TransNet: Shift Invariant Transformer Network for Side Channel Analysis
(extended version)

Suvadeep Hajra[1⋆], Sayandeep Saha[2], Manaar Alam[3], and Debdeep Mukhopadhyay[1]

[1] Indian Institute of Technology Kharagpur, Kharagpur, India
{suvadeep.hajra,debdeep.mukhopadhyay}@gmail.com
[2] Nanyang Technological University, Singapore
[3] New York University Abu Dhabi, United Arab Emirates

Abstract. Deep learning (DL) has revolutionized Side Channel Analysis (SCA) in recent years. One of the major advantages of DL in the context of SCA is that it can automatically handle masking and desynchronization countermeasures, even while they are applied simultaneously for a cryptographic implementation. However, the success of the attack strongly depends on the DL model used for the attack. Traditionally, Convolutional Neural Networks (CNNs) have been utilized in this regard. This work proposes to use Transformer Network (TN) for attacking implementations secured with masking and desynchronization. Our choice is motivated by the fact that TN is good at capturing the dependencies among distant points of interest in a power trace. Furthermore, we show that TN can be made *shift-invariant* which is an important property required to handle desynchronized traces. Experimental validation on several public datasets establishes that our proposed TN-based model, called *TransNet*, outperforms the present state-of-the-art on several occasions. Specifically, TransNet outperforms the other methods by a wide margin when the traces are highly desynchronized. Additionally, TransNet shows good attack performance against implementations with desynchronized traces even when it is trained on synchronized traces. The Tensorflow implementation of TransNet is available at https://github.com/suvadeep-iitb/TransNet.

Keywords: Side channel analysis, Masking countermeasure, Transformer network

1 Introduction

Ever since its introduction in [16], SCA poses a significant threat to cryptographic implementations. To protect the cryptographic implementations from

⋆ corresponding author

those attacks, several countermeasures have been proposed. Masking countermeasures [6] and desynchronization of traces [7] are two commonly used countermeasures against those attacks. 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. On the other hand, desynchronization of the power traces causes the PoIs of the traces to be misaligned, reducing the signal-to-noise ratio (SNR) of the individual sample points. The reduced SNR causes an increase in the number of power traces required for a successful attack. Recently, DL [22, 20] has been found to be very effective against both the countermeasures. DL methods can eliminate the necessity of critical preprocessing steps in SCA while attacking desynchronized traces [5]. DL methods can also break masking countermeasures without requiring careful selection of combining function [30].

Various DL models like Feed-Forward Neural Networks (FFNs), Recurrent Neural Networks (RNNs), Convolutional Neural Networks (CNNs) have been explored [20, 3, 19] for SCA. Among those, CNNs have been widely adopted for performing profiling SCA[4] [5, 3, 43, 14, 39]. Because of the shift-invariance property of CNNs, they can perform very well on misaligned attack traces, and thus, can eliminate critical preprocessing steps like realignment of power traces in a standard SCA [5]. Moreover, the CNN-based models have achieved state-of-the-art results in many publicly available datasets [39, 35]. However, the existing CNN-based models are limited in several aspects. Firstly, we have experimentally demonstrated that their performance gets worse as the amount of desynchronization gets larger. Secondly, to perform well on desynchronized attack traces, they are required to be trained using profiling desynchronization almost the same as attack desynchronization [43, 34]. Finally, separate models are needed to be designed to attack implementations protected by different amounts of desynchronization.

Recently, in a seminal work, Vaswani et al. [33] have introduced Transformer Network (TN), which has defeated all its CNN and RNN-based counterparts by a wide margin in almost every natural language processing task. In this work, we propose to use TN for SCA. TN can easily capture the dependency between distant PoIs, and, thus, is a natural choice against implementations protected using countermeasures like masking for which the PoIs are spread across a long range in the time axis. Moreover, by introducing a weaker notion (applicable to SCA) of shift-invariance, we have shown that TN can be shift-invariant in the context of SCA. Thus, TN can be effective against misaligned traces as well. We have proposed a TN-based DL model, namely TransNet, for performing

[4] In profiling SCA, the adversary possesses a device similar to the attack device and uses that device to train a model for the target device. The trained model is used to attack the target device. A profiling SCA assumes the strongest adversary and provides the worst-case security analysis of a cryptographic device. In this work, we have considered profiling SCA only.

SCA based on the above observations. We have also experimentally evaluated TransNet against implementations protected by masking and trace desynchronization countermeasures. Our experimental results suggest that TransNet performs better than existing state-of-the-art CNN-based models on several scenarios. Specifically, TransNet performs better than the CNN-based state-of-the-art models when the attack traces are highly desynchronized. Additionally, TransNet can perform very well on highly desynchronized traces even when trained on synchronized traces – a feature that none of the existing CNN-based models exhibits (kindly refer to Section 6.6 for a detailed discussion).

In summary, the contributions of the paper are as follows:

- Firstly, we propose to use TN for SCA. TN can naturally capture long-distance dependency ([33]), thus, it is a better choice against masking countermeasure. Additionally, we have defined a weaker notion of shift-invariance that is well applicable to SCA. Under this new notion, we have mathematically shown that the TN can be made to be shift-invariant. Thus, it can be effective against misaligned traces as well.
- Our proposed TN-based model, namely TransNet, significantly differs from off-the-shelf TN models in several design choices which are crucial for its success in SCA.
- Experimentally, we have compared TransNet with the CNN-based state-of-the-art models on four datasets. Among the four datasets, two datasets contain trace desynchronization, whereas the other two do not contain any trace desynchronization. The performance of TransNet is better than or comparable to the CNN-based state-of-the-art models on the four datasets. Particularly, TransNet outperforms the other methods by a wide margin when the amount of desynchronization is very high. In those scenarios, TransNet can bring down the guessing entropy to zero using very small number of attack traces, whereas the CNN-based methods struggle to bring it down below 20.
- We have also 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 guessing entropy below 1 using only 400 traces on *ASCAD_desync100* dataset [3] 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 guessing entropy below 20 using as much as 5000 traces in the same setting.

Several recent works [38, 13] have explored different loss functions for DL-based SCA. Some other recent works have explored different training techniques [1, 26]. However, those techniques are orthogonal to our work as we aim to explore different machine learning models. Also, several recent works [23, 10, 18, 27] have introduced novel machine learning models to attack very long traces. In contrast, our proposed model is more appropriate for performing attacks on shorter traces or selected time windows of a small number of sample points.

The organization of the paper is as follows. In Section 2, we introduce the notations and briefly describe how the SCA is performed using deep learning. Section 3 describes the general architecture of TN. In Section 4, we introduce our

proposed TN-based model, namely TransNet, for SCA. Section 5 explains how a TN-based model can accumulate information from distant PoIs. The section also proves the shift-invariance property of the TransNet like models. In Section 6, we experimentally evaluate the TransNet model on several datasets. Section 7 discusses the advantages and disadvantages of the proposed model. Finally, in Section 8, we conclude.

2 Preliminaries

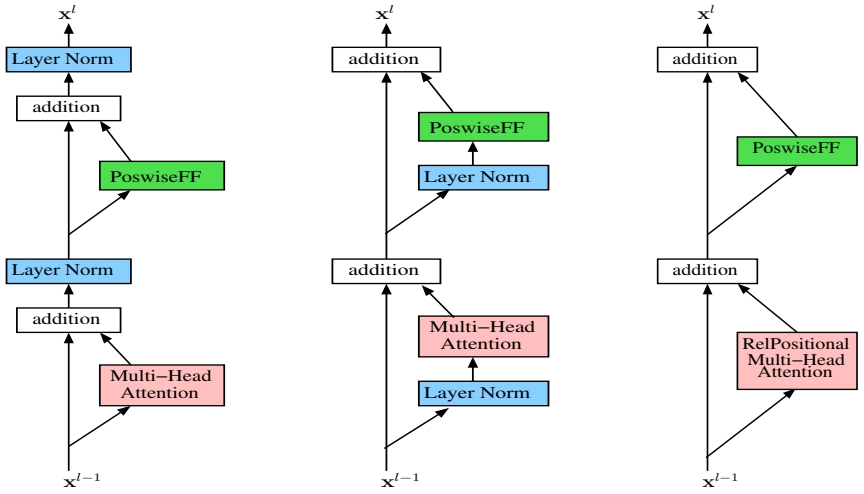
In this section, we first introduce the notations used in the paper. Then, we briefly describe how a profiling SCA is performed using DL.

2.1 Notations

Throughout the paper, we have used the following notational convention. A random variable is represented by a letter in the capital (e.g., X), whereas an instantiation of the random variable is represented by the corresponding letter in small (e.g., x) and the domain of the random variable by the corresponding calligraphic letter (e.g., $\mathcal{X}$). Similarly, a capital letter in bold (e.g., $\mathbf{X}$) is used to represent a random vector, and the corresponding small letter in bold (e.g., $\mathbf{x}$) is used to represent an instantiation of the random vector. A matrix is represented by a capital letter in roman type style (e.g., 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 is represented by $\mathrm{M}[i, j]$. $\mathbb{P}[\cdot]$ represents the probability mass/density function and $\mathbb{E}[\cdot]$ represents expectation.

