IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2020, No. 4, pp. 49–85. DOI:10.13154/tches.v2020.i4.49-85

Plaintext: A Missing Feature for Enhancing the Power of Deep Learning in Side-Channel Analysis?

Breaking multiple layers of side-channel countermeasures

Anh-Tuan Hoang[1], Neil Hanley[1], Maire O'Neill[1]

Centre for Secure Information Technologies (CSIT), ECIT, Queen's University Belfast, United Kingdom at.hoang@qub.ac.uk,n.hanley@qub.ac.uk,m.oneill@ecit.qub.ac.uk

Abstract. Deep learning (DL) has proven to be very effective for image recognition tasks, with a large body of research on various model architectures for object classification. Straight-forward application of DL to side-channel analysis (SCA) has already shown promising success, with experimentation on open-source variable key datasets showing that secret keys can be revealed with 100s traces even in the presence of countermeasures. This paper aims to further improve the application of DL for SCA, by enhancing the power of DL when targeting the secret key of cryptographic algorithms when protected with SCA countermeasures. We propose a new model, *CNN-based model with Plaintext feature extension (CNNP)* together with multiple *convolutional* filter kernel sizes and structures with deeper and narrower neural networks, which has empirically proven its effectiveness by outperforming reference profiling attack methods such as template attacks (TAs), convolutional neural networks (CNNs) and multilayer perceptron (MLP) models. Our model generates state-of-the art results when attacking the ASCAD variable-key database, which has a restricted number of training traces per key, recovering the key within 40 attack traces in comparison with order of 100s traces required by straightforward machine learning (ML) application. During the profiling stage an attacker needs no additional knowledge on the implementation, such as the masking scheme or random mask values, only the ability to record the power consumption or electromagnetic field traces, plaintext/ciphertext and the key. Additionally, no heuristic pre-processing is required in order to break the high-order masking countermeasures of the target implementation.

Keywords: Deep learning · CNN · AES · Plaintext feature extension · integer encoding · one-hot encoding · key reveal · higher-order masking

1 Introduction

Since side-channel analysis (SCA) was introduced in 1996 [Koc96] based on the difference in power consumption of bit transitions, much research has been conducted on efficient methods to both break and protect cryptographic implementations. Common attack methods such as differential power analysis (DPA) [KJJ99], correlation power analysis (CPA) [BCO04], or differential frequency-based analysis (DFA) [GHT05], allow divide and conquer strategies to significantly reduce the computational complexity of key recovery when additional power (or electromagnetic) information is available. For example, in the case of the Advanced Encryption Standard (AES)-128, it is reduced from $\mathcal{O}\left(2^{128}\right)$ to $\mathcal{O}\left(16 \times 2^{8}\right)$. In order to protect against such attacks a number of countermeasures have been proposed, many of which are now standard in commercial security products such as credit cards *etc.* At

Received: 2020-04-15 Accepted: 2020-06-15 Published: 2020-08-26

the hardware layer techniques such as dual-rail logic [TAV02, TV04, PM05, CZ06, HF12] attempt to equalise the power consumption of the underlying algorithm regardless of the data being processed, while at the algorithmic layer techniques such as masking [GT02, GM11, NSGD12, Cor14, RBN$^+$15] introduce fresh randomness to reduce the useful leakage available to an attacker. Both these techniques, as well as all other countermeasures, come with various trade-offs for the level of protection provided in terms of execution time, randomness required, silicon (or memory) size *etc.*

While statistical and machine learning (ML) have a long history, the recent progress in deep learning (DL) in particular, has led to such techniques being applied in the SCA context for key recovery in the presence of countermeasures. These attacks fall under the profiling adversarial model, where it is assumed that the attacker has a similar (or identical) training device(s) to measure a large quantity of traces in order to build an accurate power model, which then allows key recovery from the target device in relatively few traces. Among the DL approaches, convolutional neural network (CNN) based models seem most promising [MPP16, PSK$^+$18, Tim19, WPB19] due to their effectiveness when training with raw data, with the *convolutional* layer acting as a filter to pick out the relevant features for classification.

1.1 Related Work

There are a large number of available ML and DL algorithms and models, as well as more general statistical learning techniques that can be applied to SCA. Some initial DL based SCA attacks have been given by [LBM15, GHO15, MZVT16]. However, these attacks are based on an assumption that the number of masks are either very limited, or an adversary is able to fully access the internal values of the target devices when profiling, including the values of the random masks, which is generally not feasible in practice. Research into the performance of many algorithms in the context of SCA has been conducted, including support vector machines (SVMs), random forests (RFs), auto-encoders, DL, template attacks (TAs) (which are equivalent to linear discriminant analysis (LDA) or quadratic discriminant analysis (QDA) depending on how the covariance matrix is generated). In [LPB$^+$15], TAs, SVMs, and RFs are applied to various leakage datasets, concluding that TAs outperform ML-based attacks when the data dimension is low while ML-based methods have an advantage when the noise level of the leakage traces increases, or where only limited profiling of the device is possible. In [MPP16], the potential of efficient DL based side channel key recovery attacks is highlighted, showing how DL-based auto-encoder models outperform other ML and TA models when attacking either unprotected or masked cryptographic implementations. However, not all research points to DL superiority, with [PSK$^+$18] claiming that simpler methods like RF outperform CNN based DL models in some cases. The effectiveness of using CNN models is shown in [WPB19], with no dimensionality reduction required when attacking implementations of public-key cryptography when compared to other profiling attack algorithms such as SVM, TA or RF, while [PEC19] highlights the importance of L2 normalization for automated selection of points of interest (PoI). In efforts to increase the effectiveness of DL performance in the context of SCA, an in-depth analysis in [PSB$^+$18, Mag19] noted that the size of filter in a CNN model is an important factor as its length should cover the most interesting PoI to enable the combination of the corresponding leakage. However, the required length can vary where a designer inserts fake computational operations into the design. In [KPH$^+$19], adding artificial noise to the source traces was found to greatly reduce over-fitting the model the training set, improving classification accuracy.

