TransNet: Shift Invariant Transformer Network for Power Attack

Suvadeep Hajra, Sayandeep Saha, Manaar Alam and Debdeep Mukhopadhyay

Indian Institute of Technology Kharagpur, Kharagpur, India

Abstract. Masking and desynchronization of power traces are two widely used countermeasures against power attacks. Higher-order power attacks are used to break cryptographic implementations protected by masking countermeasures. However, they require to capture long-distance dependency, the dependencies among distant Points-of-Interest (PoIs) along the time axis, which together contribute to the information leakage. Desynchronization of power traces provides resistance against power attacks by randomly shifting the individual traces, thus, making the PoIs misaligned for different traces. Consequently, a successful attack against desynchronized traces requires to be invariant to the random shifts of the power traces. A successful attack against cryptographic implementations protected by both masking and desynchronization countermeasures requires to be both shift-invariant and capable of capturing long-distance dependency. Recently, Transformer Network (TN) has been introduced in natural language processing literature. TN is better than both Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN) at capturing long-distance dependency, and thus, a natural choice against masking countermeasures. Furthermore, a TN can be made shift-invariant making it robust to desynchronization of traces as well. In this work, we introduce a TN-based model, namely TransNet, for power attacks. Our experiments show that the proposed TransNet model successfully attacks implementation protected by both masking and desynchronization even when it is trained on only synchronized traces. Particularly, it can bring down the mean key rank below 1 using only 400 power traces if evaluated on highly desynchronized ASCAD_desync100 dataset even when it is trained on ASCAD dataset which has no trace desynchronization. Moreover, if compared to other state-of-the-art deep learning models, our proposed model performs significantly better when the attack traces are highly desynchronized.

Keywords: side channel analysis, masking countermeasure, transformer network

1 Introduction

Ever since its introduction in [KJJ99], power analysis attacks pose a significant threat to cryptographic implementations. To protect the cryptographic implementations from those attacks, several countermeasures have been proposed. Masking countermeasures [CJRR99, Mes00, CG00, AG01] and desynchronization of traces [CK09, CK10] are two commonly used countermeasures against those attacks. Recently, several deep learning methods [MZ13, MPP16] have been found to be very effective against both the countermeasures. In literature, the attacks against the two countermeasures are related to two widely studied problems in deep learning, namely the problem of capturing *long-distance dependency* [Mag19, ZBHV20, PA20] and *shift-invariant feature learning* [CDP17, ZBHV20, CDP17, BPS+20, ZS20, KPH+19, WHJ+20, ZBHV20, WAGP20].

In masking countermeasure, each intermediate sensitive variable of the cryptographic implementation is divided into multiple shares so that any proper subset of the shares remains independent of the sensitive variable. A successful attack against the masking scheme combines the leakages of all the shares to infer information about the sensitive variable. Such attacks are called higher-order power attacks. In many implementations, the leakage points (also known as Points-of-Interest or PoIs) of different shares can be spread across multiple sample points in the power traces. Moreover, the distances between the PoIs can be large. Thus, a higher-order attack against such implementations requires to combine the leakages of sample points which are long distance apart. In deep learning literature, this problem is referred to as the problem of capturing long-distance dependency [HBFS01, VSP+17].

On the other hand, misalignment or desynchronization of the power traces causes the PoIs of the traces to be misaligned with each other, making the signal-to-noise ratio (SNR) of the individual sample points to reduce. The reduced SNR causes an increase in the number of required power traces for a successful attack. In deep learning, the problem of misalignments in the inputs are addressed by making the deep neural models shift-invariant [Zha19].

Recently, Convolutional Neural Networks (CNNs) have been widely adopted for performing power attacks. Because of the shift-invariance property of CNNs, they can perform very well on misaligned attack traces, and thus, can eliminate critical preprocessing step like realignment of power traces in a standard power attack [CDP17]. Since the robustness of CNN-based models to misaligned traces do not depend on any property of power traces, the CNN-based models might succeed in a situation where the realignment techniques might fail. Moreover, the CNN-based models have achieved the state-of-the-art results in many publicly available datasets [ZBHV20, WAGP20].

Though the CNNs are good at shift-invariant feature extraction, they are not good at capturing long-distance dependency [VSP+17]. In a power attack, the existing CNN-based models achieve the ability to capture long-distance dependency either by interleaving pooling layers with convolutional layers or by using several fully connected layers after the convolutional layers or by a combination of both [BPS+20, KPH+19, ZBHV20]. Recently, [PA20] have proposed to use dilated convolutional layers for capturing long-distance dependency. However, it is known in the deep learning literature that interleaving some sub-sampling layers like max-pooling or average-pooling with convolutional layers, or the usage of dilated convolutional layers reduce the shift-invariance of CNN models [Zha19, YK16]. On the other hand, fully connected layers are not shift-invariant at all unless their input is already shift-invariant. As a result, the existing CNN-based models are limited in several aspects. Firstly, to perform well on desynchronized attack traces, they are required to be trained using profiling desynchronization almost same as attack desynchronization [ZS20, WHJ+20]. Secondly, even when trained using profiling desynchronization same as attack desynchronization, they fail to perform well for a very large value of desynchronization. Finally, the CNN-based models are designed considering either the amount of desynchronization in the attack traces or the input length or both. Thus, if the target traces differ in any one of those parameters, it is required to develop an entirely new model architecture.

Like CNNs, Recurrent Neural Networks (RNNs) are also not good at capturing long-distance dependency [HBFS01, VSP+17]. Though Feed-Forward Networks (FFNs) are good at that, they might introduce a very large number of parameters leading to overfitting problems during training. Besides, FFNs are not shift-invariant. To alleviate this deficiency of the existing deep learning models, Vaswani et al. [VSP+17] have introduced Transformer Network (TN) which has defeated all of its CNN and RNN based counterparts by a wide margin in almost every natural language processing task. TN can easily capture long-distance dependency, and thus, is a natural choice against masking countermeasure.

Moreover, by introducing a weaker notion of shift-invariance, we have shown that TN can be shift-invariant in the context of power attacks. Thus, TN can be effective against misaligned traces as well. In this paper, we propose to use TN for power attacks.

Our Contributions

The contributions of the paper are as follows:

- Firstly, we propose to use TN for power attacks. TN can naturally capture long-distance dependency ([VSP+17]), thus, is a better choice against masking countermeasure. Additionally, we have defined a weaker notion of shift-invariance which is well applicable to power attacks. Under the new notion, we have mathematically shown that the TN can be made to be shift-invariant in the context of power attacks. Thus, it can be effective against misaligned traces as well.

- We have proposed TransNet, a TN-based shift-invariant deep learning model, for power attacks. The architecture of TransNet significantly differs from off-the-shelf TN models in several design choices which are very crucial for its success in power attacks. The existing state-of-the-art CNN-based models are designed depending on the amount of desynchronization in the attack traces or the trace length or both. Consequently, if the target device differs in one of those parameters, an entirely new model is required to be designed. On the other hand, TransNet architecture does not depend on such parameters, thus the same architecture can be used to attack different target devices.

- Experimentally, we have shown that TransNet can perform very well on highly desynchronized attack traces even when the model is trained on only synchronized (aligned) traces. In our experiments, TransNet can reduce the mean key rank below 1 using only 400 traces on ASCAD_desync100 dataset [BPS+20] even when it is trained on aligned traces (i.e. on ASCAD dataset). On the other hand, the CNN-based state-of-the-art models struggle to reduce the mean key rank below 20 using as much as 5000 traces in the same setting.

- We have also compared TransNet with the CNN-based state-of-the-art models when the models are trained using profiling desynchronization same as attack desynchronization. In this case, the performance of TransNet is comparable with that of the CNN-based models when the amount of desynchronization is small or medium and better by a wide margin when the amount of desynchronization exceeds a certain threshold.

The organization of the paper is as follows. In Section 2, we introduce the notations and briefly describe the power attack and how they are performed using deep learning. Section 3 describes the architecture of TN. In Section 4, we compare different neural network layers in terms of their ability to capture long-distance dependency. We also show the shift-invariance property of TN. Section 5 introduces TransNet, a TN based model for power attack. In Section 6, we experimentally evaluate the TransNet model and compare its performance with other state-of-the-art models. Finally, Section 7 concludes the work.

2 Preliminaries

In this section, we first introduce the notations used in the paper. Then, we briefly describe the steps of the power attack and profiling power attack. Finally, we briefly explain how profiling power attack is performed using deep neural networks.

2.1 Notations

Throughout the paper, we have used the following convention for notations. A random variable is represented by a letter in the capital (like X) whereas an instantiation of the random variable is represented by the corresponding letter in small (like x) and the domain of the random variable by the corresponding calligraphic letter (like $\mathcal{X}$). Similarly, a capital letter in bold (like $\mathbf{X}$) is used to represent a random vector, and the corresponding small letter in bold (like $\mathbf{x}$) is used to represent an instantiation of the random vector. A matrix is represented by a capital letter in roman type style (like M). The i-th elements of a vector $\mathbf{x}$ is represented by $\mathbf{x}[i]$ and the element of i-th row and j-th column of a matrix M is represented by $\mathrm{M}[i, j]$. $\mathrm{P}[\cdot]$ represents the probability distribution/density function and $\mathrm{E}[\cdot]$ represents expectation.

2.2 Power Attack

The power consumption of a semiconductor device depends on the values that are manipulated by the device. A power attack exploits this behavior of semiconductor devices to gain information about some intermediate sensitive variables of a cryptographic implementation, and thus, the secret key of the device. More precisely, in a power attack, an adversary takes control of the target device, also known as Device under Test or DUT, and collects power measurements or traces by executing the encryption (or decryption) process multiple times for different plaintexts (or ciphertexts). Then the adversary performs a statistical test to infer the secret key of the device.

The power attack can be of two types: profiling power attack and non-profiling power attack. In a profiling power attack, the adversary is assumed to possess a clone of the DUT under his control. Using the clone device, he can build a profile of the power consumption characteristic of the DUT and use that profile for performing the actual attack. On the other hand, in a non-profiling attack, the adversary does not possess any clone device and thus, cannot build any power profile of DUT. Instead, he tries to recover the secret key from the traces of DUT only. Since the profiling power attack assumes a stronger adversary, it often provides a worst case security analysis of a cryptographic implementation. In this paper, we consider profiling power attack only.

2.3 Profiling Power Attack

A profiling power attack is performed in two phases. In the first phase, which is also called the profiling phase, the adversary sets the plaintext and key of the clone device of his own choice and collects a sufficient number of traces. For each trace, the adversary computes the value of an intermediate secret variable $Z = F(X, K)$, where X is the random plaintext, K is the key, $F(\cdot, \cdot)$ is a cryptographic primitive and build a model for

$$\mathrm{P}[\mathbf{L}|Z] = \mathrm{P}[\mathbf{L}|F(X, K)] \tag{1}$$

where $\mathbf{L}$ represents a random vector corresponding to the power traces. The conditional probabilities $\mathrm{P}[\mathbf{L}|Z]$ serves as the leakage templates in the second phase.

In the second phase, also known as attack phase, the adversary collects a number of power traces $\{\mathbf{l}^i, p^i\}_{i=0}^{T_a-1}$, where $\mathbf{l}^i$, p^i are the i-th power trace and plaintext respectively and T_a is the total number of attack traces, by executing the DUT for randomly chosen plaintexts. For all the traces, the secrete key k^* is unknown but fixed. Finally the adversary

computes the score for each possible key as

$$\begin{aligned}\hat{\delta}_k &= \prod_{i=0}^{T_a-1} \mathrm{P}[Z = F(p^i, k)|\mathbf{L} = \mathbf{l}^i] \\ &\propto \prod_{i=0}^{T_a-1} \mathrm{P}[\mathbf{L} = \mathbf{l}^i|Z = F(p^i, k)] \times \mathrm{P}[F(p^i, k)] \end{aligned} \tag{2}$$

The key $\hat{k} = \mathrm{argmax}_k\, \hat{\delta}_k$ is chosen as the predicted key. If $\hat{k} = k^*$ holds, the prediction is said to be correct. This attack is also called Template attack as the estimated $\mathrm{P}[\mathbf{L}|Z]$ in Eq. 1 can be considered as the leakage template for different values of the sensitive variable Z.

2.4 Profiling Power Attack using Deep Learning

When deep learning methods are used to perform profiling power attack, instead of building a generative model, the adversary builds a discriminative model which maps the power trace L to the intermediate sensitive variable Z. In the profiling phase, the adversary trains a deep neural model $f : \mathbb{R}^n \mapsto \mathbb{R}^{|\mathcal{Z}|}$ using the power traces as input and the corresponding intermediate sensitive variables as the label. Thus, the output of the deep neural model for a power trace $\mathbf{l}$ can be written as

$$\mathbf{p} = f(\mathbf{l}; \theta^*) \tag{3}$$

where θ^* is the parameter learned during training, and $\mathbf{p} \in \mathbb{R}^{|\mathcal{Z}|}$ such that $\mathbf{p}[i]$, for $i = 0, \cdots, |\mathcal{Z}| - 1$, represents the predicted probability for the intermediate variable $Z = i$. During the attack phase, the score of each key $k \in \mathcal{K}$ is computed as

$$\hat{\delta}_k = \prod_{i=0}^{T_a-1} \mathbf{p}^i[F(p^i, k)] \tag{4}$$

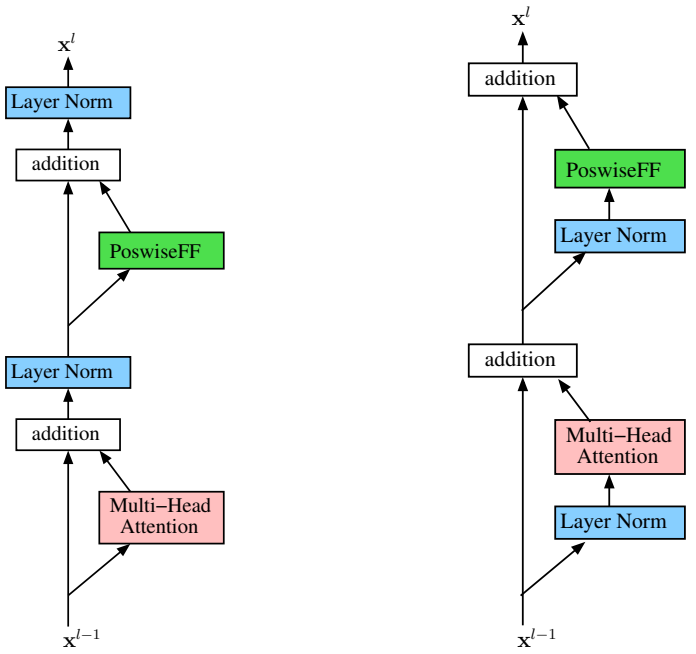
where $\{\mathbf{l}^i, p^i\}_{i=0}^{T_a-1}$ is the set of attack trace, plaintext pairs, and $\mathbf{p}^i = f(\mathbf{l}^i; \theta^*)$ is the predicted probability vector for the i-th trace. Like template attack, $\hat{k} = \mathrm{argmax}_k\, \hat{\delta}_k$ is chosen as the guessed key.

Several deep neural network architectures including Feed Forward Network (FFN), Convolutional Neural Network (CNN), Recurrent Neural Network (RNN) have been explored for profiling power attacks. In this work, we propose to use TN for the same. In the next section, we describe the architecture of a TN.