2.2 Profiling SCA using Deep Learning

Like other profiling attacks, profiling attacks using deep learning are performed in two phases: profiling and attack. However, unlike other profiling attacks like template attacks, the adversary does not build any template distribution for each value of the intermediate secret in this case. Instead, he trains a deep learning model to directly predict the values of the intermediate secret from the power traces. More precisely, in the profiling phase, the adversary sets the keys of the clone device of his own choice and collects a large number of traces for different plaintexts. For each trace, he computes the value of the intermediate secret variable $Z = F(X, K)$ where X is the random plaintext, K is the key, and $F(\cdot, \cdot)$ is a cryptographic primitive. Then, the adversary trains a DL model $f : \mathbb{R}^n \mapsto \mathbb{R}^{|\mathcal{Z}|}$ using the power traces as input and the corresponding Z variables as the label or output. Thus, the output of the deep neural model for a power trace $\mathbf{l}$ can be written as $\mathbf{p} = f(\mathbf{l}; \theta^*)$ 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 possible key $k \in \mathcal{K}$ is computed as



(a) TN Layer with Post Layer Normalization

(b) TN Layer with Pre Layer Normalization

(c) Proposed TransNet Layer

Fig. 1: A Single Transformer Layer. Figures 1a and 1b show the conventional TN layers. In Figure 1c, we show a layer of the proposed TransNet. In the proposed TransNet layer, no layer normalization has been used. Additionally, the proposed TransNet layer uses relative positional encoding [8] within the self attention layer (please refer to Section 3.4 for details).

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

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, $\hat{k} = \text{argmax}_k\, \hat{\delta}_k$ is chosen as the guessed key.

Several deep neural network architectures including Feed Forward Network (FFN) [22, 21, 20], Convolutional Neural Network (CNN) [20, 5, 3, 43, 29], Recurrent Neural Network (RNN) [20, 19, 18] have been explored for profiling SCA. 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

A Transformer Network (TN) is a deep learning model which was originally developed for a sequence processing task. The TN takes a sequence $(x_0, \cdots, x_{n-1})$ as input and generates a sequence of output vectors $(\mathbf{y}_0, \cdots, \mathbf{y}_{n-1})$. The sequence of the output vectors can then be processed differently depending on the

target task. For example, for a sequence classification task (as in SCA), the mean vector of the output vectors can be used 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 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 two main modules - a multi-head self-attention layer and a position-wise feed-forward layer. Specifically, a transformer layer is a multi-head self-attention layer followed by a position-wise feed-forward layer. Additionally, to facilitate ease of training, the input and the output of each of the modules are connected by shortcut connection [11]. Sometimes layer-normalization operations [2] are also added in a transformer layer. Figure 1 shows two different variations of the conventional transformer layer along with the layer of the proposed TransNet model. The forward pass of an L-layer TN is shown in Algorithm 1.

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`
  $\mathbf{s}_0, \mathbf{s}_1, \cdots, \mathbf{s}_{n-1} \leftarrow \text{MultiHeadSelfAttn}^l\,(\mathbf{x}_0^{l-1}, \mathbf{x}_1^{l-1}, \cdots, \mathbf{x}_{n-1}^{l-1})$
  `// 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`
  $\mathbf{s}_0, \mathbf{s}_1, \cdots, \mathbf{s}_{n-1} \leftarrow \text{LayerNormalization}\,(\mathbf{s}_0, \mathbf{s}_1, \cdots, \mathbf{s}_{n-1})$
  `// apply position-wise feed-forward operation`
  $\mathbf{t}_0, \mathbf{t}_1, \cdots, \mathbf{t}_{n-1} \leftarrow \text{PoswiseFF}^l\,(\mathbf{s}_0, \mathbf{s}_1, \cdots, \mathbf{s}_{n-1})$
  `// 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`
  $\mathbf{x}_0^l, \mathbf{x}_1^l, \cdots, \mathbf{x}_{n-1}^l \leftarrow \text{LayerNormalization}\,(\mathbf{t}_0, \mathbf{t}_1, \cdots, \mathbf{t}_{n-1})$
return $(\mathbf{x}_0^L, \mathbf{x}_1^L, \cdots, \mathbf{x}_{n-1}^L)$

Each building block of the above overall architecture is described below.

3.1 Embedding Layer

A transformer layer takes a sequence of vectors as input. However, the input to a TN is generally a sequence of discrete symbols (in case of text processing) or a se-

quence of real numbers (in case of images or power traces). The embedding layer converts the sequence of discrete symbols or real numbers $(x_0, x_1, \cdots, x_{n-1})$ into the sequence of vectors $(\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_{n-1})$. Generally, $\mathbf{x}_i = f(x_i; \mathrm{E})$, $0 \leq i < n$, holds for some embedding function $f(\cdot)$ and parameter E. The parameter E is learned during training along with the other parameters of the network.

3.2 Multi-Head Self-Attention Layer

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

Self-Attention: Self-attention layer takes a sequence of input vectors $(\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_{n-1})$ as input and generates another sequence of vectors $(\mathbf{y}_0, \mathbf{y}_1, \cdots, \mathbf{y}_{n-1})$. 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 i-th output vector $\mathbf{y}_i$ is computed using the weighted sum of the input vectors where the weights are given by the attention probabilities i.e. $\mathbf{y}_i = \sum_j p_{ij}\mathbf{x}_j$. Thus, if $\mathbf{x}_i$ and $\mathbf{x}_j$ are two vectors corresponding to the leakages of two PoIs, the state $\mathbf{y}_i$ can accumulate their information in a single step even when the distance between i and j is large. Thus, this step can be useful to combine leakages of multiple PoIs of the input traces (kindly refer to Section 5.1 for a detailed discussion).

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 element to j-th element, $0 \leq j < n$, is calculated using a scaled dot product similarity measure, i.e.
$$a_{ij} = \frac{\langle \mathrm{W}_Q\mathbf{x}_i, \mathrm{W}_K\mathbf{x}_j \rangle}{\sqrt{d_k}} = \frac{\langle \mathbf{q}_i, \mathbf{k}_j \rangle}{\sqrt{d_k}} \tag{2}$$
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 element. 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.,
$$p_{ij} = softmax(a_{ij}; a_{i,0}, \cdots, a_{i,n-1}) = \frac{e^{a_{ij}}}{\sum_{k=0}^{n-1} e^{a_{ik}}} \tag{3}$$
3. The intermediate output $\bar{\mathbf{y}}_i$ is computed by taking the weighted sum of the input vectors $\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_{n-1}$. More precisely,
$$\bar{\mathbf{y}}_i = \sum_{j=0}^{n-1} p_{ij}\mathrm{W}_V\mathbf{x}_j = \sum_{j=0}^{n-1} p_{ij}\mathbf{v}_j \tag{4}$$

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 $\bar{\mathbf{y}}_i \in \mathbb{R}^{d_v}$ into $\mathbb{R}^d$ by a trainable weight matrix $\mathrm{W}_O \in \mathbb{R}^{d \times d_v}$, i.e.

$$\mathbf{y}_i = \mathrm{W}_O \bar{\mathbf{y}}_i \tag{5}$$

Thus, the self-attention operation can be written as matrix multiplication in the following way:

$$\begin{aligned} \bar{\mathrm{Y}} &= \mathit{Self\text{-}Attention}(\mathrm{W}_Q, \mathrm{W}_K, \mathrm{W}_V) = \mathit{softmax}\left(\mathrm{A}\right) \mathrm{X}\mathrm{W}_V^T = \mathrm{P}\mathrm{X}\mathrm{W}_V^T \\ \mathrm{Y} &= \bar{\mathrm{Y}}\mathrm{W}_O^T \end{aligned} \tag{6}$$

where $\bar{\mathbf{y}}_i$, $\mathbf{y}_i$ and $\mathbf{x}_i$ denote the i^{th} rows of matrices $\bar{\mathrm{Y}}$, Y and X, 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

$$\begin{aligned} \bar{\mathrm{Y}}^{(i)} &= \mathit{Self\text{-}Attention}(\mathrm{W}_Q^{(i)}, \mathrm{W}_K^{(i)}, \mathrm{W}_V^{(i)}), \text{ for } i = 0, \cdots, H-1 \\ \bar{\mathrm{Y}} &= [\bar{\mathrm{Y}}^{(0)}, \cdots, \bar{\mathrm{Y}}^{(H-1)}] \\ \mathrm{Y} &= \bar{\mathrm{Y}}\mathrm{W}_O^T \end{aligned} \tag{7}$$

where the function Self-Attention$(\cdot,\cdot,\cdot)$ is defined in Eq. 6, $[\mathrm{A}_1, \mathrm{A}_2, \cdots, \mathrm{A}_n]$ 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 captures the dependency among the elements of the input sequence in one way. An H-head self-attention layer can capture the dependency among those in H-different ways.

