754	*-O2*
Our TA	194.04	S	Newton	*-Os*
Arm VFP	10.86	H	unknown	Default
Arm Neon	6.62	H	Newton	Default

Note: a. S/H: *S* means Software based implementation, *H* means Hardware based implementation, like special instructions; b. CFLAG: GCC compilation flag; c. IEEE754 means algorithm exploits bits hacking of IEEE754 float format; d. Newton means Newton Iteration for sqrt.

There are many different implementations of *sqrt* function for float numbers for AArch64 architecture. Software-based implementations include algorithms using Newton Iteration and bits hacking of IEEE754 float representation. Hardware-based implementations include Arm VFP support for *fsqrt*, and Arm Neon support for float *sqrt*. What's more, the performance of software-based implementations is also affected by the compilation flag. The default gcc compilation flag for TA is *-Os*, which optimizes space first; if we change it to *-O2*, the performance is more than 100x faster while the TA size increases from 55KB to 67KB. We evaluated all the above implementations by doing 3,200,000 sqrt operations, and get the results in Table 5. Our TA initially used a software implementation using Newton Iteration algorithm. After evaluation, we switched to the Arm Neon based *sqrt* implementation for the speed and ease of implementation.

Appendix B The correctness of ShadowNet



(a) Errors introduced by linear transformation.

(b) Errors introduced by mask layers

(c) Errors introduced by adaptive mask layers

Figure 6: Errors introduced by linear transformation and mask layers. *error(layer)* and *error(accumulated)* shows the relative average error introduced by each transformed linear layer and the first k transformed linear layers respectively; *output(layer)* and *output(accumulated)* shows the average output value from each linear layer and the first k linear layers respectively.

Methodology: Given the original model and the transformed model We compare them layer by layer. For each original layer and the transformed layer, we compare the output on the same input. We also treat the first k linear layers as a model, and compare the output of both models on the same input.

We use MobileNets as an example as it covers both pointwise and depthwise convolution layers. For simplicity, we treat a pointwise convolution layer and a following depthwise convolution as one linear layer.

Assume that the input tensor is X, the output tensors for the original model and the transformed model are Y and Y' respectively. We use the relative average error η as the metric to measure the impact, assuming there are N elements in both Y and Y'.

$$\eta = \frac{\sum_{i=1}^{N} |Y[i] - Y'[i]|}{\sum_{i=1}^{N} |Y[i]|} \tag{23}$$

In addition to relative average error, we also measure how much percentage has their final classification results changed due to the transformation.

The impact of linear transformation: Figure 6a shows the average relative error introduced by linear transformation on MobileNets. For each linear layer, the relative average errors are negligible, fall within the range of $(0.0001\%, 0.01\%)$ For the first k layers, the relative average error remains within $(0.0001\%, 0.01\%)$. It shows the error is negligible and has no accumulating effect. The final classification results is not changed at all.

The impact of mask layers: Figure 6b shows the average relative error introduced by the mask layers on MobileNets. As we can see, for each individual layer, the relative average errors are negligible, within $(0.0001\%, 0.01\%)$. However, the accumulated relative average error for the first k layers increases exponentially with k. It can be as high as 10^{13}. The classification results is heavily disrupted and can not be trusted at all.

We find that mask layers incur errors and the errors have accumulating effect through a chain of layers. The reason is that the weights in the mask layers are not of the same order of magnitude as the output. For example, the mask weights (usually bigger than 0.1) are orders of magnitude bigger than the output(can be as small as $1.0e-13$). We show the average output for each layer and for the first k layers in Figure 6b. The big mask dilutes the small output and the relative errors become intolerably high. On the other hand, the mask cannot be too small. The privacy of the output will be leaked when the mask are so small that they can be ignored. With the above observation, we know that the mask should be of the same order of magnitude as the output at each layer.

We introduce the *Adaptive Mask Layer*. The adaptive random scalar s is made up two parts: a fixed random scalar f generated during model conversion, and an adaptive scalar a that is generated at runtime on the layer's input, as shown in Formula 24. The adaptive scalar a can be the average absolute value of the input. We can use sampling to speed up the estimation of a. For example, we can sample 100 non-zero elements from the input, calculate the average absolute value on them.

$$s = fa \tag{24}$$

This *Adaptive Mask Layer* is compatible with our scheme, only affecting the generation of the random scalars used in the mask layers. After applying the *Adaptive Mask Layer*, the evaluation results on the same MobileNets model is shown in Figure 6c. The new accumulated errors are within a reasonable range of $(0.0001\%, 0.01\%)$. The classification results is not changed at all after applying adaptive mask layer on the model.

The impact of ShadowNet transformation: ShadowNet transformation is a combination of linear transformation and *Adaptive Mask Layer*. The evaluation results on ShadowNet is similar to what Figure 6c shows. The accumulated relative average error falls within $(0.0001\%, 0.01\%)$. The final classification results is not changed at all. The evaluation shows that ShadowNet model produces the same classification results as the original model.