3 Transformer Network

Transformer network is a deep learning model which was originally developed for a sequence processing task. For a sequence of input tokens $(x_0, x_1, \cdots, x_{n-1})$ (in the context of power attack, the sequence is a power trace), the TN generates a sequence of output vectors $(\mathbf{y}_0, \mathbf{y}_1, \cdots, \mathbf{y}_{n-1})$. Each output vector $\mathbf{y}_i$, $0 \leq i < n$, captures any dependency of the input token x_i with other tokens of the input sequence. The sequence of the output vector can then be passed to different modules depending on the target task. For example, for a sequence classification task (as in the case of a power attack), one can take the mean of the output vectors and use the mean vector to predict the class labels.

Structurally, TN is a stacked collection of transformer layers following an initial embedding layer. Thus, in an L-layer TN, the output of the initial embedding layer is used as the input of the first transformer layer, and the output of the i-th transformer



(a) Post Layer Normalization

(b) Pre Layer Normalization

Figure 1: A Single Transformer Layer [XYH+20].

layer is used as the input of $(i+1)$-th layer, $1 \leq i < L$. Finally, the output of the L-th layer is taken as the network output.

A transformer layer consists of a multi-head self-attention layer followed by a position-wise feed-forward layer. To facilitate better training of deeper networks, the input and the output of both multi-head self-attention layer and position-wise feed-forward layer are connected by shortcut connection [HZRS16] and a layer normalization operation [BKH16] is applied at the output of the shortcut connection as can be seen in Figure 1a. The forward pass of an L-layer TN is shown in Algorithm 1. Figure 1b shows a slightly different version of the transformer layer which has been introduced later (please refer to Section 3.5 for details).

Given the above overall architecture, we now describe each building block of the network.

3.1 Embedding Layer

In natural language processing tasks, sequences of discrete symbols act as the input to the network. In the embedding layer, each symbol of the input sequence is replaced by a corresponding vector representation. More precisely, let V be the vocabulary size or the total number of distinct symbols. The embedding layer consists of an embedding matrix $\mathrm{E} \in \mathbb{R}^{V \times d}$ where d is the embedding dimension or model dimension. The i-th row of E is considered as the vector representation or vector space embedding of the i-th symbol for all $i = 0, \cdots, V-1$. The embedding layer takes the input sequence of discrete symbols $(x_0, x_1, \cdots, x_{n-1})$ as input, performs a lookup operation into the embedding matrix E and returns the sequence of vectors $(\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_{n-1})$ where $\mathbf{x}_i$ is the vector space embedding of the token x_i for $0 \leq i < n$. Thus, the operation in the embedding layer can be written

Algorithm 1: Forward pass of an L layer transformer network

At the beginning
begin
  $\mathbf{x}_0^0, \mathbf{x}_1^0 \cdots, \mathbf{x}_{n-1}^0 \leftarrow \text{Embed}(x_0, x_1, \cdots, x_{n-1})$ `// embed input sequence`
for $l \leftarrow 1$ *to* L **do**
  `// apply self-attention operation`
  $\mathbf{s}_0, \mathbf{s}_1, \cdots, \mathbf{s}_{n-1} \leftarrow \text{MultiHeadSelfAttn}^l\left(\mathbf{x}_0^{l-1}, \mathbf{x}_1^{l-1}, \cdots, \mathbf{x}_{n-1}^{l-1}\right)$
  `// add shortcut connection`
  $\mathbf{s}_0, \mathbf{s}_1, \cdots, \mathbf{s}_{n-1} \leftarrow \mathbf{s}_0 + \mathbf{x}_0^{l-1}, \mathbf{s}_1 + \mathbf{x}_1^{l-1}, \cdots, \mathbf{s}_{n-1} + \mathbf{x}_{n-1}^{l-1}$
  `// apply layer normalization operation`
  $\mathbf{s}_0, \mathbf{s}_1, \cdots, \mathbf{s}_{n-1} \leftarrow \text{LayerNormalization}\left(\mathbf{s}_0, \mathbf{s}_1, \cdots, \mathbf{s}_{n-1}\right)$

  `// apply position-wise feed-forward operation`
  $\mathbf{t}_0, \mathbf{t}_1, \cdots, \mathbf{t}_{n-1} \leftarrow \text{PoswiseFF}^l\left(\mathbf{s}_0, \mathbf{s}_1, \cdots, \mathbf{s}_{n-1}\right)$
  `// add shortcut connection`
  $\mathbf{t}_0, \mathbf{t}_1, \cdots, \mathbf{t}_{n-1} \leftarrow \mathbf{t}_0 + \mathbf{s}_0, \mathbf{t}_1 + \mathbf{s}_1, \cdots, \mathbf{t}_{n-1} + \mathbf{s}_{n-1}$
  `// apply layer normalization operation`
  $\mathbf{x}_0^l, \mathbf{x}_1^l, \cdots, \mathbf{x}_{n-1}^l \leftarrow \text{LayerNormalization}\left(\mathbf{t}_0, \mathbf{t}_1, \cdots, \mathbf{t}_{n-1}\right)$
return $\left(\mathbf{x}_0^L, \mathbf{x}_1^L, \cdots, \mathbf{x}_{n-1}^L\right)$

as

$$\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_{n-1} \leftarrow \text{Embed}(x_0, x_1, \cdots, x_{n-1}) \tag{5}$$

where $\mathbf{x}_i$ be the x_i-th row of the embedding matrix E for $i = 0, \cdots, n-1$. Typically, the embedding matrix is learned along with other parameters during training.

3.2 Multi-Head Self-Attention Layer

The multi-head self-attention layer is the key layer for the ability to capture long-distance dependency. Before describing multi-head self-attention, we describe the single head self-attention first.

Self-Attention For each ordered pair $(\mathbf{x}_i, \mathbf{x}_j)$ of input vectors, the self-attention operation computes the attention probability p_{ij} from vector $\mathbf{x}_i$ to vector $\mathbf{x}_j$ based on their similarity (sometimes also based on their positions). Finally, the output representation $\mathbf{y}_i$ for the i-th token is computed using the weighted sum of the input vectors where the weights are given by the attention probabilities i.e. $\mathbf{y}_i = \sum_j p_{ij}\mathbf{x}_j$. The state $\mathbf{y}_i$ is also called the contextual representation of the input vector $\mathbf{x}_i$ since $\mathbf{y}_i$ does not only depend on $\mathbf{x}_i$ but also on the surrounding tokens or the context. In the context of power attack, if $\mathbf{x}_i$ and $\mathbf{x}_j$ are two vectors corresponding to the leakages of two PoIs, the state $\mathbf{y}_i$ gets dependent on both the leakages in a single step even when the distance between i and j is large. Thus, this step eliminates the problem of capturing long-distance dependency.

To describe the self-attention operation more precisely, let $(\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_{n-1})$ and $(\mathbf{y}_0, \mathbf{y}_1, \cdots, \mathbf{y}_{n-1})$ be the sequence of input and output vectors of a self-attention layer where $\mathbf{x}_i, \mathbf{y}_i \in \mathbb{R}^d$ for all i. Then, for each $i = 0, \cdots, n-1$, the i-th output vector $\mathbf{y}_i$ is computed as follows:

1. first the attention scores a_{ij} from i-th token to j-th token, $0 \leq j < n$, is calculated

using a scaled dot product similarity measure, i.e.

$$a_{ij} = \frac{\langle \mathrm{W}_Q \mathbf{x}_i, \mathrm{W}_K \mathbf{x}_j \rangle}{\sqrt{d_k}} = \frac{\langle \mathbf{q}_i, \mathbf{k}_j \rangle}{\sqrt{d_k}} \tag{6}$$

where $\mathrm{W}_Q, \mathrm{W}_K \in \mathbb{R}^{d_k \times d}$ are trainable weight matrices and $\langle \cdot, \cdot \rangle$ denotes dot product of two vectors. $\mathbf{q}_i, \mathbf{k}_i \in \mathbb{R}^{d_k}$ are respectively known as *query* and *key* representation of the i-th token. Note that the term "key" used here has no relation with the term "(secret) key" used in cryptography.

2. the attention probabilities p_{ij} are computed by taking softmax of the attention scores a_{ij} over the j variable, i.e.,

$$p_{ij} = \mathrm{softmax}(a_{ij}; a_{i,0}, \cdots, a_{i,n-1}) = \frac{e^{a_{ij}}}{\sum_{k=0}^{n-1} e^{a_{ik}}} \tag{7}$$

3. the intermediate output $\bar{\mathbf{y}}_i$ is computed by taking the weighted sum of the input vectors $\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_{n-1}$ where the weight is given by the attention probabilities, i.e.

$$\bar{\mathbf{y}}_i = \sum_{j=0}^{n-1} p_{ij} \mathrm{W}_V \mathbf{x}_j = \sum_{j=0}^{n-1} p_{ij} \mathbf{v}_j \tag{8}$$

where $\mathrm{W}_V \in \mathbb{R}^{d_v \times d}$ is also a trainable weight matrix and $\mathbf{v}_j = \mathrm{W}_V \mathbf{x}_j$ is called the *value* representation of the j-th input vector $\mathbf{x}_j$.

4. the final output $\mathbf{y}_i$ is computed by projecting the d_v dimensional $\bar{\mathbf{y}}_i$ into $\mathbb{R}^d$, i.e.

$$\mathbf{y}_i = \mathrm{W}_O \bar{\mathbf{y}}_i \tag{9}$$

where $\mathrm{W}_O \in \mathbb{R}^{d \times d_v}$ be a trainable weight matrix.

Thus, the self-attention operation can be written as matrix multiplication in the following way (please refer to Appendix A for detail):

$$\begin{aligned} \bar{\mathrm{Y}} &= \text{Self-Attention}(\mathrm{W}_Q, \mathrm{W}_K, \mathrm{W}_V) = \mathrm{softmax}\left(\mathrm{A}\right) \mathrm{X}\mathrm{W}_V^T = \mathrm{P}\mathrm{X}\mathrm{W}_V^T \\ \mathrm{Y} &= \bar{\mathrm{Y}}\mathrm{W}_O^T \end{aligned} \tag{10}$$

where i-th row of matrices $\bar{\mathrm{Y}}$, Y and X are $\bar{\mathbf{y}}_i$, $\mathbf{y}_i$ and $\mathbf{x}_i$ respectively. W_V^T represents the transpose of the matrix W_V. A and P are two $n \times n$ matrices such that $\mathrm{A}[i,j]$ and $\mathrm{P}[i,j]$ equals to a_{ij} and p_{ij} respectively.

Multi-Head Self-Attention In self-attention, the matrix $\bar{\mathrm{Y}}$ created by a set of parameters $(\mathrm{W}_Q, \mathrm{W}_K, \mathrm{W}_V)$ is called a single attention head. In a H-head self-attention operation, H attention heads are used to produce the output. More precisely, the output of a multi-head self attention is computed as

$$\begin{aligned} \bar{\mathrm{Y}}^{(i)} &= \text{Self-Attention}(\mathrm{W}_Q^{(i)}, \mathrm{W}_K^{(i)}, \mathrm{W}_V^{(i)}), \text{ for } i = 0, \cdots, H-1 \\ \bar{\mathrm{Y}} &= [\bar{\mathrm{Y}}^{(0)}, \cdots, \bar{\mathrm{Y}}^{(H-1)}] \\ \mathrm{Y} &= \bar{\mathrm{Y}}\mathrm{W}_O^T \end{aligned} \tag{11}$$

where the function Self-Attention$(\cdot, \cdot, \cdot)$ is defined in Eq. 10, $[\mathrm{A}_1, \mathrm{A}_2, \cdots, \mathrm{A}_n]$ denotes the row-wise concatenation of the matrices A_is and the output projection matrix $\mathrm{W}_O \in \mathbb{R}^{d \times Hd_v}$ projects the Hd_v-dimensional vector into $\mathbb{R}^d$. A single head self-attention layer is said to captures the similarity between the input tokens in one way. An H-head self-attention layer can capture the similarity between the tokens in H-different ways.

Table 1: Notations used to denote the hyperparameters of a transformer network

Notation	Description
d	model dimension
d_k	key dimension
d_v	value dimension
d_i	inner dimension
n	input or trace length
L	number of transformer layers
H	number of heads in self-attention layer

3.3 Position-wise Feed-Forward Layer

Position-wise feed-forward layer is a two layer feed-forward network applied to each element of the input sequence separately and identically. Let $\text{FFN}(\mathbf{x})$ be a two layer feed-forward network with ReLU activation [GBB11]. Thus, $\text{FFN}(\mathbf{x})$ can be given by

$$\text{FFN}(\mathbf{x}; \text{W}_1, \text{W}_2, \mathbf{b}_1, \mathbf{b}_2) = \text{W}_2 \text{max}(\text{W}_1 \mathbf{x} + \mathbf{b}_1, 0) + \mathbf{b}_2 \tag{12}$$

where $\text{W}_1 \in \mathbb{R}^{d_i \times d}$, $\text{W}_2 \in \mathbb{R}^{d \times d_i}$ are two weight matrices, $\mathbf{b}_1 \in \mathbb{R}^{d_i}$, $\mathbf{b}_2 \in \mathbb{R}^d$ are two bias vectors and the $\max(\cdot)$ operation is applied element-wise. The integer d_i is commonly referred to as inner dimension. If the sequence $(\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_{n-1})$ is the input to the position-wise feed-forward layer, then the output sequence $(\mathbf{y}_0, \mathbf{y}_1, \cdots, \mathbf{y}_{n-1})$ is computed as

$$\mathbf{y}_i = \text{FFN}(\mathbf{x}_i), \text{ for } i = 0, 1, \cdots, n-1 \tag{13}$$

The position-wise feed-forward layer adds additional non-linearity to the network. In the context of power attack, this layer helps to increase the non-linearity of the learned function from the trace $\mathbf{l}$ to the sensitive variable Z. In Table 1, we summarize the notations used to describe the transformer network. In Appendix C, we have described the conventions which are used to set the value of the hyperparameters.

In the standard architecture, as described above, there are several design choices for TN which are relevant in the context of power attack. We found that the *positional encoding* and *layer normalization* need to be chosen properly to use TN for power attacks. Thus, we describe those, one by one.

3.4 Positional Encoding

Both the self-attention layer and position-wise feed-forward layer are oblivious to the ordering of input tokens. Thus, to capture the input order, positional encoding is used in TN. Two kinds of positional encodings are commonly used in TN: 1) absolute positional encoding and 2) relative positional encoding.

Absolute Positional Encoding In absolute positional encoding, a vector space encoding $\mathbf{p}_i \in \mathbb{R}^d$ is used for each position i, $i = 0, 1, \cdots, n-1$, of the input sequence. The positional encodings are added with the token embeddings element-wise before passing to the TN.

$$\mathbf{y}_0, \mathbf{y}_1, \cdots, \mathbf{y}_{n-1} \leftarrow \text{TransformerNetwork}(\mathbf{x}_0 + \mathbf{p}_0, \mathbf{x}_1 + \mathbf{p}_1, \cdots, \mathbf{x}_{n-1} + \mathbf{p}_{n-1}) \tag{14}$$

[VSP+17] have used two kinds of absolute positional encoding. In the first kind of absolute positional encodings, the positional encodings have been derived based on some rule and hard coded into the network. In the second kind of absolute positional encoding, the encodings are learned during training along with other network parameters.