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 FFN$(\mathbf{x})$ be a two layer feed-forward network with ReLU activation [9] and hidden dimension d_i. Then, the output sequence $(\mathbf{y}_0, \mathbf{y}_1, \cdots, \mathbf{y}_{n-1})$ of the position-wise feed-forward layer is computed as $\mathbf{y}_i = \mathrm{FFN}(\mathbf{x}_i)$, for $i = 0, 1, \cdots, n-1$ where $(\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_{n-1})$ is the input sequence. The position-wise feed-forward layer helps to increase the non-linearity of the function represented by the TN. The integer hyper-parameter d_i is commonly referred to as inner dimension. In Table 1, we summarize the notations used to describe the transformer network.

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

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

Table 1: Notations used to denote the hyper-parameters of a transformer network

3.4 Positional Encoding

The relative positional encoding is introduced in [31]. In relative positional encoding, the attention score from i-th input element to j-th input element is made dependent on their relative position or distance $i - j$. [8] have further improved the scheme of [31]. In their proposed relative positional encoding, the attention score of Eq. 2 is modified to be computed as

$$a_{ij} = \frac{\langle \mathrm{W}_Q \mathbf{x}_i, \mathrm{W}_K \mathbf{x}_j \rangle + \langle \mathrm{W}_Q \mathbf{x}_i, \mathbf{r}_{i-j} \rangle + \langle \mathrm{W}_Q \mathbf{x}_i, \mathbf{s} \rangle + \langle \mathbf{r}_{i-j}, \mathbf{t} \rangle}{\sqrt{d_k}} \tag{8}$$

where 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. Finally, as before, the attention probabilities are computed as

$$p_{ij} = \mathrm{softmax}(a_{ij}; a_{i,0}, \cdots, a_{i,n-1}) = \frac{e^{a_{ij}}}{\sum_{k=0}^{n-1} e^{a_{ik}}} \tag{9}$$

In our TN model for SCA, we have used the relative positional encoding given by Eq. (8). In Section 5.2, we have shown that the TN with this relative positional encoding possesses shift-invariance property.

3.5 Layer Normalization

Layer normalization is commonly used in transformer layers. It is used in two ways: "post-layer normalization" (Figure 1a) and "pre-layer normalization" (Figure 1b). In the context of a SCA, we have found that using any layer normalization in the network makes the network difficult to train. We speculate that the layer normalization removes 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 (Figure 1c).

3.6 Training

The selection of a proper optimization algorithm and learning rate schedule is very crucial for the proper training of TN. To train TN, Adam optimizer [15] or some of its variants ([17, 37] are generally used. To train TN for SCA, we have used Adam optimizer.

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

Algorithm 2: Forward pass of an L-layer TransNet Architecture

begin
  `// embed input sequence using a 1D convolutional layer`
  $\mathbf{x}_0^0, \mathbf{x}_1^0 \cdots, \mathbf{x}_{n-1}^0 \leftarrow \text{Conv1D}(x_0, x_1, \cdots, x_{n-1})$
  `// optionally perform average-pooling to reduce sequence length`
  $\mathbf{x}_0^0, \mathbf{x}_1^0 \cdots, \mathbf{x}_{m-1}^0 \leftarrow \text{AvgPool}(\mathbf{x}_0^0, \mathbf{x}_1^0 \cdots, \mathbf{x}_{n-1}^0)$

for $l \leftarrow 1$ *to* L **do**
  `// apply self-attention operation`
  $\mathbf{s}_0, \mathbf{s}_1, \cdots, \mathbf{s}_{m-1} \leftarrow$
   $\text{RelPositionalMultiHeadSelfAttn}^l\,(\mathbf{x}_0^{l-1}, \mathbf{x}_1^{l-1}, \cdots, \mathbf{x}_{m-1}^{l-1})$
  `// add shortcut connection`
  $\mathbf{s}_0, \mathbf{s}_1, \cdots, \mathbf{s}_{m-1} \leftarrow \mathbf{s}_0 + \mathbf{x}_0^{l-1}, \mathbf{s}_1 + \mathbf{x}_1^{l-1}, \cdots, \mathbf{s}_{m-1} + \mathbf{x}_{m-1}^{l-1}$
  `// apply position-wise feed-forward operation`
  $\mathbf{t}_0, \mathbf{t}_1, \cdots, \mathbf{t}_{m-1} \leftarrow \text{PoswiseFF}^l\,(\mathbf{s}_0, \mathbf{s}_1, \cdots, \mathbf{s}_{m-1})$
  `// add shortcut connection`
  $\mathbf{x}_0^l, \mathbf{x}_1^l, \cdots, \mathbf{x}_{m-1}^l \leftarrow \mathbf{t}_0 + \mathbf{s}_0, \mathbf{t}_1 + \mathbf{s}_1, \cdots, \mathbf{t}_{m-1} + \mathbf{s}_{m-1}$

`// apply global average-pooling`
$\bar{\mathbf{y}} \leftarrow \text{GlobalAvgPooling}(\mathbf{x}_0^l, \mathbf{x}_1^l, \cdots, \mathbf{x}_{m-1}^l)$
`// get class-probabilities by applying a classification layer`
$p_0, p_1, \cdots, p_{C-1} \leftarrow \text{ClassificationLayer}(\bar{\mathbf{y}})$

return $(p_0, p_1, \cdots, p_{C-1})$

(called *max_learning_rate*, which is a hyper-parameter of the training algorithm). Once the maximum learning rate is reached, it is gradually decayed till the end of the training. The initial period of training epochs, in which the learning rate increases, is called *warm-up period*. To train our TN, we have used cosine decay with a linear warm-up as the learning rate scheduling algorithm (please refer to Appendix A for the details).

In the previous section, we have described the general architecture of TN. In the next section, we describe our proposed TN-based model – TransNet.

4 TransNet: A Transformer Network for SCA

TransNet is a multi-layer TN followed by a global pooling layer. The schematic diagram of TransNet is shown in Figure 2, and the forward pass is described in Algorithm 2. It uses a one-dimensional convolutional layer as an embedding layer. The convolutional layer is followed by an average-pooling layer[5] which is followed by several transformer layers. The transformer layers are followed

[5] Setting the pool size and stride of the average-pooling layer to 1, the model will behave as if there is no average-pooling layer. However, setting those values to a

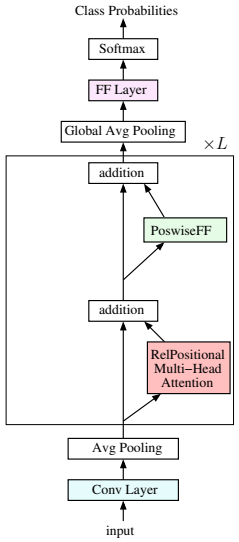


Fig. 2: Complete architecture of TransNet.

by a global pooling layer, a classification layer, and a softmax layer. Note that, unlike standard transformer layers, the TransNet layers do not use any layer normalization (Figure 1c). Moreover, in the self-attention layers, relative positional encoding as given by Eq. (8) and (9) has been used instead of more common absolute positional encoding. In Section 5.2, we have shown that the relative positional encoding scheme makes the TN shift-invariant. We trained the TransNet model using cross-entropy loss, and Adam optimizer [15]. We have used cosine decay with a linear warm-up as the learning rate scheduling algorithm (kindly refer to Appendix A for a detailed discussion).

Shift-invariance and the ability to capture the dependency among distant PoIs are two important properties to make a deep learning model effective against trace misalignments and masking countermeasures, respectively. In the next section, we explain how TransNet (in general TN) can capture the dependency among distant sample points. We also mathematically show that a TN with relative positional encoding (TransNet in particular) is shift-invariant.

5 Long Distance Dependency and Shift-Invariance

5.1 Learning Long Distance Dependency using TN

The PoIs remain spread over a long distance in power traces for many implementations. For example, in many software implementations of masking countermeasure, different shares leak at different sample points in the power traces.

larger value will make the model computationally efficient at the cost of attack efficacy and shift-invariance. Kindly refer to Section 6.8 for experimental results.

Moreover, the distance between the PoIs corresponding to different shares might be significantly large. Thus, a successful attack against those implementations requires capturing the dependency among those distant PoIs. The problem of learning dependency between distant sample points is known as the problem of learning long-distance dependency. This problem has been widely studied in the deep learning literature [12, 33]. To be able to capture the dependence between two elements of the input sequence, the signal from the two input elements should be propagated to each other by the forward and backward passes through the layers of the deep neural network. Moreover, a shorter path between the two elements makes it easier to learn their dependency, whereas a longer path makes it difficult [33]. As can be seen in Figure 3, the self-attention layer can connect any elements of the input sequence using a constant number of steps. Thus, TN is very good at capturing long-distance dependency. In fact, in [33], Vaswani et al. have argued that TN is better than both CNN and RNN in capturing long-distance dependency. This property of TN makes it a natural choice for a SCA against many cryptographic implementations.