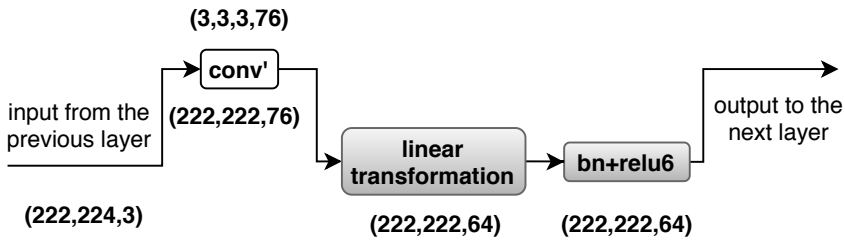


Figure 7: ShadowNet transformation without mask layers. We only show the transformation on the one Convolution layer from the example CNN. The attacker can observe both the input to the *conv'* and the output to the next layer.

Appendix C An example attack on ShadowNet without mask layer

We have introduced the ShadowNet scheme with mask layer in section 4. We analyze the necessity of the mask layer in this section. We will show that without the mask layer, the attacker can infer model weights information with controlled input.

Let us use the example CNN from Figure 1. The input is a three-dimensional RGB image that can be represented as a three-dimensional matrix $X[224][224][3]$. Similarly, we use matrix $W[3][3][3][64]$ to represent the weights of the convolutional layer. We use $\mu_B, \sigma_B^2, \gamma, \beta$ to represent the rolling mean, rolling variance, scale and bias of the Batchnorm layer. They are the trained parameters of the Batchnorm layer, and every output channel has one group of such parameters. To simplify the discussion, we do not consider the impact of the *ReLU6* activation layer.

The attacker starts the attack by feeding the transformed network in Figure 7 with the following input and observes the output.

- set only $X[0][0][0] = 1$, represented as x_0, all other element in X are zeros, assume output $Y[0][0][0] = y_0$ is observed.
- set only $X[0][0][0] = 2$, represented as x_1, all other element in X are zeros, assume output $Y[0][0][0] = y_1$ is observed.

We use w_0 to represent $W[0][0][0][0]$ for simplicity, and use ε to represent the hyper parameter in the Batchnorm layer. By simulating the convolutional layer and the Batchnorm layer's computation, we have the following equations.

$$\begin{cases} \gamma \dfrac{x_0 w_0 - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} + \beta = y_0 & (25) \\ \gamma \dfrac{x_1 w_0 - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} + \beta = y_1 & (26) \\ x_0 = 1 & (27) \\ x_1 = 2 & (28) \end{cases}$$

Let us use p to represent $\frac{\gamma}{\sqrt{\sigma_B^2+\varepsilon}}$. By solving the above equations, we have the following representation of w_0.

$$w_0 = \frac{y_1 - y_0}{p} \tag{29}$$

As we can see, without the mask layer, the attacker can infer the relationship between the weights W and the Batchnorm layer's parameters p.

We assume that the Batchnorm layer is present in the above analysis. When there is no Batchnorm layer, which is possible in some CNNs, the attacker can infer the weights easily with the following formula.

$$w_0 = y_1 - y_0 \tag{30}$$

Appendix D Algorithm for transformation matrix generation

Algorithm 1: Linear Transformation from Weight W to T

Result: T, tensor of shape(H,W,C,M)
Input: W, tensor of shape(H,W,C,N);
obf_ratio, $(obf\_ratio > 1.)$;
Output: T, tensor of shape(H,W,C,M);

$M \leftarrow int(N * obf\_ratio)$;
$R \leftarrow M - N$;
$rand\_weight \leftarrow random\_tensor(shape = (H,W,C,R))$;
$T \leftarrow zero\_tensor(shape = (H,W,C,M))$;
$shuffled\_array \leftarrow shuffle([0,1,2,...,M-1])$;
$obf\_dic \leftarrow \{\ \}$;
// obfuscation dictionary
$i \leftarrow 0$;
while $i < N$ **do**
    $chn\_from \leftarrow shuffled\_array[i]$;
    $chn\_to \leftarrow where(shuffled\_array == i)$;
    $chn\_rand \leftarrow random\_int(0,R)$;
    // scalar of channel i, range(0,1)
    $scalar \leftarrow random\_float(0,1)$;
    $obf\_dic[i] \leftarrow (chn\_from, chn\_to, chn\_rand, scalar)$;
    $i \leftarrow i + 1$;
end
// fill T with shuffled_array and obf_dic
$i \leftarrow 0$;
while $i < M$ **do**
    // get the channel to be shuffled to current index
    $chn\_idx \leftarrow shuffled\_array[i]$;
    $chn\_from \leftarrow obf\_dic[chn\_idx][0]$;
    $chn\_to \leftarrow obf\_dic[chn\_idx][1]$;
    $chn\_rand \leftarrow obf\_dic[chn\_idx][0]$;
    $scalar \leftarrow obf\_dic[chn\_idx][0]$;
    if $chn\_idx < N$ **then**
        // used for normal channel
        for $h \leftarrow 0$ **to** H **by** 1 **do**
            for $w \leftarrow 0$ **to** W **by** 1 **do**
                for $c \leftarrow 0$ **to** C **by** 1 **do**
                $T[h][w][c][i] \leftarrow$
                $(W[h][w][c][chn\_from] - rand\_weight[h][w][c][chn\_rand])/scalar$;
                end
            end
        end
    else
        // used for random channel
        $T[h][w][c][i] \leftarrow rand\_weight[chn\_idx - N]$;
    end
end