Relative Positional Encoding One of the drawbacks of absolute positional encoding is that it makes the encoding of each token dependent on the position in which it appears. However, in most of the sequence processing tasks, the relative distance between any two tokens is more important than their absolute positions. In fact, making the token representations dependent only on the relative distances from other tokens helps to make the token representation shift-invariant (please refer to Section 4.2 for a detailed discussion).

The relative positional encoding is introduced in [SUV18]. To make the attention score from i-th token to j-th token dependent only on their relative position instead of their absolute position, they have modified the computation of the attention score of Eq. 6 as

$$a_{ij} = \frac{\langle \mathrm{W}_Q \mathbf{x}_i, \mathrm{W}_K \mathbf{x}_j + \mathbf{r}_{i-j} \rangle}{\sqrt{d_k}} = \frac{\langle \mathbf{q}_i, \mathbf{k}_j \rangle + \langle \mathbf{q}_i, \mathbf{r}_{i-j} \rangle}{\sqrt{d_k}} \tag{15}$$

where $\mathbf{q}_i, \mathbf{k}_j$ are as defined in Eq. 6 and the vectors $(\mathbf{r}_{-n+1}, \cdots, \mathbf{r}_0, \cdots, \mathbf{r}_{n-1})$ are the relative positional encoding and are also learned during training. [DYY+19] have interpreted the two terms of the numerator of RHS of the above equation as "content based addressing" and "content dependent (relative) positional bias" respectively. They have further improved the computation of the above attention score as

$$a_{ij} = \frac{\langle \mathrm{W}_Q \mathbf{x}_i, \mathrm{W}_K \mathbf{x}_j \rangle + \langle \mathrm{W}_Q \mathbf{x}_i, \mathbf{r}_{i-j} \rangle + \langle \mathrm{W}_Q \mathbf{x}_i, \mathbf{s} \rangle + \langle \mathbf{r}_{i-j}, \mathbf{t} \rangle}{\sqrt{d_k}} \tag{16}$$

$$= \frac{\langle \mathbf{q}_i, \mathbf{k}_j \rangle + \langle \mathbf{q}_i, \mathbf{r}_{i-j} \rangle + \langle \mathbf{q}_i, \mathbf{s} \rangle + \langle \mathbf{r}_{i-j}, \mathbf{t} \rangle}{\sqrt{d_k}} \tag{17}$$

where the last two terms of the numerator of the RHS of the above equation have been included to capture "global content bias" and "global (relative) positional bias" respectively. Finally, as before, the attention probabilities are computed as

$$p_{ij} = \mathrm{softmax}(a_{ij}; a_{i,0}, \cdots, a_{i,n-1}) = \frac{e^{a_{ij}}}{\sum_{k=0}^{n-1} e^{a_{ik}}} \tag{18}$$

In our TN model for power attacks, we have used the relative positional encoding given by Eq. (17).

3.5 Layer Normalization

Initially, TN has been introduced with "post-layer normalization" in which multi-head self-attention layers and position-wise feed-forward layers are followed by a layer normalization operation (please refer to Figure 1a). Later, pre-layer normalization (Figure 1b) has been introduced [BA19, CGRS19, WLX+19] in which the layer normalization is applied before the multi-head self-attention or position-wise feed-forward layers. Later study [XYH+20] found that pre-layer normalization stabilizes the training allowing the usage of a larger learning rate and thus, achieving faster convergence. However, in our experiments, we have found that the use of any kind of layer normalization in the network makes the network difficult to train for power attacks. We speculate that the layer normalization operations remove the informative data-dependent variations from traces, effectively making the input independent of the target labels. Thus, in our TN model, we have not used any layer normalization layer.

Proper training of TN is crucial for getting good performance. In the next section, we briefly discuss the training and learning rate scheduling algorithms which are used to train TN.

3.6 Training

Training a TN is a bit tricky in practice. An improper training might lead to divergence of the training loss or converging to a sub-optimal value. The selection of a proper

optimization algorithm and learning rate schedule play important roles to properly train a TN model.

3.6.1 Optimization Algorithm

Adam optimizer [KB15] is the most widely used optimization algorithm for training TN [VSP+17, CGRS19, WLX+19, DYY+19]. Several works [DCLT19, TCLT19, JCL+20] have used a slightly improved version of Adam optimizer call AdamW [LH19]. Recently, [YLR+20] has proposed new layer-wise adaptive optimization technique called LAMB for large batch optimization which has been used in several works [LCG+20].

3.6.2 Learning Rate Schedule

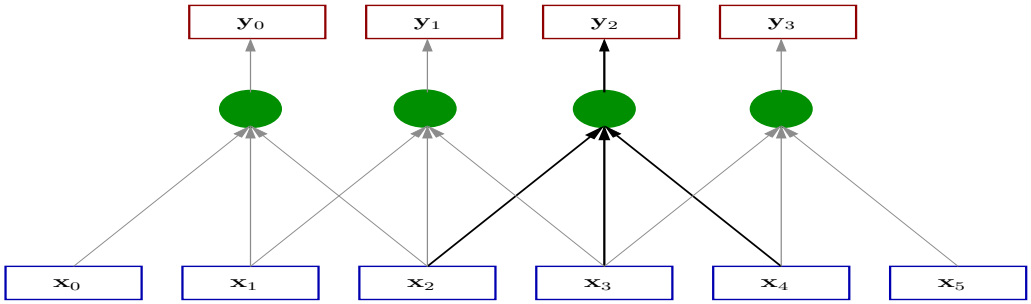
TNs are typically trained using a learning rate schedule which initially increases the learning rate from a low value until it reaches a maximum value (called max_learning_rate, which is a hyperparameter of the training algorithm). Once the maximum learning rate is reached, it is gradually decayed till the end of the training. The initial period in which the learning rate is increased is called *warm-up* period. The learning rate is typically increased linearly from zero to the maximum value during the warm-up period. Several common algorithms exist to decay the learning rate after the warm-up period. Linear decay, square-root decay, and cosine decay are some examples of commonly used learning rate schedules. Some of those learning rate scheduling algorithms have been discussed in Appendix B. To train our TN, we used cosine decay with a linear warm-up as the learning rate scheduling algorithm (please refer to Appendix B for a detailed description of the algorithm).

In Section 1, we argued that shift-invariance and the ability to capture long-distance dependency are two important properties to make a deep learning model effective against traces misalignments and masking countermeasures respectively. In the next section, we compare the ability to capture long-distance dependency of the transformer layer with other deep neural network layers. We also mathematically show that a TN with relative positional encoding can be shift-invariance as well.

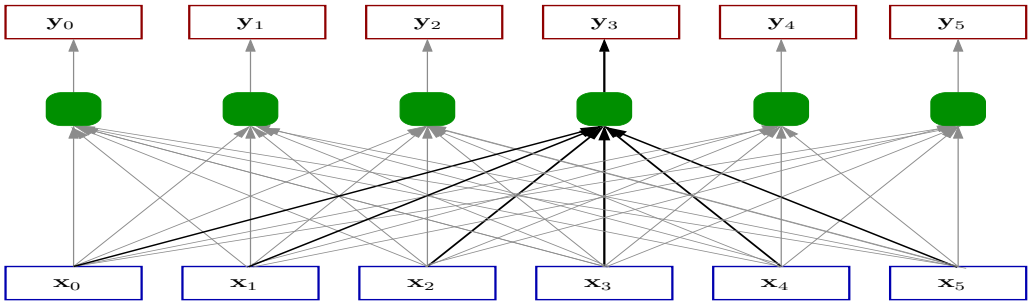
4 Ability to Capture Long Distance Dependency and Shift-Invariance of Transformer Network

4.1 Comparison of the Ability of Learning Long Distance Dependency

The problem of learning long-distance dependency using a deep neural network has been widely studied in the literature [HBFS01, VSP+17]. In [VSP+17], Vaswani et al. have argued that the key factor which affects the learning of dependency between two tokens is the length of the forward and backward propagation of signal between them. A shorter path between the two tokens makes it easier to learn their dependency whereas a longer path makes it difficult. They have further compared the ability to learn long-distance dependency of the three layers - convolutional layer, recurrent layer, and self-attention layer based on the path length between any two positions of the input sequence. When the two tokens are n distance apart, the maximum path length through the three neural network layers is shown in Table 2. As can be seen in the table, the self-attention layer can attend to any token of the input sequence using a constant number of steps (please refer to Figure 2b). A recurrent neural network requires $\mathcal{O}(n)$ steps. On the other hand, a stack of $\mathcal{O}(n/k)$ convolutional layers, where k is the kernel width, are required to connect the two tokens (please refer to Figure 2a). The maximum path length in a CNN can be reduced by interleaving pooling layers with convolutional layers or by using dilated convolution [YK16]. However, those approaches make CNN less shift-invariant. Another alternative to reduce



(a) A convolutional layer with kernel width 3. Each circle represents an element-wise weighted sum operation where weights are given by the kernel. All the circles share the same parameters. In this case, the i-th output token only depends on the i-th to $i+2$-th input tokens. Thus, capturing dependency between two tokens which are more than two distance apart requires more than one convolutional layers.



(b) A self-attention layer. Each rounded corner square of the figure represents the weighted sum of the input vectors where weights are set to be the attention probabilities. All of them share the same parameters. As shown in the figure, each output state becomes dependant on all input vectors after a single self-attention layer. Moreover, the number of parameters of a self-attention layer does not depend on the sequence length.

Figure 2: Comparison of propagation of dependency through convolutional and self-attention layers. An arrow from one state to another implies the second state is dependent on the first state.

the path length using a convolutional layer is to increase the kernel width k. However, as the number of parameters of a convolutional layer linearly depends on k, making k too large might explode the number of parameters. Moreover, increasing k to make it very large (i.e. making it comparable to the sequence length n) makes the convolutional layer less shift-invariant. Note that both transformer layer and fully connected layer make each of their output state dependant on all the input states, thus, both can capture long-distance dependency. However, unlike in a transformer layer where the number of parameters is independent of the input length, in a fully connected layer, the number of parameters is proportional to the input length. Thus, the use of a fully connected layer to capture long-distance dependency might make the number of parameters of the model very large leading to an overfitting problem during training. Moreover, a fully connected layer is not shift-invariant.

4.2 Shift-Invariance of Transformer Network

In computer vision, a function f is called invariant to a set of transformations $\mathcal{T}$ from $\mathcal{X}$ to $\mathcal{X}$ if $f(\mathbf{x}) = f(T(\mathbf{x}))$ holds for all $\mathbf{x} \in \mathcal{X}$ and $T \in \mathcal{T}$. In a power attack, the inputs are generally very noisy. In fact, in a power attack, one trace is often not sufficient to predict the correct key, instead, information from multiple traces is required to extract for the

Table 2: The maximum path length between any two tokens which are n distance apart through various types of neural network layers. Shorter the length, better the network at learning long-distance dependency [HBFS01, VSP+17]. Note that the maximum path length of a CNN can be reduced by interleaving convolutional layers with the pooling layers or using dilated convolutional layers. However, those approaches make the network less shift-invariant.

Layer Type	Self-Attention	Recurrent	Convolutional
Maximum Path Length	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n/k)$

same. Thus, in this context, we are interested in the information contained in $f(\mathbf{X})$ about the sensitive variable Z where $\mathbf{X}$ represents the random variable corresponding to the power traces. In other words, we are interested in the conditional probability distribution $\mathrm{P}[f(\mathbf{X})|Z]$. Thus, for power attacks, the invariance property can be defined in terms of $\mathrm{P}[f(\mathbf{X})|Z]$. However, for the shake of simplicity, we define the shift-invariance property only in terms of the conditional expectation $\mathrm{E}[f(\mathbf{X})|Z]$. Thus, in the context of power attack, we define the following weaker notion of invariance.

Definition 1. A function f is said to be invariant to a set of transformation $\mathcal{T}$ with associated probability distribution function $\mathcal{D}_{\mathcal{T}}$ if

$$\mathrm{E}[f(\mathbf{X})|Z] = \mathrm{E}[f(T(\mathbf{X}))|Z] \tag{19}$$

holds where $T \sim \mathcal{D}_{\mathcal{T}}$ and $\mathbf{X}$, Z are random variables respectively representing the input and intermediate sensitive variable. $\mathrm{E}[\cdot|Z]$ represents the conditional expectation where the expectation is taken over all relevant random variables other than Z.

To show the shift-invariance of TN, we consider the following transformer architecture, leakage model, and the set of transformations.

The Transformer Model We consider a single layer TN followed by a global pooling layer. The result can be approximately extended for multilayer TN as well. In the rest of the section, we denote the single layer TN followed by global pooling layer as $\mathrm{TN}_{1\mathrm{L}}$. The output of $\mathrm{TN}_{1\mathrm{L}}$ can be given by the following operations:

$$\begin{aligned}
&\mathbf{Y}_0, \cdots, \mathbf{Y}_{n-1} = \mathrm{SelfAttention}(\mathbf{X}_0, \cdots, \mathbf{X}_{n-1}) \\
&\mathbf{U}_0, \cdots, \mathbf{U}_{n-1} = \mathbf{Y}_0 + \mathbf{X}_0, \cdots, \mathbf{Y}_{n-1} + \mathbf{X}_{n-1} \\
&\mathbf{U}'_0, \cdots, \mathbf{U}'_{n-1} = \mathrm{FFN}(\mathbf{U}_0), \cdots, \mathrm{FFN}(\mathbf{U}_{n-1}) \\
&\mathbf{U}''_0, \cdots, \mathbf{U}''_{n-1} = \mathbf{U}'_0 + \mathbf{U}_0, \cdots, \mathbf{U}'_{n-1} + \mathbf{U}_{n-1}
\end{aligned} \tag{20}$$

$$\mathrm{TN}_{1\mathrm{L}}(\mathbf{X}_0, \cdots, \mathbf{X}_{n-1}) = \frac{1}{n} \sum_{i=0}^{n-1} \mathbf{U}''_i \tag{21}$$

where $(\mathbf{X}_0, \cdots, \mathbf{X}_{n-1})$ is the sequence of random vectors corresponding to the input of the network, $\mathrm{TN}_{1\mathrm{L}}(\mathbf{X}_0, \cdots, \mathbf{X}_{n-1})$ is the random vector corresponding to the final output (i.e. the output of global average pooling) of the network which is fed to a classification layer for the classification task. $\mathrm{SelfAttention}(\cdots)$ and $\mathrm{FFN}(\cdot)$ respectively represent the self-attention and position-wise feed-forward operations. Please refer to Algorithm 1 or Figure 1a for a description of a single layer TN. Additionally, we assume that the network uses relative positional encoding, and thus the attention scores and attention probabilities in the self-attention layer are computed by Eq. (16) and (18).