5.2 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 SCA, the inputs are generally very noisy. In fact, in SCA, one trace is often not sufficient to predict the correct key; instead, information from multiple traces is required to extract for the 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. Thus, for SCA, the invariance property can be defined in terms of $\mathbb{P}[f(\mathbf{X})|Z]$. However, for the sake of simplicity, we define the shift-invariance property only in terms of the conditional expectation $\mathbb{E}[f(\mathbf{X})|Z]$. Thus, in the context of SCA, 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*

$$\mathbb{E}[f(\mathbf{X})|Z] = \mathbb{E}[f(T(\mathbf{X}))|Z] \tag{10}$$

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

This section shows that a single layer TN model followed by a global pooling layer is shift-invariant. Towards that goal, we define the network architecture, leakage model, and the set of shift-transformations considered for the proof.

The Transformer Model: As stated in the previous paragraph, we consider a single layer TN followed by a global pooling layer. The result can be extended for multilayer TN, albeit with some minor errors arising because of the finite

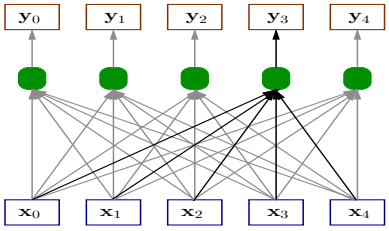


Fig. 3: Signal flow through a self-attention layer. As shown in the figure, each output position becomes dependent on all input positions after a single self-attention layer.

length of the input. Note that for the input of finite length, such errors also arise for CNN models [36]. In the rest of the section, we denote the single layer TN followed by the global pooling layer as $\text{TN}_{1\text{L}}$. The output of $\text{TN}_{1\text{L}}$ can be given by the following operations:

$$\begin{aligned}
&\mathbf{Y}_0, \cdots, \mathbf{Y}_{n-1} = \mathit{RelPositionalSelfAttention}(\mathbf{X}_0, \cdots, \mathbf{X}_{n-1}) \\
&\mathbf{U}_0, \cdots, \mathbf{U}_{n-1} = \mathbf{Y}_0 + \mathbf{X}_0, \cdots, \mathbf{Y}_{n-1} + \mathbf{X}_{n-1} \\
&\mathbf{U}'_0, \cdots, \mathbf{U}'_{n-1} = \text{FFN}(\mathbf{U}_0), \cdots, \text{FFN}(\mathbf{U}_{n-1}) \\
&\mathbf{U}''_0, \cdots, \mathbf{U}''_{n-1} = \mathbf{U}'_0 + \mathbf{U}_0, \cdots, \mathbf{U}'_{n-1} + \mathbf{U}_{n-1}
\end{aligned} \tag{11}$$

Finally the output of $\text{TN}_{1\text{L}}$ is defined as

$$\text{TN}_{1\text{L}}(\mathbf{X}_0, \cdots, \mathbf{X}_{n-1}) = \frac{1}{n} \sum_{i=0}^{n-1} \mathbf{U}''_i \tag{12}$$

where $(\mathbf{X}_0, \cdots, \mathbf{X}_{n-1})$ is the sequence of random vectors corresponding to the input of the network, $\text{TN}_{1\text{L}}(\mathbf{X}_0, \cdots, \mathbf{X}_{n-1})$ 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. $\mathit{RelPositionalSelfAttention}(\cdots)$ and $\text{FFN}(\cdot)$ respectively represent the self-attention and position-wise feed-forward operations. The description of the single layer TN is given in Figure 1c. Note that $\mathit{RelPositionalSelfAttention}(\cdots)$ is the self-attention operation implemented using relative positional encoding. In other words, the attention scores and attention probabilities in the self-attention layer are computed by Eqs. (8) and (9).

The Leakage Model: We consider the leakage model of the software implementation of a first-order masking scheme. However, the results can be easily extended for any higher-order masking scheme. Thus, 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 [19, 32] 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 two shares: M and $Z_M = Z \oplus M$) do not go out of the trace window, the range of time instances of traces considered for the attack. More precisely, we define the set of transformations $\mathcal{T}^{\text{shift}}$ as

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

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.

Next, we state Lemma 1 which will lead us to our last assumption.

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.* (8) *and* (9) *and* l *is the distance between the two PoIs.*

Thus, according to Lemma 1, the attention probabilities can be 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 our main result (Proposition 1) simple, we take the following assumption on the trained $\mathrm{TN}_{1\mathrm{L}}$ 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.* (9)*) 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 can be approximately realized in practice when we use a relative positional encoding.

With Assumption 1, 2 and 3, we summarize the main result in Proposition 1.

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

$$\mathbb{E}\left[TN_{1L}(T(\mathbf{X}_{-n+1+m_2}, \cdots, \mathbf{X}_{n-1+m_1}))|Z\right] = \mathbb{E}\left[TN_{1L}(\mathbf{X}_0, \cdots, \mathbf{X}_{n-1})|Z\right] \quad (13)$$

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.

Thus, according to Proposition 1, $\mathrm{TN_{1L}}$ is shift invariant. In Section 6.6, we have experimentally verified the shift-invariance of TransNet.

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

6 Experimental Results

In this section, we experimentally evaluate the efficacy of TransNet. In Section 6.1 to 6.3, we summarize the dataset details, hyperparameter settings with the experimental settings and the state-of-the-art-methods to which TransNet has been compared with. Section 6.4 and 6.5 compare TransNet with other state-of-the-art methods on four different datasets. In Section 6.6, we verify the shift invariance of TransNet. Finally, Section 6.7 and 6.8 study the effect of several hyper-parameters on TransNet results.

6.1 Datasets Details

For comparing TransNet with other methods, we have used the following datasets.

ASCAD: ASCAD datasets have been introduced by [3]. The original dataset is a collection of $60,000$ traces of a first-order masked implementation of AES in a software platform. 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*. 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* and $[0, 100)$ in case of *ASCAD_desync100*. Note that the random displacements have been added to both the profiling and attack traces. Each of the three derived datasets contains $50,000$ traces for profiling and $10,000$ traces for the attack. For computational efficiency, the length of each trace is reduced by keeping only 700 sample points that correspond to the interval $[45400, 46100)$ of the original traces.

We created two more datasets namely *ASCAD_desync200* and *ASCAD_desync400* using the API provided by [3]. 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 2.

	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

Table 2: Summary of the ASCAD datasets.

DPA contest v4.2: DPA contest v4.2 dataset [4] contains traces of a software implementation of AES. The implementation is protected by Rotating SBOX Masking (RSM). Following [39], we have assumed the mask to be known. Thus, the implementation behaves like an unprotected implementation.

	DPA contest v4.2	AES RD	AES HD
Profiling Dataset Size	4500	25000	50000
Attack Dataset Size	500	25000	25000
Trace Length	4000	3500	1250

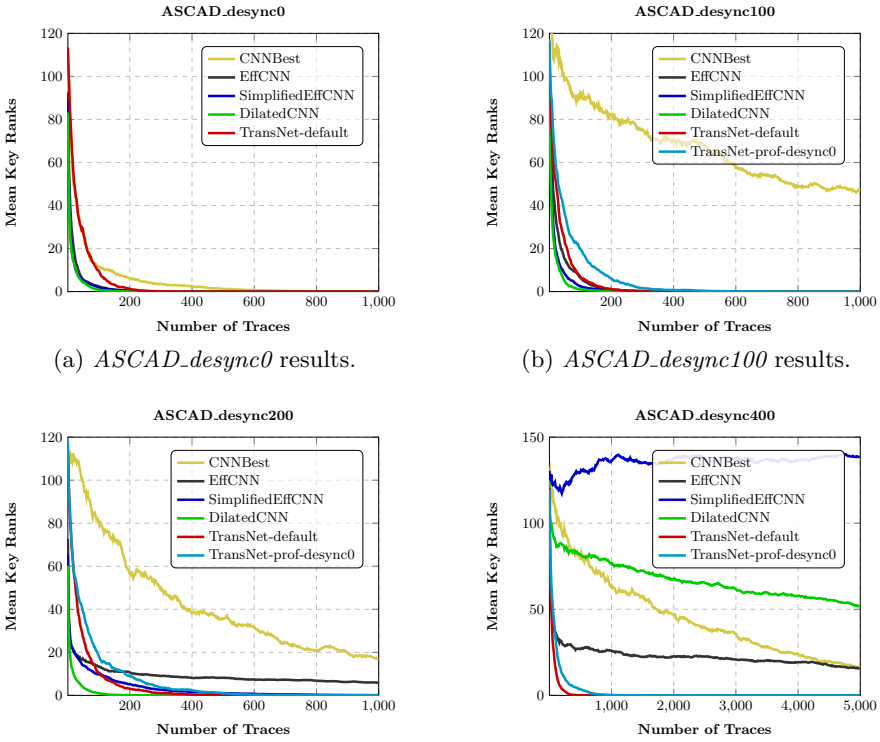
Table 3: Details of the DPA contest v4.2, AES RD and AES HD datasets.

AES RD: AES RD [7] contains the traces of a software implementation of AES protected by random delay countermeasure. The sensitive variable is taken to be the first round sbox operation. We have used the same train-test split which has been used in [39].