The Leakage Model To show the shift-invariance of TN, we consider the leakage model of a software implementation of a first-order masking scheme. However, the results can be easily extended for any higher-order masking scheme. More precisely, we take the following assumptions:

Assumption 1. *(Second Order Leakage Assumption) In the sequence of input vectors* $(\mathbf{X}_{-n+1+m_2}, \cdots, \mathbf{X}_0, \cdots, \mathbf{X}_{n-1}, \cdots, \mathbf{X}_{n-1+m_1})$*, the input vectors* $\mathbf{X}_{m_1}$ *and* $\mathbf{X}_{m_2}$ *(*$0 \leq m_1 < m_2 < n$*,* $m_2 - m_1 = l > 0$*) are the leakages corresponding to the mask* M *and masked sensitive variable* $Z_M = Z \oplus M$ *where* Z *is the sensitive variable. Thus, we can write* $\mathbf{X}_{m_1} = f_1(M) + \mathbf{N}_1$ *and* $\mathbf{X}_{m_1+l} = f_2(Z_M) + \mathbf{N}_2$ *where* $f_1, f_2 : \mathbb{R} \mapsto \mathbb{R}^d$ *are two deterministic functions of* M*,* Z_M *respectively and* $\mathbf{N}_1, \mathbf{N}_2 \in \mathbb{R}^d$ *are the noise component of* $\mathbf{X}_{m_1}$ *and* $\mathbf{X}_{m_1+l}$ *respectively. Note that,* $\mathbf{N}_1$ *and* $\mathbf{N}_2$ *are independent of both* M *and* Z_M*. The objective of the network is to learn a mapping from* $\{\mathbf{X}_{m_1}, \mathbf{X}_{m_1+l}\}$ *to* Z*.*

Assumption 2. *(IID Assumption) All the vectors* $\{\mathbf{X}_i\}_{-n+m_2<i<n+m_1}$ *are identically distributed. Moreover, all the variables of the set* $\{\mathbf{X}_i\}_{i \neq m_1, m_1+l}$ *are mutually independent. Additionally,* $\mathbf{X}_{m_1}$ *and* $\mathbf{X}_{m_1+l}$ *are independent to the rest of the random variables i.e* $\{\mathbf{X}_i\}_{i \neq m_1, m_1+l}$*.*

Note that the assumptions considered in the above leakage model are very well-known assumptions for a first-order masking scheme. In fact, previous studies [Mag19, TAM20] have taken such assumptions to generate synthetic power traces.

The Shift Transformations We define the set of shift transformations to be all shift transformations for which the PoIs (i.e. the leakage points corresponding to the mask M and the masked sensitive variable $Z_M = Z \oplus M$) do not go out of the trace window. More precisely, we define the set of transformations $\mathcal{T}^{\text{shift}}$ as

$$\begin{aligned} \mathcal{T}^{\text{shift}} &= \{T^s : s \in \mathbb{Z} \text{ and } -m_1 \leq s < n - m_2\} \\ \text{where, } T^s(\mathbf{X}_{-n+1+m_2}, \cdots, \mathbf{X}_0, \cdots, \mathbf{X}_{n-1}, \cdots, \mathbf{X}_{n-1+m_1}) &= \mathbf{X}_{0-s}, \cdots, \mathbf{X}_{n-1-s} \end{aligned} \tag{22}$$

In other words, the set of shift transformations $\mathcal{T}^{\text{shift}}$ consists of transformations T^s, where $-m_1 \leq s < n - m_2$, which shifts the input trace by s positions. The bound $-m_1 \leq s < n - m_2$ on the value of s ensures that the PoIs m_1 and m_2 do not go out of the window because of the shift operations. Note that the input to the transformations is a trace of size larger than n which is required as during the shift operations some sample points go out of the window and some sample points enter into the window.

With the above definitions, we summarize the main results of this section using the following proposition:

Proposition 1. *There exists a set of parameters for which* TN_{1L} *satisfies the following equation:*

$$\mathrm{E}\left[TN_{1L}(T(\mathbf{X}_{-n+1+m_2}, \cdots, \mathbf{X}_{n-1+m_1}))|Z\right] = \mathrm{E}\left[TN_{1L}(\mathbf{X}_0, \cdots, \mathbf{X}_{n-1})|Z\right] \tag{23}$$

where T *is a shift transformation drawn from any arbitrary distribution* $\mathcal{D}_{\mathcal{T}^{shift}}$ *over the set* $\mathcal{T}^{shift}$ *and the conditional expectations are taken over all the relevant variables other than* Z*.*

Before going to prove the above proposition, we will state and prove another result. The results are stated in the following lemma.

Lemma 1. *For any* $0 < \epsilon < 1$*, the parameters* W_Q*,* W_K*,* $\{\mathbf{r}_{-n+1}, \cdots, \mathbf{r}_0, \cdots, \mathbf{r}_{n-1}\}$ *and* $\mathbf{t}$ *of the transformer layer of* TN_{1L} *can be set such that* $p_{i,i+l} > 1-\epsilon$ *for all* $i = 0, \cdots, n-1-l$*, and* $p_{ij} = 1/n$ *for all* $i = n-l, \cdots, n-1$ *and* $j = 0, \cdots, n-1$ *hold where* $\mathrm{W}_Q, \mathrm{W}_K$*,* p_{ij}*,* $\{\mathbf{r}_{-n+1}, \cdots, \mathbf{r}_0, \cdots, \mathbf{r}_{n-1}\}$ *and* $\mathbf{t}$ *are as defined in Eq. (16) and (18).*

Proof. Since $\text{TN}_{1\text{L}}$ uses relative positional encoding, the attention scores in the self-attention layer is calculated following Eq. (16). Thus, from Eq. (16), we get the attention scores a_{ij}s as

$$a_{ij} = \frac{\langle \text{W}_Q \mathbf{x}_i, \text{W}_K \mathbf{x}_j \rangle + \langle \text{W}_Q \mathbf{x}_i, \mathbf{r}_{i-j} \rangle + \langle \text{W}_Q \mathbf{x}_i, \mathbf{s} \rangle + \langle \mathbf{r}_{i-j}, \mathbf{t} \rangle}{\sqrt{d_k}} \tag{24}$$

Setting W_Q, W_K, $\{\mathbf{r}_i\}_{i \neq l}$ all to zero of appropriate dimensions, $\mathbf{r}_l = c\sqrt{d_k}\mathbf{1}$ and $\mathbf{t} = \mathbf{1}$ where $\mathbf{1}$ is a vector whose only first element is 1 and rest are zero, and c is a real constant, we get

$$a_{ij} = \begin{cases} c & \text{if } j = i + l \\ 0 & \text{otherwise} \end{cases} \tag{25}$$

for $0 \leq i < n - l$. Thus, using Eq. 18, we compute the attention probabilities for $0 \leq i < n - l$ as

$$p_{ij} = \begin{cases} \frac{e^c}{e^c + n - 1} & \text{if } j = i + l \\ \frac{1}{e^c + n - 1} & \text{otherwise} \end{cases} \tag{26}$$

Setting $c > \ln\left(\frac{1-\epsilon}{\epsilon}\right) + \ln(n-1)$, we get $p_{i,i+l} > 1 - \epsilon$ for all $0 \leq i < n - l$ and $0 < \epsilon < 1$. Similarly, it is straight forward to show that $p_{ij} = 1/n$ for any $n - l \leq i < n$ and $0 \leq j < n$ for the same value of the parameters. □

In other words, Lemma 1 states that the attention probabilities can be learned during the model training such that the attention from the i-th sample point, for $0 \leq i < n-l$, can be mostly concentrated to the $(i + l)$-th sample point. Moreover, the attention probability $p_{i,i+l}$ can be made arbitrarily close to 1. Thus, to keep the proof of Proposition 1 simple, we take the following assumption on the trained $\text{TN}_{1\text{L}}$ model:

Assumption 3. *$P_{i,i+l} = 1$ for $0 \leq i < n-l$ where $P_{i,j}$ is the random variable representing the attention probability from i-th sample point to j-th sample point (and is defined by Eq. (18)) in the transformer layer of TN_{1L}. For $n - l \leq i < n$, $P_{i,j} = 1/n$ for all $j = 0, \cdots, n - 1$.*

Note that Assumption 3 is not true in general. However, it can be approximately true when we use a relative positional encoding. In Section 6.7, we have verified the correctness of this assumption in a practical scenario. Now, with Assumption 1, 2 and 3, we provide the proof of Proposition 1.

Proof of Proposition 1: Recall that the output of $\text{TN}_{1\text{L}}$ can be described by the following steps:

$$\begin{aligned}
&\mathbf{Y}_0, \cdots, \mathbf{Y}_{n-1} = \text{SelfAttention}(\mathbf{X}_0, \cdots, \mathbf{X}_{n-1}) \\
&\mathbf{U}_0, \cdots, \mathbf{U}_{n-1} = \mathbf{Y}_0 + \mathbf{X}_0, \cdots, \mathbf{Y}_{n-1} + \mathbf{X}_{n-1} \\
&\mathbf{U}'_0, \cdots, \mathbf{U}'_{n-1} = \text{FFN}(\mathbf{U}_0), \cdots, \text{FFN}(\mathbf{U}_{n-1}) \\
&\mathbf{U}''_0, \cdots, \mathbf{U}''_{n-1} = \mathbf{U}'_0 + \mathbf{U}_0, \cdots, \mathbf{U}'_{n-1} + \mathbf{U}_{n-1} \\
&\text{TN}_{1\text{L}}(\mathbf{X}_0, \cdots, \mathbf{X}_{n-1}) = \frac{1}{n} \sum_{i=0}^{n-1} \mathbf{U}''_i
\end{aligned}$$

From Eq. (8) and (9), we get

$$\mathbf{Y}_j = \text{W}_O \left(\sum_{k=0}^{n-1} P_{jk} \text{W}_V \mathbf{X}_k \right) \tag{27}$$

Thus, we can write $\mathbf{Y}_{m_1}$ (where m_1 is defined in Assumption 1) as

$$\mathbf{Y}_{m_1} = \mathrm{W}_O \left(\sum_{k=0}^{n-1} P_{m_1 k} \mathrm{W}_V \mathbf{X}_k \right) = \mathrm{W}_O \mathrm{W}_V \mathbf{X}_{m_1+l}, \text{ and thus} \tag{28}$$

$$\mathbf{U}_{m_1} = \mathrm{W}_O \mathrm{W}_V \mathbf{X}_{m_1+l} + \mathbf{X}_{m_1} \tag{29}$$

The last step of Eq. (28) follows since $i = m_1$ satisfies $P_{i,i+l} = 1$ in Assumption 3. Similarly we can write $\mathbf{Y}_i$ for $0 \leq i < n-l, i \neq m_1$ as

$$\mathbf{Y}_i = \mathrm{W}_O \left(\sum_{k=0}^{n-1} P_{ik} \mathrm{W}_V \mathbf{X}_k \right) = \mathrm{W}_O \mathrm{W}_V \mathbf{X}_{i+l}, \text{ and thus} \tag{30}$$

$$\mathbf{U}_i = \mathrm{W}_O \mathrm{W}_V \mathbf{X}_{i+l} + \mathbf{X}_i \tag{31}$$

For $n-l \leq i < n$, we can write

$$\mathbf{Y}_i = \frac{1}{n} \mathrm{W}_O \mathrm{W}_V \sum_{k=0}^{n-1} \mathbf{X}_k$$

$$\text{and, } \mathbf{U}_i = \frac{1}{n} \mathrm{W}_O \mathrm{W}_V \sum_{k=0}^{n-1} \mathbf{X}_k + \mathbf{X}_i$$

since, by Assumption 3, $P_{ij} = 1/n$ for $j = 0, \cdots, n-1$ and $n-l \leq i < n$. Now we compute $\mathbf{U}''_i$ for $i = 0, \cdots, n-1$.

$$\mathbf{U}''_i = \mathrm{FFN}(\mathbf{U}_i) + \mathbf{U}_i \tag{32}$$

Note that among all the $\{\mathbf{U}''_i\}_{0 \leq i < n}$, only $\mathbf{U}''_{m_1}$ and $\{\mathbf{U}''_i\}_{n-l \leq i < n}$ involve both the terms $\mathbf{X}_{m_1}$ and $\mathbf{X}_{m_1+l}$, thus can be dependent on the sensitive variable Z (from Assumption 1). Rest of the $\mathbf{U}''_i$s are independent of Z (from Assumption 2). The output of $\mathrm{TN}_{1\mathrm{L}}$ can be written as

$$\begin{aligned} \mathrm{TN}_{1\mathrm{L}}(\mathbf{X}_0, \cdots, \mathbf{X}_{n-1}) &= \frac{1}{n} \sum_{i=0}^{n} \mathbf{U}''_i \\ &= \frac{1}{n} \mathbf{U}''_{m_1} + \frac{1}{n} \sum_{0 \leq i < n-l, i \neq m_1} \mathbf{U}''_i + \frac{1}{n} \sum_{n-l \leq i < n} \mathbf{U}''_i \end{aligned} \tag{33}$$

The expectation of the output conditioned on Z can be given by

$$\begin{aligned} &\mathrm{E}[\mathrm{TN}_{1\mathrm{L}}(\mathbf{X}_0, \cdots, \mathbf{X}_{n-1}) | Z] \\ &= \mathrm{E}\left[\frac{1}{n} \mathbf{U}''_{m_1} + \frac{1}{n} \sum_{n-l \leq i < n} \mathbf{U}''_i + \frac{1}{n} \sum_{0 \leq i < n-l, i \neq m_1} \mathbf{U}''_i \,\middle|\, Z \right] \\ &= \frac{1}{n} \mathrm{E}[\mathbf{U}''_{m_1} | Z] + \frac{1}{n} \sum_{n-l \leq i < n} \mathrm{E}[\mathbf{U}''_i | Z] + \frac{1}{n} \sum_{0 \leq i < n-l, i \neq m_1} \mathrm{E}[\mathbf{U}''_i] \end{aligned} \tag{34}$$

The second step follows because the random variables $\{\mathbf{U}_i\}_{0 \leq i < n-l, i \neq m_1}$ are independent of Z. To complete the proof of Proposition 1, we compute

$$\begin{aligned} &\mathrm{E}\left[\mathrm{TN}_{1\mathrm{L}}(T^s(\mathbf{X}_{-n+1+m_2}, \cdots, \mathbf{X}_{n-1+m_1})) | Z\right] \\ &= \mathrm{E}\left[\mathrm{TN}_{1\mathrm{L}}(\mathbf{X}_{-s}, \cdots, \mathbf{X}_{n-1-s}) | Z\right] \\ &= \mathrm{E}\left[\frac{1}{n} \mathbf{U}''_{m_1} + \frac{1}{n} \sum_{n-l-s \leq i < n-s} \mathbf{U}''_i + \frac{1}{n} \sum_{-s \leq i < n-l-s, i \neq m_1} \mathbf{U}''_i \,\middle|\, Z \right] \\ &= \frac{1}{n} \mathrm{E}[\mathbf{U}''_{m_1} | Z] + \frac{1}{n} \sum_{n-l-s \leq i < n-s} \mathrm{E}[\mathbf{U}''_i | Z] + \frac{1}{n} \sum_{-s \leq i < n-l-s, i \neq m_1} \mathrm{E}\left[\mathbf{U}''_i\right] \end{aligned} \tag{35}$$

From Assumption 2, we get

$$\frac{1}{n}\sum_{n-l\le i<n}\mathrm{E}\left[\mathbf{U}''_i|Z\right]=\frac{1}{n}\sum_{n-l-s\le i<n-s}\mathrm{E}\left[\mathbf{U}''_i|Z\right],$$
$$\text{and }\frac{1}{n}\sum_{0\le i<n-l,i\neq m_1}\mathrm{E}\left[\mathbf{U}''_i\right]=\frac{1}{n}\sum_{-s\le i<n-l-s,i\neq m_1}\mathrm{E}[\mathbf{U}''_i]$$

Thus, comparing the right hand side of Eq. (34) and Eq. (35) we have

$$\mathrm{E}[\mathrm{TN}_{1\mathrm{L}}(\mathbf{X}_0,\cdots,\mathbf{X}_{n-1})|Z]=\mathrm{E}\left[\mathrm{TN}_{1\mathrm{L}}(T^s(\mathbf{X}_{-n+1+m_2},\cdots,\mathbf{X}_{n-1+m_1}))|Z\right]$$

which completes the proof.