AES HD: AES HD [28] contains the traces of an unprotected AES implemented on FPGA. The trace window corresponds to the register update of the last round. Like *AES RD* dataset, for this dataset also, we have used the train-test split used in [39]. The statistics of the *DPA contest v4.2*, *AES RD* and *AES HD* are summarized in Table 3.

6.2 Other State-of-the-art Methods

We have compared TransNet with the following methods:



(a) *ASCAD_desync0* results. (b) *ASCAD_desync100* results.

(c) *ASCAD_desync200* results. (d) *ASCAD_desync400* results.

Fig. 4: Comparison with the CNN-based models on the *ASCAD* datasets. *TransNet-default* and *TransNet-prof-desync0* respectively denote the TransNet model trained with and without using profiling desynchronization. All the CNN-based models have been trained using profiling desync same as attack desync.

CNNBest: The *CNNBest* model has been introduced in [3]. To evaluate *CNNBest* model on *ASCAD_desync0*, *ASCAD_desync50* and *ASCAD_desync100* datasets, we have used the trained model provided by the authors in their official GitHub repository[6]. For the evaluation on *ASCAD_desync200* and *ASCAD_desync400* datasets, we have trained the model on the two datasets using their code.

EffCNN: In [39], Zaid et al. have proposed a methodology for constructing CNN-based models that are robust to trace misalignments. For comparison with TransNet, we built the models following their methodology for different datasets (kindly refer to Appendix F for the descriptions of the models).

SimplifiedEffCNN: In [35], the authors have suggested removing the first convolutional and batch normalization layer of the *EffCNN* models. These simplified

[6] https://github.com/ANSSI-FR/ASCAD.git

EffCNN models are easier to train and provide an improvement in results over *EffCNN*. Thus, we also compare *SimplifiedEffCNN* models with TransNet. The *SimplifiedEffCNN* models are constructed by removing the first convolutional layer of the *EffCNN* models.

DilatedCNN: In [24], Paguada et al. have used dilated convolutional layer to capture long distance dependency. Thus, we compare TransNet with their approach. We created two models similar to *EffCNN* models and replaced the first convolutional layer of the models with dilated convolution. We trained and tuned the *DilatedCNN* 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 the kernel width and dilation rate of the first convolutional layer.

6.3 Hyper-parameter Setting and Experimental Setup

Like in [3, 39, 18], we have used the identity leakage model. For all the experiments, we have set the number of transformer layers to 2, 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 heads H to 2. We set the kernel width of the convolutional layer to 11. For *ASCAD*, *DPA contest 4.2*, *AES RD* and *AES HD* datasets, the pool-size hyper-parameter has been set to 1, 4, 4 and 2 respectively. The pool-size hyper-parameter has been set to make the trace length less than 1000. The other hyper-parameter values have been set based on some initial experiments on the ASCAD dataset, which also worked well for the other datasets. We have also experimentally seen that TransNet performs equally well for a wide range of values of those hyper-parameters (please refer to Section 6.7 for experimental results). The complete list of hyper-parameters is given in Appendix G.

For each experiment, we trained three models using the same hyper-parameters. For each of the three models, we repeated the attack 100 times. The final mean key ranks have been generated by taking the average of the results of the total 300 attacks. Note that the mean key rank is a widely used metric in the literature to measure the success of an attack [3]. The keys are ranked from 0 to 255.

6.4 Results on ASCAD Datasets

This section compares TransNet with the other state-of-the-art methods on ASCAD datasets based on trace counts required to perform the attacks. All the CNN-based models have been trained using profiling desync, same as attack desync. On the other hand, we trained two TransNet models for each of the experiments: one using no profiling desync and the other using profiling desync same as the attack desync. We refer to the model which is trained using profiling desync as *TransNet-default* and the model which is trained using no profiling desync as *TransNet-prof-desync0*. The results are shown in Figure 4. From Figure 4a, 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 desynchronization increases slightly (Figure 4b), the performance of *CNNBest*

deteriorates drastically, indicating that it is the least shift-invariant among all the six models. As the amount of trace desynchronization gets larger further, the performance of *CNNBest* and *EffCNN* becomes inferior by a large margin compared to the other four methods (Figure 4c). The performance of the rest of the four methods is 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-default requires about 800 traces to bring it down to 0 (Figure 4d). Moreover, the *TransNet-prof-desync0* model, which has been trained on only synchronized traces, also brings 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.

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[7]. Moreover, TransNet can perform very well on highly desynchronized attack traces even when the model is trained on only synchronized traces.

In the next section, we provide experimental results on the other datasets.

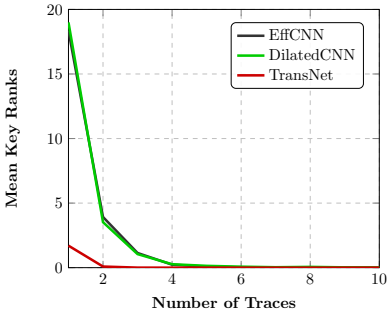


Fig. 5: Results on *DPA contest v4.2* dataset.

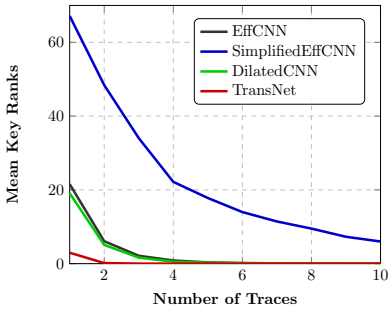


Fig. 6: Results on *AES RD* dataset.

6.5 Experimental Results on the Other Datasets

In this section, we compare TransNet with the CNN-based models on *DPA contest v4.2*, *AES RD* and *AES HD* datasets (please refer to Section 6.1 for the details of the datasets). For the experiments on *DPA contest v4.2* and *AES RD* datasets, we set the *pool_size* hyper-parameter of TransNet to 4 and for *AES HD* dataset, we set it to 2. We have tuned the hyper-parameter *train_step* though we found that setting it to 30000 works very well across all three datasets. All other hyper-parameters are kept the same as in the experiments on the ASCAD

[7] Note that the length of the power traces of the software implementations is typically in the order of $1e5$. For example, the traces of the ASCAD dataset are 100000 points long. Thus, a desync value such as 400 is possible in those traces.

dataset. For the experiments on *AES HD* dataset, we have trained TransNet as an ensemble of bit-models as proposed in [41]. In the ensemble of bit-models, each bit of the label is predicted independently. Thus, the rest of the model and the training process remain the same apart from the classification layer.

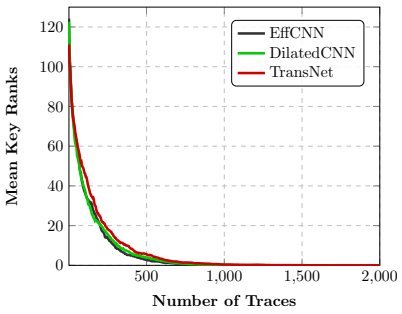


Fig. 7: Results on *AES HD* dataset.

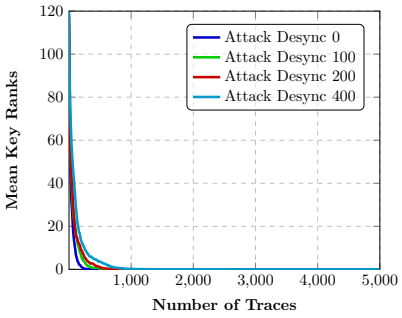


Fig. 8: Shift-invariance of *TransNet*.

The results on *DPA contest v4.2*, *AES RD* and *AES HD* datasets are respectively shown in Figures 5, 6 and 7. The figures show that TransNet performs better than the CNN-based counterparts on *DPA contest v4.2* and *AES RD* whereas slightly worse on *AES HD*. In particular, on the *DPA contest v4.2* and *AES RD*, the TransNet models bring down the mean key rank below 1 using at most two attack traces, whereas the CNN-based models require at least four for the same. Such a difference in the number of required traces may be critical for the success of the attack when the cipher is used in some leakage resilient mode [25]. For example, many existing leakage resilient modes assume 2-simulatability where at most two observations are available for a single key. A successful attack using 2 traces implies that the schemes are not secure.

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

6.6 Verifying the Shift Invariance of TransNet

In Section 5.2, we have mathematically shown that TransNet is shift-invariant. To examine whether this property persists in practice, we performed experiments on the ASCAD datasets described in Section 6.1. To evaluate the achieved shift-invariance of the TransNet models, we trained the models on synchronized traces and evaluated on desynchronized traces. Note that the length of traces of the first three 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 8.

The figure shows that the results get only slightly worse as desynchronization in the attack traces increases. Thus, it can be considered as strong evidence for achieving almost shift-invariance by the TransNet models. On the other hand, interleaving of pooling layers with the convolution layers or using of the flattening layer reduces the shift-invariance of the CNN models [36]. As a result, the existing state-of-the-art CNN models fail to perform well on desynchronized attack traces while trained on synchronized profiling traces. To verify the fact, we also trained the EffCNN models on *ASCAD_desync0* dataset and tested on *ASCAD_desync100*, *ASCAD_desync200* and *ASCAD_desync400* datasets. The results are provided in Appendix E. The results imply EffCNN fails to perform well as the attack desync gets larger when it is trained on synchronized traces. Thus, TransNet is a better choice when the amount of desynchronization in attack traces is significantly larger than that of profiling traces.