□

4.3 Discussion

A CNN can also capture the long-distance dependency while maintaining shift invariance. Stack of several convolutional layers (without any interleaved pooling layers) followed by a global pooling layer (instead of flattening layer as in the existing neural network architecture [BPS+20, ZBHV20]) is shift invariant. To adopt such a network for capturing highly non-linear long-distance dependency, several fully connected layers can be added after the global pooling layer. The resultant network is both shift-invariant and capable of capturing long-distance dependency. However, the practical efficacy and advantages of such architectures need to be evaluated. In this work, we explore an alternative architecture based on TN.

In the previous section, we have seen that a TN is good at capturing long-distance dependency and, at the same time, also shift-invariant. In the next section, we propose a TN-based deep learning model, named TransNet, for power attack.

5 TransNet: A Transformer Network for Power Attack

For power attacks, we have developed a TN-based model called TransNet. TransNet is a multi-layer TN followed by a global pooling layer. More precisely, TransNet is a stack of an initial convolutional layer (which has been used as the embedding layer of the network), an optional average pooling layer, followed by several transformer layers followed by a global pooling layer which is finally followed by a classification layer and softmax layer. The schematic diagram of TransNet is shown in Figure 3. Here we describe key differences in TransNet architecture from off-the-shelf TN architectures.

Embedding Layer As discussed in Section 3.1, the input sequence $(x_0,\cdots,x_{n-1})$ to a TN is generally a sequence of discrete symbols $\{x_i\}_{i=0}^{n-1}$. The embedding layer converts the discrete tokens of the input sequence into their vector space representation. In side channel analysis, the inputs (i.e. power traces) are a sequence of continuous random variables. Thus, an embedding layer based on lookup table is not appropriate to use. Instead we have used an one dimensional convolutional layer with number of filters equals to d, where d is the model dimension, as the embedding layer. Thus, we can replace Eq. 5 of Section 3.1 as

$$\mathbf{x}_0,\mathbf{x}_1,\cdots,\mathbf{x}_{n-1}\leftarrow\mathrm{Conv1D}(l_0,l_1,\cdots,l_{n-1})$$

where $(l_0,l_1,\cdots,l_{n-1})$ is the sequence of real number representing a power trace. Note that in signal processing applications, any signal processing tasks are generally preceded by various preprocessing operations like smoothing by moving average

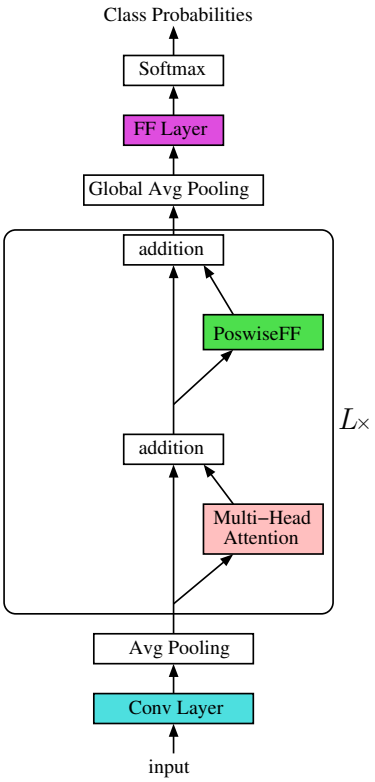


Figure 3: Architecture of TransNet.

operation. Such operation acts as a low pass filter and thus, reduce the noise in the signal [WHJ+20, WAGP20]. We expect the convolutional layer to perform some of those operations along with embedding the input signals. We have set the stride of the convolutional layer to 1 whereas the kernel width is considered as a hyperparameter.

Pooling Layer Though TN is very effective at capturing long-distant dependency, it is very costly when the input sequences are very long. In fact, the self-attention layer has a quadratic time and memory cost in terms of the sequence length. Thus, we have added an average pooling layer after the embedding layer. Setting a large value of pool_size in the pooling layer, would reduce the length of the sequences, and thus, providing efficient execution of the network at the cost of accuracy. In other words, this layer provides an efficacy-efficiency trade-off.

Transformer Layers The pooling layer is followed by several transformer layers. The architecture of the transformer layers is as described in Section 5 except layer normalization and self-attention layer. We have not used any layer normalization. And since we are using relative positional encoding, the attention probabilities in the self attention layers are computed using Eq. (16) and Eq. (18). The number of transformer layers to be used in the model is considered as a hyperparameter.

Layer Normalization The transformer layers are generally accompanied by layer normalization. As discussed in Section 3.5, layer normalization can be used in two different ways. However, for side-channel applications, we found it difficult to train our network when we add any kind of layer normalization. We suspect that the layer normalization operations remove the data-dependent variations from the input signals making the data uninformative. Thus, we have not used any layer normalization

operation in the transformer layers.

Positional Encoding As discussed in Section 3.4, both absolute and relative positional encodings are used in TN. For our network, we have used the relative positional encoding as given by Eq. (16) and (18) (introduced in [DYY+19]). We have already shown in Section 4.2 that the relative positional encoding scheme makes the TN shift-invariant.

Global Pooling The TN generates a sequence of vector as output, i.e.

$$\mathbf{y}_0, \cdots, \mathbf{y}_{n-1} \leftarrow \text{TransformerNetwork}(\mathbf{x}_0, \cdots, \mathbf{x}_{n-1}) \tag{36}$$

where each $\mathbf{y}_i$ is a vector of dimension d. In our network, we produce the final d-dimensional output vector $\bar{\mathbf{y}}$ by using a global average pooling, i.e.

$$\bar{\mathbf{y}} = \frac{1}{n} \sum_{i=0}^{n-1} \mathbf{y}_i \tag{37}$$

Class Probabilities The deep learning models for side-channel analysis are trained as classification tasks. Thus, given the d-dimensional output $\bar{\mathbf{y}}$, the class scores are computed as

$$s_i = \mathbf{w}_i^T \bar{\mathbf{y}} + b_i, \text{ for } i = 0, \cdots, C-1 \tag{38}$$

where C is the total number of classes, $\mathbf{w_i}$ and b_i are trainable weight vector and biases for class i. The class probabilities are computed using a softmax over the class scores i.e.

$$p_i = \text{softmax}(s_i; s_0, \cdots, s_{C-1}) = \frac{e^{s_i}}{\sum_{j=0}^{C-1} e^{s_j}} \tag{39}$$

The full architecture of the TransNet model has been shown in Figure 3. We trained the model using cross-entropy loss and Adam optimizer [KB15]. For the learning rate schedule, we have used cosine decay with a linear warm-up (please refer to Appendix B for the details of the learning rate scheduling algorithm).

In the next section, we provide the experimental results of TransNet.

6 Experimental Results

In this section, we experimentally evaluate the efficacy of TransNet for power attacks in the presence of both first-order masking countermeasure and trace misalignments. First, we evaluate the shift-invariance of TransNet on both synthetic and real datasets. Then, we compare the efficacy of TransNet with one CNN-based baseline model and three other state-of-the-art CNN-based methods. Next, we study the sensitivity of the performance of TransNet on the hyperparameters. We also evaluate the efficacy-efficiency trade-off provided by the pooling layer of TransNet. Finally, we investigate the outputs of the attention layers to understand their influence on the shift-invariance of the TransNet model.

6.1 Datasets

TransNet is particularly better than other alternatives in the presence of a combination of masking countermeasure and trace misalignments. Thus, to evaluate TransNet, we select a dataset with no first-order leakage. ASCAD database ([BPS+20]) is a collection of datasets

Table 3: Summary of the synthetic datasets.

	desync0	desync50	desync100
Profiling Dataset size	30000	30000	30000
Attack Dataset size	1000	1000	1000
Trace Length	200	200	200
Mask Leakage Point	25	25	25
Sbox Leakage Point	75	75	75
Profiling Dataset Desync	0	0	0
Attack Dataset Desync	0	50	100

which are publicly available and they do not have any first-order leakage. Thus we choose to evaluate TransNet on ASCAD datasets. Additionally, to verify the shift-invariance of TransNet, we perform experiments on the synthetic dataset as well. Here we provide the details of the datasets:

6.1.1 Synthetic Dataset

We use the following procedure for generating the synthetic traces

$$\mathbf{l}_i[j] = \begin{cases} M_i + \mathcal{N}(0, 30) & \text{if } j = 25 + s_i, \\ \text{SBOX}(X_i \oplus K) \oplus M_i + \mathcal{N}(0, 30) & \text{if } j = 75 + s_i, \\ R_{ij} + \mathcal{N}(0, 30) & \text{otherwise,} \end{cases} \tag{40}$$

where $\mathbf{l}_i[j]$ is the j-th sample point of the i-th trace, SBOX$(\cdot)$ represents the AES sbox and $\mathcal{N}(0, 30)$ is a real number generated from Gaussian distribution of 0 mean and 30 standard deviation. M_i, X_i are the mask and plaintext of the i-th trace and K is the secrete key that is held constant for all traces. Each of M_i, X_i and R_{ij} are uniformly random integer from the range $[0, 255]$. s_i is the random displacement of the i-th trace. Each of s_is is uniformly random integer from the range $[0, \text{desync})$ where desync indicates the maximum length of displacement in the dataset. Note that the above model for creating a synthetic dataset is common and previously used in the literature [Mag19, TAM20]. We generated 30000 traces for profiling and 1000 traces for evaluation. Each trace contains 200 sample points. For generating the profiling dataset, we set desync to 0. Thus, the profiling traces are perfectly aligned. For evaluation we generated three attack sets with desync value set to 0, 50 and 100. We provide the summary for our synthetic datasets in Table 3.

6.1.2 ASCAD Datasets

ASCAD datasets have been introduced by [BPS+20]. The original dataset is a collection of $60,000$ traces collected from the ATMega8515 device which runs a Software implementation of AES protected by a first-order masking scheme. Each trace contains $100,000$ sample points. From the original dataset, they have further created three datasets named ASCAD_desync0, ASCAD_desync50, and ASCAD_desync100. Each of the three derived datasets contains $50,000$ traces for profiling and $10,000$ traces for the attack. Further, for computational efficiency, the length of each trace of the derived datasets is reduced by keeping only 700 sample points that correspond to the interval $[45400, 46100)$ of the original traces. The window corresponds to the leakages of the third sbox of the first round of AES execution. The ASCAD_desync0 has been created without any desynchronization of the traces. However, ASCAD_desync50 and ASCAD_desync100 have been created by randomly shifting the traces where the length of random displacements have been generated from a uniform distribution in the range $[0, 50)$ in case of ASCAD_desync50

Table 4: Summary of the ASCAD datasets.

	desync0	desync50	desync100	desync200	desync400
Profiling Dataset Size	50000	50000	50000	50000	50000
Attack Dataset Size	10000	10000	10000	10000	10000
Indices of Profiling Traces	$[0, 50000)$	$[0, 50000)$	$[0, 50000)$	$[0, 50000)$	$[0, 50000)$
Indices of Attack Traces	$[50000, 60000)$	$[50000, 60000)$	$[50000, 60000)$	$[50000, 60000)$	$[50000, 60000)$
Trace Length	700	700	700	1500	1500
Target Points	$[45400, 46100)$	$[45400, 46100)$	$[45400, 46100)$	$[45000, 46500)$	$[45000, 46500)$
Profiling Dataset Desync	0	50	100	200	400
Attack Dataset Desync	0	50	100	200	400

and $[0, 100)$ in case of ASCAD_desync100. Note that the random displacements have been added to both the profiling traces and attack traces.

Apart from the above three derived datasets, we created two more datasets namely ASCAD_desync200 and ASCAD_desync400 using the API provided by [BPS+20]. As the name suggests, we have misaligned the traces by a random displacement in the range $[0, 200)$ for ASCAD_desync200 dataset and $[0, 400)$ for ASCAD_desync400 dataset. Each trace of the two derived datasets is 1500 sample point long and corresponds to the interval $[45000, 46500)$ of the original traces. We provide a summary of the derived datasets in Table 4.

6.2 Hyper-parameter Setting of TransNet

Unless stated otherwise, we have set the number of transformer layers to 2, pool size of the pooling layer to 1. We have set the model dimension d to 128, the dimension of the key vectors and value vectors (i.e. d_k and d_v) to 64 and the number of head H to 2. We set the kernel width of the convolutional layer to 11 for the experiments on ASCAD datasets and 1 for the experiments on synthetic datasets. The complete list of the hyperparameter setting of our implemented TransNet is given in Appendix C.

In the next section, we verify the shift-invariance property of TransNet.

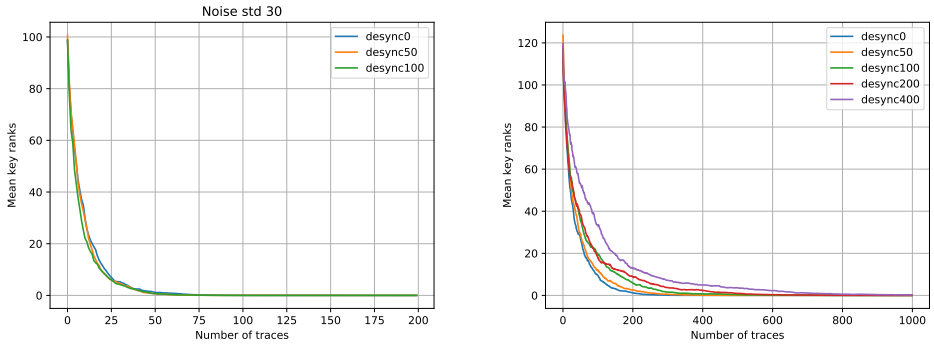
6.3 Shift-Invariance of TransNet

The goal of this section is to evaluate the achieved shift-invariance of TransNet. Earlier, we have formally shown that TransNet can be shift-invariant. In this section, we experimentally investigate the achieved shift-invariance of our trained TransNet model. Towards that goal, we train TransNet using aligned traces and evaluate the trained model using misaligned traces. Note that the profiling traces of the derived ASCAD datasets other than ASCAD_desync0 are misaligned. But, for the experiments of this section, we trained all the models on only aligned traces.

6.3.1 Results

The results of TransNet on the synthetic datasets are shown in Figure 4a. Note that the profiling traces of the synthetic datasets do not have any misalignments. Thus, we train a single model and use it to evaluate all three synthetic datasets. The plots of Figure 4a suggest that the performance of TransNet is similar on all the three datasets though they are desynchronized by different amounts. Moreover, the TransNet performs very well on the highly desynchronized 'desync100' dataset even though the model is trained using

traces without any desynchronization. This provides strong evidence of the shift-invariance property of the TransNet model as claimed in the theoretical analysis



(a) Results of TransNet on synthetic dataset. (b) Results of TransNet on ASCAD dataset.

Figure 4: Evaluation of the shift-invariance of TransNet model. Figure (a) plots the results for synthetic datasets and Figure (b) plots the same for ASCAD datasets. All the models used for the evaluations are trained using only aligned traces, the misalignments have been introduced only in the attack traces. desync$X$ implies that the attack traces are randomly shifted by an amount in the range $[0, X)$. Results on both synthetic and real datasets suggest the model's robustness to trace misalignments justifying the theoretical claim.

To examine whether the shift-invariance property of the TransNet model persists for the real dataset, we performed experiments on the derived ASCAD datasets. Note that the length of traces of the first three derived datasets namely ASCAD_desync0, ASCAD_desync50 and ASCAD_desync100 is 700 and the last two derived datasets namely ASCAD_desync200 and ASCAD_desync400 is 1500. Thus, we trained two TransNet models. The first one was trained for trace length 700 and we evaluated it on ASCAD_desync0, ASCAD_desync50 and ASCAD_desync100 datasets. The second model was trained for trace length 1500 and we evaluated that model on the ASCAD_desync200 and ASCAD_desync400 datasets. Both the models were trained using only aligned traces. The results are plotted in Figure 4b. The figure shows that the results get only slightly worse as the amount of trace desynchronization gets larger. Thus, it can be considered as strong evidence for achieving almost shift-invariance by the TransNet models.

Next, we compare the efficacy of TransNet with other state-of-the-art methods.

6.4 Comparison with Other Methods

The goal of this section is to compare the results of TransNet model with that of other state-of-the-art models on various ASCAD datasets (please refer to Section 6.1 for the details of the datasets). We compare TransNet with the following CNN-based approaches:

CNNBest The CNNBest model has been introduced in [BPS+20]. It has five convolutional layers followed by one flattening layer and three fully connected layers. The authors have shown that CNNBest performs similar to other profiling power attacks when the amount of trace misalignments is low and better by a large margin when the amount of trace misalignments is high. Thus, we choose CNNBest as a baseline method to compare with TransNet. To evaluate CNNBest model on ASCAD_desync0, ASCAD_desync50 and ASCAD_desync100 datasets, we have used the trained model provided by them online. For the evaluation on ASCAD_desync200 and ASCAD_desync400 datasets, we have trained the model on the two datasets using their code.

EffCNN In [ZBHV20], Zaid et al. have proposed a methodology for constructing CNN-based models that are robust to trace misalignments. By doing experiments across various datasets, they have shown that their method can indeed be used to construct CNN models which performs much better than other alternatives while having very small model sizes at the same time. Thus, we used models constructed from their approach to compare with TransNet. Following the recommendation of [WAGP20], for performing experiments using EffCNN models, we have vertically standardized the traces of ASCAD_desync0 dataset and horizontally standardized the traces for the rest of the datasets.

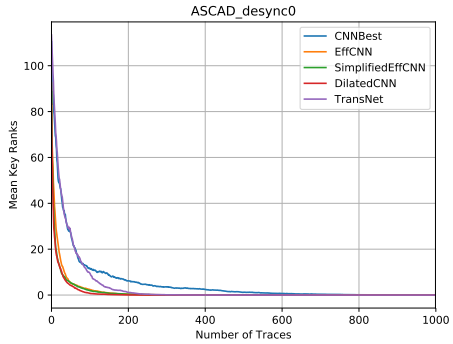
SimplifiedEffCNN In [WAGP20], the authors have suggested removing the first convolutional and batch normalization layer of the EffCNN models. They have shown that these simplified EffCNN models are easier to train and provides an improvement in attack efficiency. Thus, we also compare SimplifiedEffCNN models with TransNet. Like EffCNN, we have used vertical standardization for ASCAD_desync0 and horizontal standardization of for the rest of the datasets as a preprocessing.

DilatedCNN In [PA20], Paguada et al. suggested to use dilated convolutional layer to capture long distance dependency. Thus, we compare TransNet with their approach. For the comparison, we created two models - one for the datasets ASCAD_desync0 and ASCAD_desync100 (for which the trace length is 700), and the other for the datasets ASCAD_desync200 and ASCAD_desync400 (for which the trace length is 1500). Following their approach, we replaced the first convolutional layer of the two models with dilated convolution. We tuned the models for additional four sets of hyper-parameters: $[l_k = 16, dr = 4]$, $[l_k = 16, dr = 6]$, $[l_k = 32, dr = 3]$ and $[l_k = 64, dr = 2]$ where l_k and dr are respectively the kernel width and dilation rate of the dilated convolutional layer. Please refer to Appendix D for the details of the models used. As for EffCNN and SimplifiedEffCNN, we have used vertical standardization of ASCAD_desync0 and horizontal standardization for the rest of the datasets as a preprocessing.

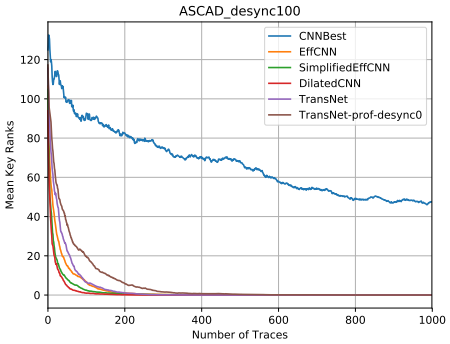
6.4.1 Results

For the experiments of this section, all the CNN-based models have been trained using profiling desync same as attack desync. On the other hand, for each of the experiments, we trained two TransNet models: one using no profiling desync and other using profiling desync same as the attack desync. We refer the model which is trained using profiling desync as 'TransNet' and the model which is trained using no profiling desync as 'TransNet-prof-desync0'. The results are shown in Figure 5. From Figure 5a, we observe that on ASCAD_desync0 dataset i.e. when there is no trace misalignment, all the methods perform well. However, as the amount of trace desynchronization gets larger, the performance of CNNBest and EffCNN becomes inferior by a large margin compared to the other four methods (Figure 5c). The performance of the rest of the four methods are similar up to desync 200. However, for desync 400, all the CNN-based alternatives struggle to bring down the mean key rank below 20 using as much as 5000 traces whereas TransNet requires about 800 traces to bring it down to 0 (Figure 5d). Moreover, the TransNet-prof-desync0 model which has been trained on only synchronized traces, also bring down the mean key rank below 1 using only 1000 traces suggesting the robustness of TransNet training to the amount of desync in the profiling traces.

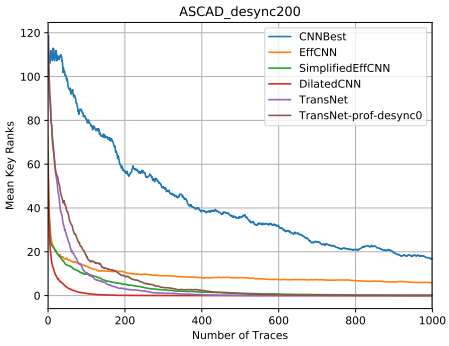
To compare the robustness of TransNet with the CNN-based methods to the amount of desynchronization in the profiling traces, we trained two models for all the methods - one with profiling desync 0 and other with desync 200 and tested the models with attack desync 0 and 400. The results can be seen in Figure 6. In Figure 6a, we plot results of the models which have been trained using profiling desync 0. In this case, all the methods perform equally well for attack desync 0. However, for attack desync 400, TransNet reduces



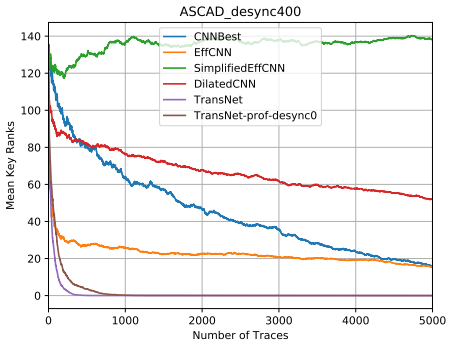
(a) Results on ASCAD_desync0.



(b) Results on ASCAD_desync100.



(c) Results on ASCAD_desync200.

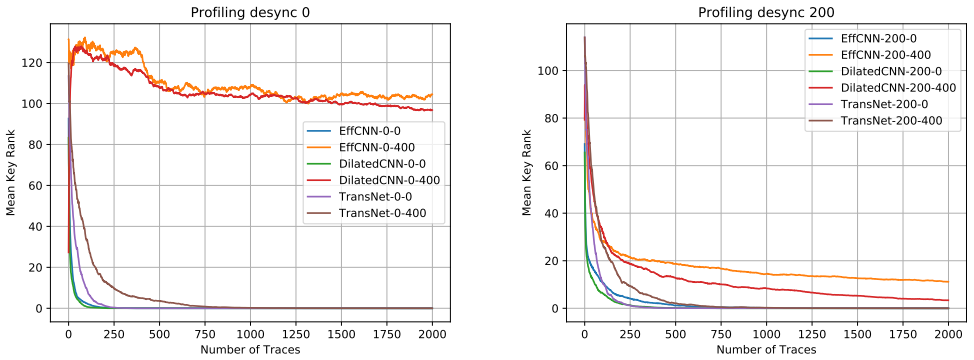


(d) Results on ASCAD_desync400.

Figure 5: Comparison of TransNet with CNNBest, EffCNN, SimplifiedEffCNN and DilatedCNN models on the ASCAD datasets. Figure (a), (b), (c) and (d) respectively plots the results for dataset ASCAD_desync0, ASCAD_desync100, ASCAD_desync200 and ASCAD_desync400. TransNet-prof-desync0 denotes the TransNet model which is trained using no profiling desynchronization. For rest of the models, the distribution of desynchronization in profiling and attack traces are the same.

the average key rank below 1 using 1000 traces whereas the CNN-based methods fails to bring down the average key rank significantly below 100 using 2000 traces. The results for profiling desync 200 are shown in Figure 6b. In this case, though the CNN-based methods still perform considerably good for attack desync 0, their performances for attack desync 400 still widely lagging the performance of TransNet. Overall, the Figure 6 shows that the performance of TransNet is almost invariant to the profiling desync whereas the CNN-based methods are required to be trained using profiling desync whose value is close to the value of attack desync.

In summary, we can say that TransNet performs far better than CNNBest on desynchronized attack traces. Though other CNN-based models perform similar or slightly better than TransNet when the amount of desynchronization in the attack traces is comparatively small, their performances get poor as the amount of desynchronization crosses a threshold. Moreover, TransNet can perform very well on highly desynchronized attack traces even when the model is trained on only aligned profiling traces. In this aspect, our work is in



(a) Profiling desync 0.

(b) Profiling desync 200.

Figure 6: Comparison of the robustness of TransNet and CNN-based models to profiling desync. Figure (a) plots results for models which are trained using no profiling desync. On the other hand, Figure (b) plots results for models which are trained using profiling desync 200. The label of the form ⟨model_name⟩-X-Y denotes model ⟨model_name⟩ trained with profiling desync X and evaluated with attack desync Y.

line with the work of [CDP17] in which Cagli et al. have argued for an end-to-end model replacing the critical preprocessing steps like the realignments of traces. However, their model was still required to be trained on misaligned profiling traces to perform well on misaligned attack traces. Our approach can be considered as going a step forward in that direction where the model can be trained on aligned profiling traces as well to make it perform well on misaligned attack traces.

Finally, constructing deep learning models using the strategy of [ZBHV20] requires to design separate model architectures for different target devices and different amount of target trace desynchronization. Later in [PA20], Paguada et al. have relaxed this requirement. However, they still require to create different model architecture for different trace lengths. On the other hand, TransNet provides a single monolithic architecture which is rich enough to capture the complexity of the broader problems. Thus, the same architecture can be tuned using a different set of profiling traces to make it work for different target devices and thus, it reduces the necessity of human intervention further.

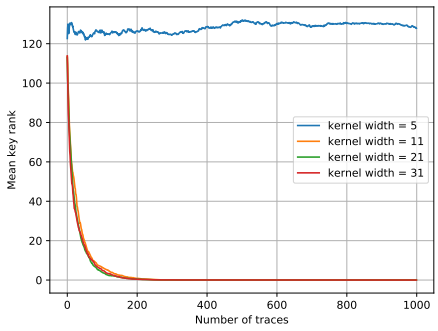
Next, we study the sensitivity of TransNet to different hyperparameters.

6.5 Sensitivity of TransNet to Different Hyperparameters and Hyperparameter Tuning

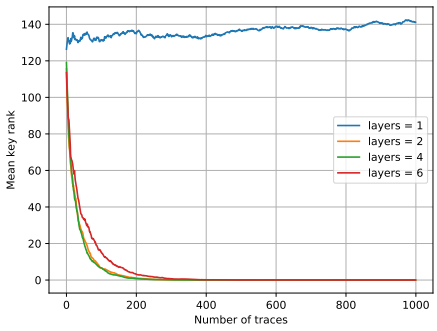
The number of hyperparameters introduced by TransNet is considerably large (please refer to Appendix C for a full list of TransNet hyperparameters). However, we set most of the hyperparameters following the standard convention used in transformer literature (please refer to Appendix C) leaving only the following hyperparameters to be tuned:

d_model The hyperparameter, also has been denoted by d, represents the model dimension or the output dimension of the transformer layers.

n_head The hyperparameter, also has been denoted by H, represents the number of heads used in the multi-head self-attention layers.



(a) Results with varying kernel width of first convolutional layer.



(b) Results with varying number of transformer layers.

Figure 7: Sensitivity of TransNet to kernel width and number of transformer layers. Figure (a) shows the results with varying kernel widths and Figure (b) shows with varying numbers of layers. The models have been both trained and evaluated with desync 100.

conv_kernel_size The hyperparameter represents the kernel width of the first convolutional layer of TransNet.

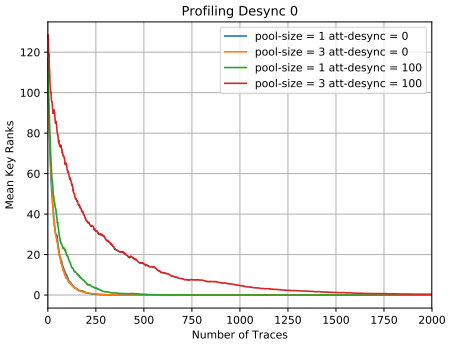
pool_size The hyperparameter represents the pool size (the stride is also set to be equal to pool size) of the average pooling layer after the convolutional layer of TransNet.

n_layer The hyperparameter, also has been denoted by L, represents the number of transformer layers in TransNet.

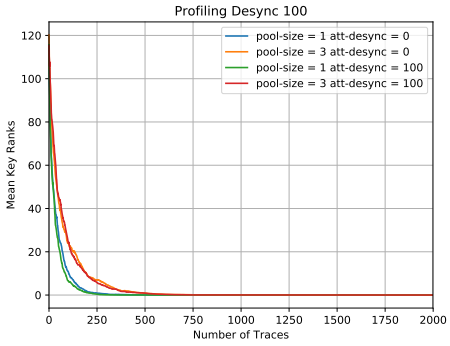
Among the above five hyperparameters, pool_size provides an efficacy-efficiency trade-off (please refer to Section 6.6 for a detailed discussion) and can be set to 1 for obtaining best results. We found that a default value of 128 for d_model and 2 for n_head works very well across the experiments. In this section, we study the sensitivity of TransNet to the remaining two hyperparameters namely conv_kernel_width and n_layer. We have shown the results in Figure 7. In Figure 7a, we plot the results of TransNet by varying the kernel width. From the figure, we can observe that, though the network fails to converge for kernel width 5, it performs almost similarly for kernel width 11, 21, and 31. This implies that kernel width can be chosen to be any value from a wide range like $[11, 31]$. Next, we study the sensitivity of TransNet to the number of transformer layers. The results are plotted in Figure 7b. As it can be seen in the figure, though the network has failed to converge for n_layer equal to 1, it has performed almost equally well for n_layer equal to 2, 4 and 6. Consequently, it can be said that any value for n_layer in the range $[2, 6]$ is good enough for TransNet. Thus, we can conclude that most of the hyperparameters of TransNet can be set to the default value. It also performs well for a wide range of values for the rest of the hyperparameters which reduces the overhead of hyperparameter tuning further.