6.7 Sensitivity of TransNet to Several Hyper-parameters

The complete list of TransNet hyper-parameters can be found in Appendix G. We set most of the hyper-parameters following the standard convention used in transformer literature (please refer to Appendix G). And we found that a default value for the rest of the hyper-parameters works very well across all four datasets. Here, apart from *learning_rate* and *train_epoch*, we list the important hyper-parameters which might take a significant role for a new application:

d_model: The hyper-parameter, also has been denoted by d, represents the model dimension or the output dimension of the transformer layers.

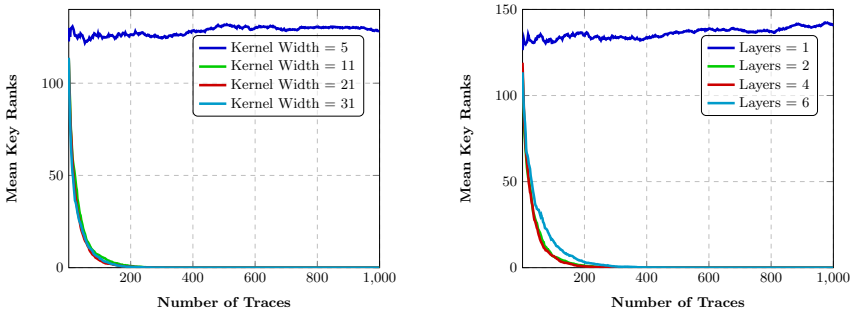
n_head: The hyper-parameter, also has been denoted by H, represents the number of heads used in the multi-head self-attention layers.

conv_kernel_size: The hyper-parameter represents the kernel width of the first convolutional layer of TransNet.

pool_size: The hyper-parameter 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 hyper-parameter, also has been denoted by L, represents the number of transformer layers in TransNet.

Among the above five hyper-parameters, *pool_size* can be used to improve the computational and memory efficiency of TransNet (please refer to Section 6.8 for a detailed discussion) and can be set to 1 for obtaining the best results. However, setting to a larger value would bring computational efficiency in exchange of a slight loss of accuracy and shift-invariance. We found that a default value of 128 for *d_model* and 2 for *n_head* works very well across all the tested datasets. In this section, we further study the sensitivity of TransNet to the other two



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

(b) Results with varying number of transformer layers.

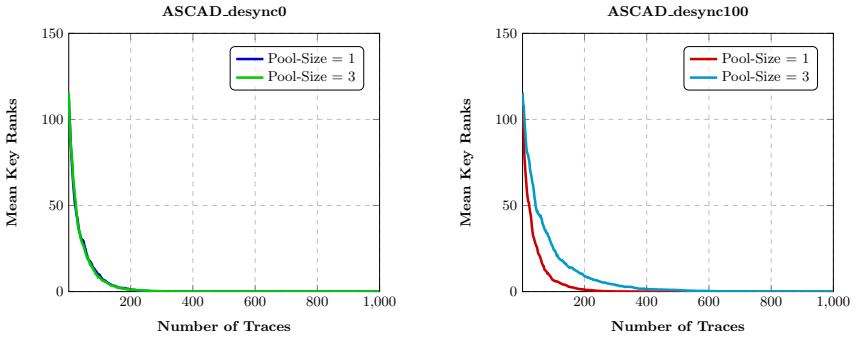
Fig. 9: Sensitivity of TransNet to kernel width and number of transformer layers. Figure (a) shows the results with varying kernel widths and Figure (b) shows with varying numbers of layers. The models have been both trained and evaluated with desync 100.

hyper-parameters namely *conv_kernel_width* and *n_layer*. The results are shown in Figure 9.

In Figure 9a, 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 9b. 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. Therefore, the hyper-parameters like *conv_kernel_width* and *n_layer* are not required to be tuned in a very fine-grained manner.

6.8 Efficiency-Efficacy Trade-off by Varying Pool Size

This section investigates the effect of different pool sizes on the TransNet results and computational time. Note that we expect the results of TransNet to be worse while gaining some computational efficiency for a larger value of pool size. Thus, we performed experiments with pool sizes 1 and 3 on *ASCAD_desync0* and *ASCAD_desync100* datasets. Figure 10a plots the results for *ASCAD_desync0* dataset. The figure shows that TransNet performs equally well for both the pool sizes. The results on *ASCAD_desync100* dataset are plotted in Figure 10b. In this case, the result with pool size 3 performs is slightly worse than that with pool size 1. Note that the deterioration of the performance of TransNet for larger pool size on *ASCAD_desync100* is well expected as the use of sub-sampling layers like the average-pooling layer reduces the shift-invariance of DL models [42]. To



(a) Results on *ASCAD_desync0*. (b) Results on em ASCAD_desync100.

Fig. 10: Effect of pool size on TransNet results. Figure (a) plots the results for *ASCAD_desync0* dataset and Figure (b) plots for *ASCAD_desync100* dataset.

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 4. 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 a pool size of 1. Thus, using a larger value of pool size provides computational efficiency.

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

Table 4: Training time for different pool sizes.

7 Discussion

Though TransNet has outperformed the CNN-based state-of-the-art models on several datasets, the quadratic time and memory complexity with respect to the trace length limit its applicability to shorter attack windows (in the order of 1000). However, the trace length can be larger in many practical scenarios. One way to extend TransNet to traces of such length is by using linear or log-linear TN. We take the exploration of those variations for SCA as future work. Instead, in the current work, we choose to demonstrate the efficacy of TN for limited length traces. Our experimental results suggest that TransNet is a better alternative to the other models in many such scenarios in terms of attack efficacy.

Our experimental results suggest that the existing CNN-based state-of-the-art models fail to perform well on highly desynchronized traces. One way to improve the performance of the CNN models in such scenarios is by replacing

the flattening layer with the global pooling layer in the models. We have compared one such model in Appendix D with TransNet on highly desynchronized *ASCAD_desync400* dataset. The results suggest that the performance of CNN models is still worse than TransNet. However, we agree that it might be possible to tune the CNN models further to improve their performance.

8 Conclusion

In this work, we have introduced TN in the context of profiling SCA. TN is good at capturing the dependency among the distant PoIs, which makes it a natural choice against many masked implementations. Moreover, we have shown that TN can also be made shift-invariant using some design choices. The shift-invariance of TN makes it highly effective against misaligned traces as well. Based on the above advantages of TN, we have proposed TransNet, a TN-based deep learning model for performing SCA. We have also experimentally evaluated the proposed model on four datasets. It is better than or comparable to other state-of-the-art methods on the four datasets. The advantage of TransNet over existing state-of-the-art methods is particularly observable when the traces are highly desynchronized. In those situations, TransNet can bring down the guessing entropy to zero using a very small number of attack traces, whereas the other methods fail to bring it down below 20. Additionally, TransNet can perform very well on highly desynchronized attack traces even when trained on synchronized profiling traces showing its low dependence on profiling desync for better training. The results suggest that TransNet provides a viable alternative to existing CNN-based models for SCA.

References

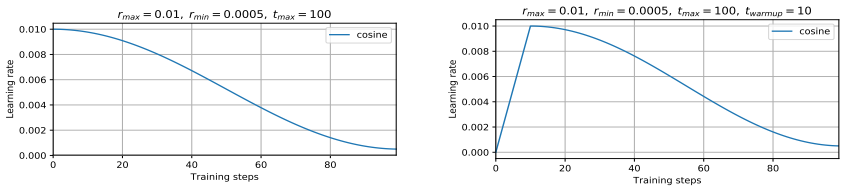
1. Abdellatif, K.M.: Mixup data augmentation for deep learning side-channel attacks. IACR Cryptol. ePrint Arch. p. 328 (2021)
2. Ba, L.J., Kiros, J.R., Hinton, G.E.: Layer normalization. CoRR **abs/1607.06450** (2016)
3. Benadjila, R., Prouff, E., Strullu, R., Cagli, E., Dumas, C.: Deep learning for side-channel analysis and introduction to ASCAD database. J. Cryptogr. Eng. **10**(2), 163–188 (2020)
4. Bhasin, S., Bruneau, N., Danger, J., Guilley, S., Najm, Z.: Analysis and improvements of the DPA contest v4 implementation. In: SPACE, India. LNCS, vol. 8804, pp. 201–218. Springer (2014)
5. Cagli, E., Dumas, C., Prouff, E.: Convolutional neural networks with data augmentation against jitter-based countermeasures - profiling attacks without pre-processing. In: CHES, Taiwan. LNCS, vol. 10529, pp. 45–68. Springer (2017)
6. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In: CRYPTO, USA. LNCS, vol. 1666, pp. 398–412. Springer (1999)
7. Coron, J., Kizhvatov, I.: An efficient method for random delay generation in embedded software. In: CHES, Switzerland. LNCS, vol. 5747, pp. 156–170. Springer (2009)

8. Dai, Z., Yang, Z., Yang, Y., Carbonell, J.G., Le, Q.V., Salakhutdinov, R.: Transformer-XL: Attentive language models beyond a fixed-length context. In: ACL, Italy, Volume 1. pp. 2978–2988. ACL (2019)
9. Glorot, X., Bordes, A., Bengio, Y.: Deep sparse rectifier neural networks. In: AISTATS, USA. JMLR Proceedings, vol. 15, pp. 315–323. JMLR.org (2011)
10. Gohr, A., Jacob, S., Schindler, W.: Subsampling and knowledge distillation on adversarial examples: New techniques for deep learning based side channel evaluations. In: Dunkelman, O., Jr., M.J.J., O'Flynn, C. (eds.) SAC, Canada. LNCS, vol. 12804, pp. 567–592. Springer (2020)
11. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: CVPR, USA. pp. 770–778. IEEE Computer Society (2016)
12. Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J.: Gradient flow in recurrent nets: The difficulty of learning long-term dependencies (2001)
13. Kerkhof, M., Wu, L., Perin, G., Picek, S.: Focus is key to success: A focal loss function for deep learning-based side-channel analysis. In: Balasch, J., O'Flynn, C. (eds.) COSADE, Belgium. LNCS, vol. 13211, pp. 29–48. Springer (2022)
14. Kim, J., Picek, S., Heuser, A., Bhasin, S., Hanjalic, A.: Make some noise. unleashing the power of convolutional neural networks for profiled side-channel analysis. TCHES **2019**(3), 148–179 (2019)
15. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. In: ICLR, USA, (2015)
16. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: CRYPTO, USA. LNCS, vol. 1666, pp. 388–397. Springer (1999)
17. Loshchilov, I., Hutter, F.: Decoupled weight decay regularization. In: ICLR, USA. OpenReview.net (2019)
18. Lu, X., Zhang, C., Cao, P., Gu, D., Lu, H.: Pay attention to raw traces: A deep learning architecture for end-to-end profiling attacks. TCHES **2021**(3), 235–274 (2021)
19. Maghrebi, H.: Deep learning based side channel attacks in practice. IACR Cryptol. ePrint Arch. **2019**, 578 (2019)
20. Maghrebi, H., Portigliatti, T., Prouff, E.: Breaking cryptographic implementations using deep learning techniques. In: SPACE, India. LNCS, vol. 10076, pp. 3–26. Springer (2016)
21. Martinasek, Z., Hajny, J., Malina, L.: Optimization of power analysis using neural network. In: CARDIS, Germany. LNCS, vol. 8419, pp. 94–107. Springer (2013)
22. Martinasek, Z., Zeman, V.: Innovative method of the power analysis. Radioengineering **22**(2), 586–594 (2013)
23. Masure, L., Belleville, N., Cagli, E., Cornelie, M., Couroussé, D., Dumas, C., Maingault, L.: Deep learning side-channel analysis on large-scale traces - A case study on a polymorphic AES. In: Chen, L., Li, N., Liang, K., Schneider, S.A. (eds.) ESORICS, UK. LNCS, vol. 12308, pp. 440–460. Springer (2020)
24. Paguada, S., Armendariz, I.: The forgotten hyperparameter: - introducing dilated convolution for boosting cnn-based side-channel attacks. In: ACNS Satellite Workshops, Italy. LNCS, vol. 12418, pp. 217–236. Springer (2020)
25. Pereira, O., Standaert, F., Vivek, S.: Leakage-resilient authentication and encryption from symmetric cryptographic primitives. In: Ray, I., Li, N., Kruegel, C. (eds.) ACM SIGSAC, USA. pp. 96–108. ACM (2015)
26. Perin, G., Chmielewski, L., Picek, S.: Strength in numbers: Improving generalization with ensembles in machine learning-based profiled side-channel analysis. TCHES **2020**(4), 337–364 (2020)

27. Perin, G., Wu, L., Picek, S.: Exploring feature selection scenarios for deep learning-based side-channel analysis. IACR Cryptol. ePrint Arch. p. 1414 (2021)
28. Picek, S., Heuser, A., Jovic, A., Bhasin, S., Regazzoni, F.: The curse of class imbalance and conflicting metrics with machine learning for side-channel evaluations. TCHES **2019**(1), 209–237 (2019)
29. Picek, S., Samiotis, I.P., Kim, J., Heuser, A., Bhasin, S., Legay, A.: On the performance of convolutional neural networks for side-channel analysis. In: SPACE, India. LNCS, vol. 11348, pp. 157–176. Springer (2018)
30. Prouff, E., Rivain, M., Bevan, R.: Statistical analysis of second order differential power analysis. IACR Cryptol. ePrint Arch. p. 646 (2010)
31. Shaw, P., Uszkoreit, J., Vaswani, A.: Self-attention with relative position representations. In: NAACL-HLT, USA, Volume 2. pp. 464–468. ACL (2018)
32. Thapar, D., Alam, M., Mukhopadhyay, D.: Transca: Cross-family profiled side-channel attacks using transfer learning on deep neural networks. IACR Cryptol. ePrint Arch. **2020**, 1258 (2020)
33. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. In: NIPS, USA. pp. 5998–6008 (2017)
34. Won, Y., Hou, X., Jap, D., Breier, J., Bhasin, S.: Back to the basics: Seamless integration of side-channel pre-processing in deep neural networks. IACR Cryptol. ePrint Arch. **2020**, 1134 (2020)
35. Wouters, L., Arribas, V., Gierlichs, B., Preneel, B.: Revisiting a methodology for efficient CNN architectures in profiling attacks. TCHES **2020**(3), 147–168 (2020)
36. Yarotsky, D.: Universal approximations of invariant maps by neural networks. CoRR **abs/1804.10306** (2018)
37. You, Y., Li, J., Reddi, S.J., Hseu, J., Kumar, S., Bhojanapalli, S., Song, X., Demmel, J., Keutzer, K., Hsieh, C.: Large batch optimization for deep learning: Training BERT in 76 minutes. In: ICLR, Ethiopia. OpenReview.net (2020)
38. Zaid, G., Bossuet, L., Dassance, F., Habrard, A., Venelli, A.: Ranking loss: Maximizing the success rate in deep learning side-channel analysis. TCHES **2021**(1), 25–55 (2021)
39. Zaid, G., Bossuet, L., Habrard, A., Venelli, A.: Methodology for efficient CNN architectures in profiling attacks. TCHES **2020**(1), 1–36 (2020)
40. Zhang, A., Lipton, Z.C., Li, M., Smola, A.J.: Dive into Deep Learning (2020), https://d2l.ai
41. Zhang, L., Xing, X., Fan, J., Wang, Z., Wang, S.: Multi-label deep learning based side channel attack. In: AsianHOST, China. pp. 1–6. IEEE (2019)
42. Zhang, R.: Making convolutional networks shift-invariant again. In: ICML, USA. Proceedings of Machine Learning Research, vol. 97, pp. 7324–7334. PMLR (2019)
43. Zhou, Y., Standaert, F.: 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 Learning Rate Schedule

The learning rate schedule plays a vital role in the proper training of DL models. In literature, there exist several commonly used learning rate schedules [40]. In this work, we have used a cosine decay learning rate schedule. In cosine decay schedule, starting from a maximum value, the learning rate is decreased following



(a) Without any warm-up steps (b) With linear warm-up for 10 steps

Fig. 11: Plots of learning rates against training steps using cosine decay scheduling algorithm. Each step corresponds to the processing of one batch of training examples.

a cosine curve. Thus, at t-th step, the learning rate r_t is computed as

$$r_t = r_{\min} + \frac{r_{\max} - r_{\min}}{2} \times (1 + \cos(\pi t / t_{\max})) \tag{14}$$

where $r_{\max}$, $r_{\min}$ are respectively the maximum and minimum value of learning rate; and $t_{\max}$ is the total number of training steps. An example of the cosine decay learning rate schedule is plotted in Figure 11a.

For the training of large neural network models, several initial training steps are kept as warm-up steps during which the learning rate is warmed up to a desired high value starting from a small value [40]. 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 11b plots the evolution of the learning rate in cosine decay with the linear warm-up schedule.

B Proof of Lemma 1

The attention probabilities in the self-attention layer of $\text{TN}_{1\text{L}}$ is calculated following Eqs. (8) and (9). If we set 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 in Eq. (8) and Eq. (9), we have p_{ij} equals to $\frac{e^c}{e^c+n-1}$ if $j = i + l$ and $\frac{1}{e^c+n-1}$ otherwise for $0 \leq i < n - l$. 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.