6.6 Efficiency-Efficacy Trade-off by Varying Pool Size

In this section, we investigate the effect of different pool sizes on the TransNet results. Note that we expect the results of TransNet to be worse while gaining some computational efficiency for a larger value of pool size. Moreover, a study like [Zha19] demonstrated the



(a) Results for profiling desync 0.



(b) Results for profiling desync 100.

Figure 8: Effect of pool size on TransNet results. Figure (a) plots the results for models which are trained using profiling desync 0 and Figure (b) plots for models which are trained using profiling desync 100.

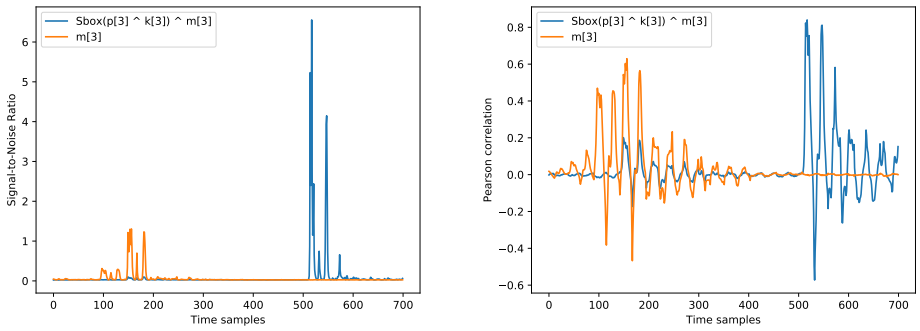
Table 5: Training time for different pool sizes.

	Pool Size 1	Pool Size 3
Training Time (sec/1000 traces)	0.41	0.06

loss of shift-invariance of deep neural networks by the usage of the pooling layer. Thus, we also expect the loss of shift-invariance of TransNet with the increase of the pool size. Thus, we performed experiments with pool size 1 and 3 on ASCAD datasets. Figure 8a plots the results for profiling desync 0. In this case, both pool sizes perform similarly on attack desync 0. However, when evaluated on attack traces with desync 100, the model with a pool size of 3 performs substantially worse than that with pool size 1. We suspect the loss of shift-invariance of the TransNet model with increased pool size as the cause for its worse performance for attack desync 100. Next, we performed the same experiments with profiling desync 100. The results are shown in Figure 8b. In this case, the results of the model with pool size 3 for attack desync 100 has improved over the results obtained from the same model with profiling desync 0 though the other results remain almost the same. Thus, we conclude that introducing misalignments in the profiling traces can significantly improve the performance of TransNet when the pool size is greater than 1. To observe the gain of using a larger pool size on the training time of TransNet, we measured the training time of TransNet with the two pool sizes. The training times are shown in Table 5. As can be seen from the table, a TransNet model with a pool size of 3 is almost 6.8 times faster than the one with pool size 1. Thus, using a larger value of pool size provides computational efficiency in the exchange of a slight deterioration in its effectiveness.

6.7 Attention Probabilities and Shift Invariance of TransNet

For showing the shift invariance of transformer network in Section 4.2, we assumed that $p_{i,i+l} = 1$ for all $i = 0, \cdots, n-l$ (Assumption A3 of Section 4.2) where $p_{i,j}$ is the attention probability from i-th sample point to j-th sample point and l is the distance between the mask leakage point and sbox leakage point. In this section, we show that the assumption approximately holds for TransNet. We show that, in TransNet, $p_{i,j}$ is on an average significantly large for $|i-j| = l$ and close to zero otherwise. To determine the range of values for l on ASCAD dataset, we computed the SNR (Signal-to-Noise Ratio) and



(a) Signal-to-Noise Ratio.

(b) Pearson correlation.

Figure 9: Signal-to-Noise Ratio and Pearson correlation on ASCAD dataset.

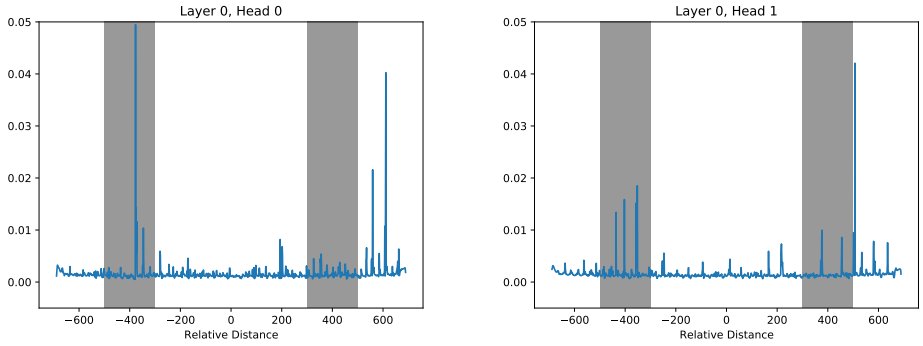
Pearson correlation for $m[3]$ and $\text{Sbox}(p[3] \oplus k[3]) \oplus m[3]$ where $m[3]$, $p[3]$, and $k[3]$ are the mask, plaintext, and key byte of the third sbox of the first round. The SNR and Pearson correlation are plotted in Figure 9. The plot of SNR reveals that the leakages due to the manipulation of the mask byte are spread over approximately 100 to 200-th sample points. On the other hand, the leakages due to the manipulation of the output of the sbox are spread over approximately 500 to 600-th sample points. Thus the value of l ranges between $[300, 500]$. Consequently, we expect high attention probabilities between sample points when their relative distances are about 300 to 500 or -500 to -300. To verify this fact, we plotted average attention probabilities between sample points according to their relative distances. More precisely, we have computed the average attention probabilities for relative distance r where r vary from $-(n-1)$ to $n-1$ (n is the trace length) as follows:

$$\mathcal{R}_r = \{(i,j) | 0 \leq i,j < n \text{ and } i - j = r\}$$
$$p_{\text{avg}}(r) = \frac{1}{T_a |\mathcal{R}_r|} \sum_{0 \leq k < T_a} \sum_{i,j \in \mathcal{R}_r} p_{ij}^k$$

where T_a is the total number of attack traces, p_{ij}^k denotes the attention probability from i-th token to j-th token of k-th trace and $|\mathcal{X}|$ denotes the cardinality of set $\mathcal{X}$. The plots are shown in figure 10.

The gray regions in the plots correspond to the region in which we expect the attention probabilities to be significantly higher than those in other regions. The peaks in or near the gray regions of Figure 10a and 10b indicate that both the attention heads of 0-th layer are putting significantly higher attention weight to sample points which are approximately 300 to 500 distance apart. Additionally, the 0-th attention head also shows some peaks for relative distances close to 600. This may be due to the sbox leakages near the 600 to 700-th sample points which are not properly visible in the SNR plot (Figure 9a) but are more prominent in the correlation plot (Figure 9b).

In the previous paragraph, we have observed that the self-attention layers put significantly higher attention probabilities to sample points which are approximately 300 to 500 distance apart from the target sample points. Since the attention probabilities do not depend on any positional information of the sample points apart from the relative positional encoding, we hypothesize that the higher attention probabilities depending on the relative distances of the sample points are contributed by the relative positional encoding. To verify the hypothesis, we investigate the contribution of relative positional encoding in the attention probabilities and its dependency on the relative distance of the sample points. Towards that goal, we plot the average differences in the probability scores



(a) Layer 0, Head 0.

(b) Layer 0, Head 1.

Figure 10: Plots of average attention probabilities against relative distances between two sample points. The gray area corresponds to the distances between leakages of the mask and masked sbox on the ASCAD dataset. Figure (a) and (b) respectively plots the results for the first and second head of the self-attention of the first transformer layer.

from a sample point to another while using and not using the relative positional encoding. More precisely, for each relative distance r, $-(n-1) \leq r < n$, we compute the average difference score as

$$\begin{aligned} \mathcal{R}_r &= \{(i,j) | 0 \leq i,j < n \text{ and } i-j = r\} \\ p_{\text{diff}}(r) &= \frac{1}{T_a |\mathcal{R}_r|} \sum_{0 \leq k < T_a} \sum_{i,j \in \mathcal{R}_r} \left(p_{ij}^k - \tilde{p}_{ij}^k \right), \end{aligned} \tag{41}$$

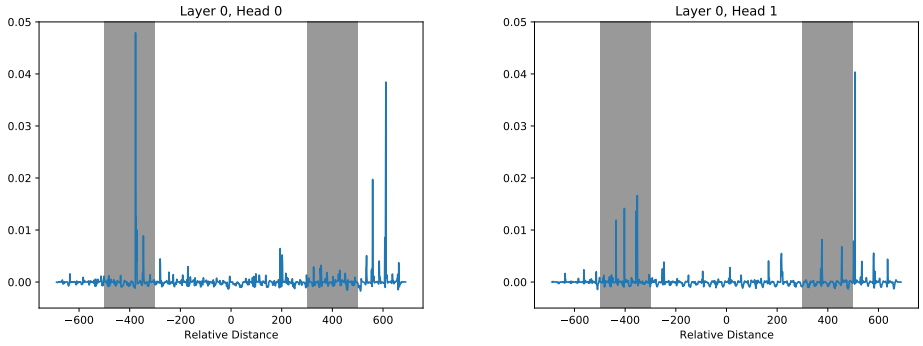
where $\tilde{p}_{ij}^k$ is defined (ignoring the super script k) as

$$\begin{aligned} \tilde{a}_{ij} &= \frac{\langle \mathbf{q}_i, \mathbf{k}_j \rangle + \langle \mathbf{q}_i, \mathbf{u} \rangle}{\sqrt{d_k}} \\ \tilde{p}_{ij} &= \text{softmax}\left(\tilde{a}_{ij}; \tilde{a}_{i,0}, \cdots, \tilde{a}_{i,n-1}\right) \end{aligned}$$

Note that $\tilde{p}_{ij}$ has been defined by excluding the terms related to the relative positional encoding from the definition of p_{ij} (Eq. 17). The plots are shown in Figure 11. By comparing Figure 11 with Figure 10, we can observe that the peaks of the two figures coincide with each other. Thus, we can conclude that the high peaks in the attention probabilities are caused by the relative positional encoding.

7 Conclusion

In this work, we have proposed to use TN for power attacks. A TN is better than other networks such as RNN or CNN in capturing long-distance dependency. Thus, it is a natural choice for a higher-order power attack. Using relative positional encoding, TN can be made shift-invariant as well. Thus, it is also highly effective against misaligned traces. We have proposed TransNet, a TN-based deep learning model, for power attacks. We have also experimentally evaluated the proposed TransNet model on synthetic datasets as well as ASCAD datasets. The results show that TransNet performs very well on highly misaligned attack traces even when it is trained on only aligned profiling traces. In comparison with other state-of-the-art deep learning models, it performs better by a wide margin on ASCAD datasets when the attack traces are highly misaligned. It also



(a) Layer 0, Head 0.

(b) Layer 0, Head 1.

Figure 11: Plots of $p_{\text{diff}}(r)$ against relative distance r between two sample points. The gray area corresponds to the distances between leakages of the mask and masked sbox on the ASCAD dataset.

performs considerably better than other models when the amount of desynchronization in the profiling traces is significantly less than that in the attack traces.

Recently, CNN-based deep learning models have gained popularity for performing power attacks. In this work, we present the TN-based deep learning model as a feasible alternative to the CNN-based models. The training of our proposed model, TransNet, is almost invariant to the amount of desynchronization in the profiling traces, and thus a better choice than CNN-based models when the attack traces are highly desynchronized or the amount of desynchronization in the profiling traces is significantly less than that in the attack traces. Moreover, the existing state-of-the-art CNN-based models are designed based on the input length or the amount of desynchronization in the attack traces or both. If the target device differs in any of the above parameters, entirely new models are required to be designed. Our proposed TransNet provides a single architecture that can be tuned for different target devices reducing the requirement of human intervention in power attacks further. We believe, starting from this work, the transformer network can be further improved for performing power attacks.

References

[AG01] Mehdi-Laurent Akkar and Christophe Giraud. An implementation of DES and AES, secure against some attacks. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 309–318. Springer, 2001.

[BA19] Alexei Baevski and Michael Auli. Adaptive input representations for neural language modeling. In *ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[BKH16] Lei Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *CoRR*, abs/1607.06450, 2016.

[BPS+20] Ryad Benadjila, Emmanuel Prouff, Rémi Strullu, Eleonora Cagli, and Cécile Dumas. Deep learning for side-channel analysis and introduction to ASCAD database. *J. Cryptogr. Eng.*, 10(2):163–188, 2020.

[CDP17] Eleonora Cagli, Cécile Dumas, and Emmanuel Prouff. Convolutional neural networks with data augmentation against jitter-based countermeasures - profiling attacks without pre-processing. In Wieland Fischer and Naofumi Homma, editors, *CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 45–68. Springer, 2017.

[CG00] Jean-Sébastien Coron and Louis Goubin. On boolean and arithmetic masking against differential power analysis. In Çetin Kaya Koç and Christof Paar, editors, *CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000*, volume 1965 of *Lecture Notes in Computer Science*, pages 231–237. Springer, 2000.

[CGRS19] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *CoRR*, abs/1904.10509, 2019.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *CRYPTO '99, Santa Barbara, California, USA, August 15-19, 1999*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.

[CK09] Jean-Sébastien Coron and Ilya Kizhvatov. An efficient method for random delay generation in embedded software. In Christophe Clavier and Kris Gaj, editors, *CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 156–170. Springer, 2009.

[CK10] Jean-Sébastien Coron and Ilya Kizhvatov. Analysis and improvement of the random delay countermeasure of CHES 2009. In Stefan Mangard and François-Xavier Standaert, editors, *CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*, pages 95–109. Springer, 2010.

[DCLT19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[DYY+19] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc Viet Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In Anna Korhonen, David R. Traum, and Lluís Màrquez, editors, *ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 2978–2988. Association for Computational Linguistics, 2019.

[GBB11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey J. Gordon, David B. Dunson, and Miroslav Dudík, editors, *AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011*, volume 15 of *JMLR Proceedings*, pages 315–323. JMLR.org, 2011.

[HBFS01] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber. Gradient flow in recurrent nets: The difficulty of learning long-term dependencies. 2001.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016.

[JCL+20] Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S. Weld, Luke Zettlemoyer, and Omer Levy. Spanbert: Improving pre-training by representing and predicting spans. *Trans. Assoc. Comput. Linguistics*, 8:64–77, 2020.

[KB15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[KPH+19] Jaehun Kim, Stjepan Picek, Annelie Heuser, Shivam Bhasin, and Alan Hanjalic. Make some noise. unleashing the power of convolutional neural networks for profiled side-channel analysis. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):148–179, 2019.

[LCG+20] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A lite BERT for self-supervised learning of language representations. In *ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[LH19] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[Mag19] Houssem Maghrebi. Deep learning based side channel attacks in practice. *IACR Cryptol. ePrint Arch.*, 2019:578, 2019.