C Proof of Proposition 1

From the Eqs (11), we have $\mathbf{U}_i = \mathbf{Y}_i + \mathbf{X}_i$, $\mathbf{U}''_i = \text{FFN}(\mathbf{U}_i) + \mathbf{U}_i$, for $i = 0, \cdots, n-1$ where $\mathbf{Y}_0, \mathbf{Y}_1, \cdots, \mathbf{Y}_{n-1} = \mathit{RelPositionalSelfAttention}(\mathbf{X}_0, \mathbf{X}_1, \cdots, \mathbf{X}_{n-1})$. And the output of $\text{TN}_{1\text{L}}$ is given by $\text{TN}_{1\text{L}}(\mathbf{X}_0, \cdots, \mathbf{X}_{n-1}) = \frac{1}{n}\sum_{i=0}^{n-1} \mathbf{U}''_i$.
From Eq. (4) and (5), we get $\mathbf{Y}_j = \text{W}_O \left(\sum_{k=0}^{n-1} P_{jk} \text{W}_V \mathbf{X}_k \right)$. Thus, we can write $\mathbf{Y}_{m_1}$ (where m_1 is defined in Assumption 1) as

$$\mathbf{Y}_{m_1} = \text{W}_O \left(\sum_{k=0}^{n-1} P_{m_1 k} \text{W}_V \mathbf{X}_k \right) = \text{W}_O \text{W}_V \mathbf{X}_{m_1+l}, \text{ and thus} \tag{a2}$$

$$\mathbf{U}_{m_1} = \text{W}_O \text{W}_V \mathbf{X}_{m_1+l} + \mathbf{X}_{m_1} \tag{a3}$$

Eq. (a2) 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

$$\mathbf{Y}_i = \text{W}_O \left(\sum_{k=0}^{n-1} P_{ik} \text{W}_V \mathbf{X}_k \right) = \text{W}_O \text{W}_V \mathbf{X}_{i+l}, \text{ and thus} \tag{a4}$$

$$\mathbf{U}_i = \text{W}_O \text{W}_V \mathbf{X}_{i+l} + \mathbf{X}_i \tag{a5}$$

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

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

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 $i = 0, \cdots, n-1$.

$$\mathbf{U}''_i = FFN(\mathbf{U}_i) + \mathbf{U}_i \tag{a6}$$

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 $\text{TN}_{1\text{L}}$ can be written as

$$\text{TN}_{1\text{L}}(\mathbf{X}_0, \cdots, \mathbf{X}_{n-1}) = \frac{1}{n} \sum_{i=0}^{n} \mathbf{U}''_i = \frac{1}{n} \mathbf{U}''_{m_1} + \frac{1}{n} \sum_{0 \leq i < n-l, i \neq m_1} \mathbf{U}''_i + \frac{1}{n} \sum_{n-l \leq i < n} \mathbf{U}''_i \tag{a7}$$

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

$$\mathbb{E}[\text{TN}_{1\text{L}}(\mathbf{X}_0, \cdots, \mathbf{X}_{n-1})|Z] = \frac{1}{n} \mathbb{E}[\mathbf{U}''_{m_1}|Z] + \frac{1}{n} \sum_{n-l \leq i < n} \mathbb{E}[\mathbf{U}''_i|Z] + \frac{1}{n} \sum_{0 \leq i < n-l, i \neq m_1} \mathbb{E}[\mathbf{U}''_i] \tag{a8}$$

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, we compute

$$\begin{aligned} &\mathbb{E}\left[\text{TN}_{1\text{L}}(T^s(\mathbf{X}_{-n+1+m_2}, \cdots, \mathbf{X}_{n-1+m_1}))|Z\right] \\ &= \mathbb{E}\left[\text{TN}_{1\text{L}}(\mathbf{X}_{-s}, \cdots, \mathbf{X}_{n-1-s})|Z\right] \\ &= \frac{1}{n} \mathbb{E}[\mathbf{U}''_{m_1}|Z] + \frac{1}{n} \sum_{n-l-s \leq i < n-s} \mathbb{E}[\mathbf{U}''_i|Z] + \frac{1}{n} \sum_{-s \leq i < n-l-s, i \neq m_1} \mathbb{E}\left[\mathbf{U}''_i\right] \end{aligned} \tag{a9}$$

From Assumption 2, we get

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

Thus, comparing the right hand side of Eq. (a8) and Eq. (a9) we have

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

which completes the proof.

D Comparison with CNN using Global Pooling Model

The state-of-the-art CNN models use a flattening layer after all the convolutional model to convert the two-dimensional feature representation into a one-dimensional feature representation. However, the use of a flattening layer reduces the shift-invariance of the CNN models resulting in their poor performance on highly desynchronized traces (ref. Figure 4d). This section compares TransNet to a CNN model that uses global pooling instead of the flattening layer. For this purpose, we have used the same model as EffCNN (desync400) except for the flattening layer replaced by the global pooling layer. We refer to the resulting model as *EffCNN+GlobalPooling*. The results of *EffCNN+GlobalPooling* on highly desynchronized *ASCAD_desync0* dataset is compared with that of TransNet in Figure 12. The results suggest that TransNet performs significantly better than *EffCNN+GlobalPooling*.

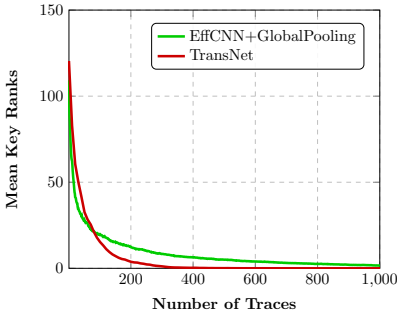


Fig. 12: Comparison of TransNet with *EffCNN+GlobalPooling* on the *ASCAD_desync400* datasets.

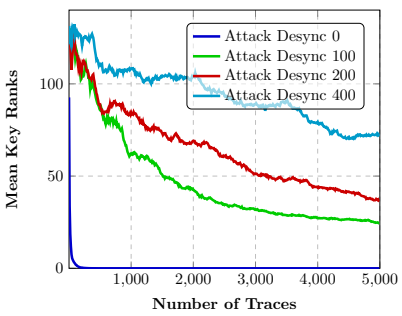


Fig. 13: Results of *EffCNN*. The models have been trained with profiling desync 0.

E Sensitivity of EffCNN to Profiling Desynchronization

As the experiments of TransNet in Section 6.6, we verify the robustness of EffCNN training to the amount of profiling desync. To verify that, we trained

the EffCNN models using only synchronized traces and tested them on desynchronized traces. The results are shown in Figure 13. From the figure, it can be seen that as the amount of desynchronization in the attack traces increases, the performance of the models gets worse rapidly, suggesting the superiority of TransNet over EffCNN when the profiling desync is significantly less than the attack desync.

F EffCNN Model Architectures for ASCAD_desync200 and ASCAD_desync400 Datasets

The EffCNN models for *ASCAD_desync0*, *ASCAD_desync100*, *DPA contest v4.2*, *AES HD*, and *AES RD* are taken from their GitHub repository[8]. We constructed the models for *ASCAD_desync200* and *ASCAD_desync400* datasets using their methodology [39]. They are given in Table 5.

Block	**Layer**	**desync200**	**desync400**
1	Conv	kw = 1, nf = 32	kw = 1, nf = 32
	SeLU		
	BatchNorm		
	Avg pooling	ps = 2	ps = 2
2	Conv	kw = 100, nf = 64	kw = 200, nf = 64
	SeLU		
	BatchNorm		
	Avg pooling	ps = 100	ps = 200
3	Conv	kw = 3, nf = 128	kw = 3, nf = 128
	SeLU		
	BatchNorm		
	Avg pooling	ps = 2	ps = 1
5	Flattening		
6	FC	d_out = 25	d_out = 30
	SeLU		
7	FC	d_out = 25	d_out = 30
	SeLU		
7	FC	d_out = 25	d_out = 30
	SeLU		
	FC	d_out = 256	d_out = 256
	Softmax layer		

Table 5: The architecture of the EffCNN models for *ASCAD_desync200* and *ASCAD_desync400* datasets. *kw* and *nf* denote the kernel width and the number of filters of the convolutional layers. Similarly, *ps* denotes the pool size and strides of the average pooling layers. And *d_out* denotes the output dimension of the fully connected layers.

[8] https://github.com/gabzai/Methodology-for-efficient-CNN-architectures-in-SCA

G Hyper-parameter Setting

We set several hyper-parameters of TransNet using the standard conventions 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 commonly 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 a large value like 16. Considering the simplicity of the problem in SCA, we set this hyper-parameter to 2. The input length n is set to be equal to the trace length. The relative positional encoding takes one hyper-parameter named *clamp_len*. It is enough to set this hyper-parameter to be equal to n. We found that setting the two dropout related hyper-parameters namely *dropout* and *dropatt* to a value of 0.05 or 0.1 works equally well.

Complete list of hyper-parameters used for training TransNet is given in Table 6.

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

Table 6: Hyperparameter setting of TransNet

Note that we set some hyper-parameters like *init*, *init_std*, *max_learning_rate*, *clip*, *min_lr_ratio* and *warmup_step* to the default value used in the implementation of [8]. For the other datasets, we use the same hyper-parameter settings except the *pool_size* and *train_steps*. The *train_steps* hyper-parameter is set to

30000 for all the other datasets. The *pool_size* is set to 2, 4 and 4 for the dataset *AES HD*, *DPA contest v4.2* and *AES RD* respectively.