[Mes00] Thomas S. Messerges. Securing the AES finalists against power analysis attacks. In Bruce Schneier, editor, *FSE 2000, New York, NY, USA, April 10-12, 2000, Proceedings*, volume 1978 of *Lecture Notes in Computer Science*, pages 150–164. Springer, 2000.

[MPP16] Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. Breaking cryptographic implementations using deep learning techniques. In Claude Carlet, M. Anwar Hasan, and Vishal Saraswat, editors, *SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings*, volume 10076 of *Lecture Notes in Computer Science*, pages 3–26. Springer, 2016.

[MZ13] Zdenek Martinasek and Vaclav Zeman. Innovative method of the power analysis. *Radioengineering*, 22(2):586–594, 2013.

[PA20] Servio Paguada and Igor Armendariz. The forgotten hyperparameter: - introducing dilated convolution for boosting cnn-based side-channel attacks. In Jianying Zhou, Mauro Conti, Chuadhry Mujeeb Ahmed, Man Ho Au, Lejla Batina, Zhou Li, Jingqiang Lin, Eleonora Losiouk, Bo Luo, Suryadipta Majumdar, Weizhi Meng, Martín Ochoa, Stjepan Picek, Georgios Portokalidis, Cong Wang, and Kehuan Zhang, editors, *ACNS 2020 Satellite Workshops, AIBlock, AIHWS, AIoTS, Cloud S&P, SCI, SecMT, and SiMLA, Rome, Italy, October 19-22, 2020, Proceedings*, volume 12418 of *Lecture Notes in Computer Science*, pages 217–236. Springer, 2020.

[SUV18] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In Marilyn A. Walker, Heng Ji, and Amanda Stent, editors, *NAACL-HLT, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 2 (Short Papers)*, pages 464–468. Association for Computational Linguistics, 2018.

[TAM20] Dhruv Thapar, Manaar Alam, and Debdeep Mukhopadhyay. Transca: Cross-family profiled side-channel attacks using transfer learning on deep neural networks. *IACR Cryptol. ePrint Arch.*, 2020:1258, 2020.

[TCLT19] Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Well-read students learn better: The impact of student initialization on knowledge distillation. *CoRR*, abs/1908.08962, 2019.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *NIPS, Long Beach, CA, USA, December 4-9, 2017*, pages 5998–6008, 2017.

[WAGP20] Lennert Wouters, Victor Arribas, Benedikt Gierlichs, and Bart Preneel. Revisiting a methodology for efficient CNN architectures in profiling attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):147–168, 2020.

[WHJ+20] Yoo-Seung Won, Xiaolu Hou, Dirmanto Jap, Jakub Breier, and Shivam Bhasin. Back to the basics: Seamless integration of side-channel pre-processing in deep neural networks. *IACR Cryptol. ePrint Arch.*, 2020:1134, 2020.

[WLX+19] Qiang Wang, Bei Li, Tong Xiao, Jingbo Zhu, Changliang Li, Derek F. Wong, and Lidia S. Chao. Learning deep transformer models for machine translation. In Anna Korhonen, David R. Traum, and Lluís Màrquez, editors, *ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 1810–1822. Association for Computational Linguistics, 2019.

[XYH+20] Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. On layer normalization in the transformer architecture. In *ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 10524–10533. PMLR, 2020.

[YK16] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. In Yoshua Bengio and Yann LeCun, editors, *ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[YLR+20] Yang You, Jing Li, Sashank J. Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training BERT in 76 minutes. In *ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[ZBHV20] Gabriel Zaid, Lilian Bossuet, Amaury Habrard, and Alexandre Venelli. Methodology for efficient CNN architectures in profiling attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(1):1–36, 2020.

[Zha19] Richard Zhang. Making convolutional networks shift-invariant again. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 7324–7334. PMLR, 2019.

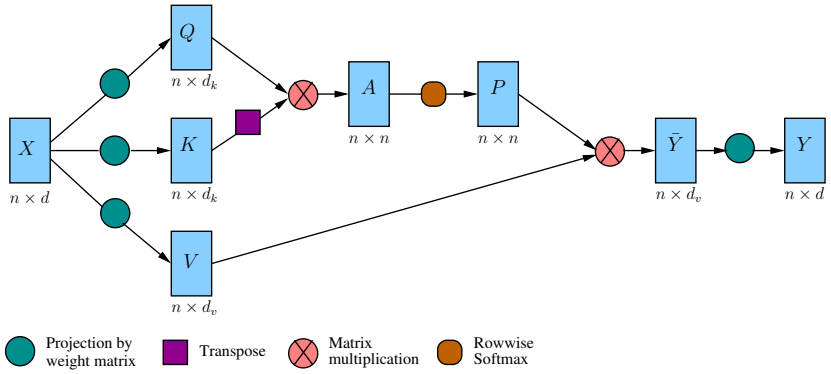


Figure 12: Single Head Self-Attention.

[ZLLS20] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. 2020. https://d2l.ai.

[ZS20] Yuanyuan Zhou and François-Xavier Standaert. Deep learning mitigates but does not annihilate the need of aligned traces and a generalized resnet model for side-channel attacks. *J. Cryptogr. Eng.*, 10(1):85–95, 2020.

A Self-Attention Operation

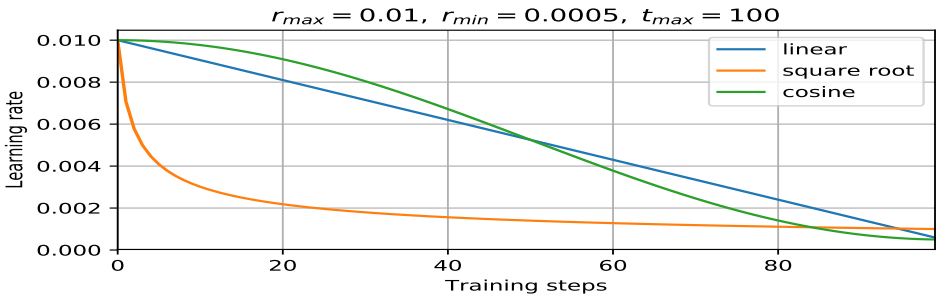
The output of a single head self-attention operation can be written using the following equations:

$$
\begin{aligned}
\bar{\mathrm{Y}} &= \text{Self-Attention}(\mathrm{W}_Q, \mathrm{W}_K, \mathrm{W}_V) \\
&= \text{softmax}\left(\frac{\mathrm{QK}^T}{\sqrt{d_k}}\right)\mathrm{V} \\
&= \text{softmax}\left(\frac{\mathrm{XW}_Q^T\mathrm{W}_K\mathrm{X}^T}{\sqrt{d_k}}\right)\mathrm{XW}_V^T \\
&= \text{softmax}\left(\mathrm{A}\right)\mathrm{XW}_V^T \\
&= \mathrm{PXW}_V^T \\
\mathrm{Y} &= \bar{\mathrm{Y}}\mathrm{W}_O^T
\end{aligned}
$$

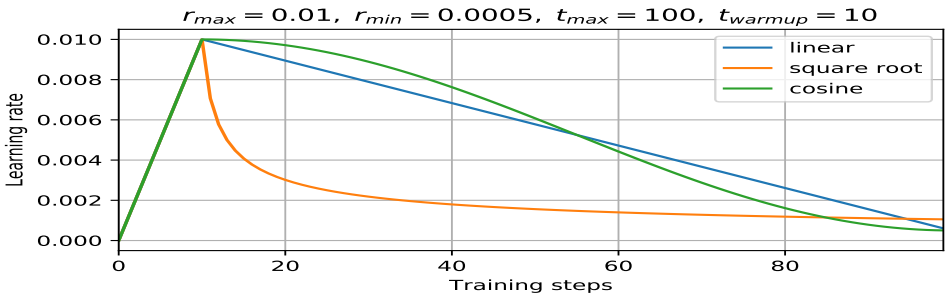
where $\mathrm{X}, \mathrm{Q}, \mathrm{K}, \mathrm{V}, \mathrm{Y}$ and $\bar{\mathrm{Y}}$ are the matrices whose i-th row are $\mathbf{x}_i$, $\mathbf{q}_i$, $\mathbf{k}_i$, $\mathbf{v}_i$, $\mathbf{y}_i$ and $\bar{\mathbf{y}}_i$ respectively. M^T and softmax(M) respectively represent the transpose and row-wise softmax operations applied to the matrix M. As stated before, the entry of i row and j-th column of the matrices A and P are a_{ij} and p_{ij} respectively where a_{ij} and p_{ij}s are respectively defined in Eq. 6 and 7. The schematic diagram of self-attention layer is shown in Figure 12.

B Learning Rate Scheduling

Learning rate schedule plays an important role in any stochastic gradient descent based optimizations. Setting proper learning rate scheduling requires several considerations [ZLLS20].



(a) Without any warm-up steps



(b) With linear warm-up for 10 steps

Figure 13: Plots of learning rates against training steps using various learning rate scheduling algorithms. Each step corresponds to the processing of one batch of training examples.

- First of all, the absolute value of learning rate is important as setting a large value for learning rate might make the optimization algorithm to diverge where as setting a small value might make the algorithm to take too long to reach the optimal value.

- Secondly, the learning rate should be decreased with training steps properly as decreasing the learning rate too slowly might make the optimization algorithm to oscillate around the optimal value instead of reaching the value whereas decreasing the learning rate too fast might results in a failure to reach the optimal value.

- Thirdly, for the optimization of a large deep neural network, setting a large value of learning rate during the initial period of the training might result in sub-optimal optimization. Since the initial set of parameters are random, the initial gradient direction might not be too informative. Thus, setting a large gradient update during the initial period of the optimization might lead to converging to a sub-optimal minima. One common approach to optimize a large neural network is to use several *warm-up* steps, during which the learning rate is gradually increased to a large value starting from a small value. After the *warm-up* steps, some standard learning rate scheduling is followed.

Here, we describe several commonly used learning rate schedule.

Linear Decay In linear decay scheduling, the learning rate is decreased linearly with the training steps. More precisely, at t-th step, the learning rate r_t is computed as

$$r_t = r_{\max} + \frac{r_{\min} - r_{\max}}{t_{\max}} \times t \tag{42}$$

where $r_{\max}$, $r_{\min}$ are the maximum and minimum learning rate and $t_{\max}$ is the maximum training step. All of $r_{\max}$, $r_{\min}$ and $t_{\max}$ are considered as hyper-parameters.

Square Root Decay In square root decay scheduling, the learning rate is decreased as the square root of the training steps. More precisely, at t-th step, the learning rate r_t is computed as

$$r_t = r_{\max} \times \frac{1}{\sqrt{1+t}} \tag{43}$$

where $r_{\max}$ is the maximum/initial learning rate. $r_{\max}$ is generally considered as a hyper-parameter.

Cosine Decay In cosine decay scheduling, the learning rate is decreased following a cosine curve. Thus, at t-th step, the learning rate r_t is computed as

$$r_t = r_{\max} + \frac{r_{\max} - r_{\min}}{2} \times (1 + \cos(\pi t / t_{\max})) \tag{44}$$

where $r_{\max}$, $r_{\min}$ and $t_{\max}$ are as defined before and are hyper-parameters of the scheduling algorithm.

The evolution of learning rate with the increase in training steps in various learning rate scheduling algorithms are shown in Figure 13a.

As stated earlier, for the training of large neural network models, several steps are kept as warm-up steps during which the learning rate is warmed up to a desired value starting from a small value. For training TN, it is common to use a linear warm-up schedule. In a linear warm-up schedule, the learning rate is linearly increased to $r_{\max}$ starting from a low value (most often zero) over a duration of t_{warmup} steps where t_{warmup}, known as warmup steps, is a hyper-parameter. Figure 13b plots the evolution of learning rate with training step in the above three learning scheduling algorithm with a linear warm-up.

C Hyper-parameter Setting

We set several hyperparameters of TransNet using the standard conventions which are followed in natural language processing (NLP). We set $d_k = d_v = d/H$ which is commonly followed in NLP. In NLP, d_i is set to $4d$. However, we found $d_i = 2d$ also provide good results, thus we set $d_i = 2d$. In NLP, the number of heads H is set to 16. Considering the simplicity of the problem in power attack, we set this hyperparameter to 2. The input length n is set to be equal to the trace length. The relative positional encoding takes one hyperparameter named clamp_len. It is enough to set this hyperparameter to be equal to n. Our implemented TransNet model takes two hyperparameters for dropout: dropout and dropatt. In NLP, these two hyperparameters are set to 0.1. We found a value of 0.05 or 0.1 works equally well.

Complete list of hyperparameters used for training TransNet is given below.

hyper-parameters	ASCAD_desync			
	0	100	200	400
n_layer (L)	2	2	2	2
d_model (d)	128	128	128	128
n_head (H)	2	2	2	2
d_head (d_v)	64	64	64	64
d_inner (d_i)	256	256	256	256
dropout	0.05	0.05	0.05	0.05
dropatt	0.05	0.05	0.05	0.05
conv_kernel_size	11	11	11	11
pool_size	1	1	1	1
clamp_len	690	690	1500	1500
untie_r	True	True	True	True
smooth_pos_emb	False	False	False	False
untie_pos_emb	True	True	True	True
init	normal	normal	normal	normal
init_std	0.02	0.02	0.02	0.02
max_learning_rate	0.00025	0.00025	0.00025	0.00025
(gradient) clip	0.25	0.25	0.25	0.25
min_lr_ratio	0.004	0.004	0.004	0.004
warmup_steps	0	0	0	0
batch_size	256	256	256	256
train_steps	30000	50000	80000	100000

For the experiments on the synthetic datasets, we used the same hyper-parameter setting except *conv_kernel_size*, *clamp_len* and *train_step*. We set the *conv_kernel_size* to one, *clamp_len* to 200 and *train_step* to 1000.

Note that we set some hyperparameters like init, init_std, max_learning_rate, clip, min_lr_ratio and warmup_step to the default value used in the implementation of [DYY+19].

D Details of Architectures of DilatedCNN Models

In [PA20], Paguada et al. have used two CNN models one for ASCAD fixed key dataset and the other for ASCAD random key dataset. For ASCAD fixed key dataset, they have used the EffCNN model [ZBHV20] as the base model for their experiments. Since we have also performed experiments on ASCAD fixed key dataset, we build two models using the approach proposed in [ZBHV20]. Since in the approach of [ZBHV20], the model architectures depend on the trace length and the amount of desynchronization, we build two models - one for datasets ASCAD_desync0 and ASCAD_desync100 in which the trace length is 700, and the other for datasets ASCAD_desync200 and ASCAD_desync400 in which the trace length is 1500. Further, for building the model for the datasets ASCAD_desync0 and ASCAD_desync100, we have used the approach of [ZBHV20] et al. and assumed the maximum trace desynchronization to be 100. Similarly, for building the model for the datasets ASCAD_desync200 and ASCAD_desync400, we have used the same approach of [ZBHV20] et al. and assumed the maximum trace desynchronization to be 400. Finally, the first convolutional layer of both models has been replaced by dilated convolutional layer. Following [PA20], we have tuned the models for an additional set of hyperparameters: $[l_k = 16, dr = 4], [l_k = 16, dr = 6], [l_k = 32, dr = 3]$ and $[l_k = 64, dr = 2]$ where l_k and dr are the kernel width and dilation rate of the dilated convolutional layer respectively.