In terms of design assessments and tools for analysis, the deep learning leakage assessment (DL-LA) framework was proposed in [WMM19] to determine whether an attacker is able to extract information from side channel measurements, while [vdVP19] proposes a tool to enable users understand how a change in experimental inputs influences the performance of ML. A gradient visualization tool is presented in [MDP19], showing temporal moments where sensitive information leaks.

A non-profiled DL attack approach was proposed in [Tim19]. The secret key is revealed by combining key guesses with an analysis of DL accuracy/loss metrics, with the intuition that these will be highest/lowest, for the correct key. Although this method requires training a DL model for each hypothesis key hence is computationally intense, it was successful in attacking sensitive data protected by multiple masks.

A state of the art effort in including SCA domain knowledge into DL architectures was given by [HGG18], in which the plaintext was given as an additional input to increase the accuracy when directly training on the key value. While this paper shares a similar approach to input domain knowledge, there is a significant difference with regards to architecture aspects such as the length of PoI and network structure, affecting how the domain knowledge and PoI leakage spread through the network layers, enabling our model to work against cryptographic implementations with complex high-order countermeasures. Differing from [HGG18], who developed a straightforward CNN model to deal with unprotected AES implementations, and implementations using shift-based countermeasures, this work develops a DL model to attack an AES implementation with high-order masking and other SCA countermeasures. Beside the *plaintext extension*, which is similar with [HGG18], our proposed CNNP models are developed with small *convolutional* filter kernel sizes (referred as PoI size, which determines the number of features combined per CNN layer), multiple PoI sizes and multiple *convolutional* structures in combination, before being fed into a deep and narrow neural structure.

Three considerations are taken into our model design:

- Interesting PoI may be at differing distances in relation to each other, hence multiple feature size extraction and combination lengths are used to better account for this variability.
- Including specific features such as the plaintext or ciphertext will increase the attacking accuracy. In addition, encoding methods for plaintext feature extension, such as integer and one-hot, are investigated and evaluated.
- Network structure (deeper and wider network) affects the accuracy with the same number of parameters [BC14, AJR+18], in which deeper networks (which incorporate more layers for better feature embedding) provides better accuracy compared to wider networks (increased number of neurons with larger number of feature maps per layer).

Given these considerations, we built a complex model with multiple *convolutional* filter kernel sizes working together such that the most interesting PoI are covered during the combination of the side-channel leakage [PSB+18, Mag19], and additional features such as the plaintext are provided as an extra input. Our network is also deeper but narrower than that of VGG16 used in [PSB+18]. We label the traces using the output values of the SBox in round 1 so that the model can be used even where only a limited number of key values are available to be profiled.

1.2 Our Contributions

This paper shows the limitations of straight-forward application of available ML models for SCA in using traces from a protected AES implementation as a case study. We propose a number of CNN models that look to enhance DL from a side-channel aspect, taking domain knowledge into consideration. Our proposed models allow key recovery when targeting an AES implementation protected with higher-order masking, outperforming in terms of required traces for key recovery (40 on the ASCAD variable key database) approaches based on TAs, MLs or current CNN models applied to the same dataset. Additionally fewer epochs are required during the training stage leading to lower computational requirements. Our main contributions include:

1. Showing the advantage of using the plaintext as a feature in straight forward application of available DL models for SCA. Omitting this feature significantly reduces the effectiveness of ML models in attacking AES implementations with countermeasure methods applied.

2. Comparison of two different methods, absolute value and one-hot encoding, for including plaintext data to the model to enhance its accuracy.

3. Introduction of a new CNN model architecture for SCA, a **CNN**-based model with **P**laintext feature extension that we denote CNNP, as well as multiple parallel feature extraction and combination *convolutional* layers, and a new deep and narrow network structure. While CNNP is introduced in this paper targeting the AES algorithm as an example, the approach is generic to other targets. Additionally this model architecture is not tailored to any specific AES implementation, with the fact that designers can always apply additional countermeasures such as random delays, d^{th}-order masking or secret sharing are also taken into consideration.

4. Reducing the requirements of applying DL on SCA, particularly when attacking implementations with countermeasures applied, in which thousands of traces can be required to break higher-order protected AES implementations using DL. This work allows for key recovery with only 40 traces attack on the same database.

5. Discussion around hyper-parameter selection, such as *convolutional* layer size, multiple filter kernel size, plaintext encoding mechanism, and their effect on the CNNP model.

Evaluation and verification of the robustness of our proposed CNNP models are carried out through experimental analysis on the open source ASCAD databases.

2 Background

2.1 SCA and Countermeasures

2.1.1 SCA Approaches

SCAs work on the principal that the power consumed by a device is dependent on the operation being performed and the data being processed. This allows an adversary to estimate what the power consumption should be for some intermediate value that is a function of some known data (*e.g.* a plaintext or ciphertext byte), and some unknown data (*e.g.* a secret key byte). In a non-profiled attack, a statistical distinguisher (*e.g.* students t-test) can then be used between the actual power traces, and the estimated traces in order to determine if the secret key hypotheses is correct. In order to improve the effectiveness of the distinguisher, a leakage model is generally first applied to the intermediate value. For a profiled attack, as in this paper, it is assumed that an attacker has a similar or identical device that he can record a large number of acquisitions on, allowing an accurate profiling of the actual power leakage of the device.

While for non-profiling attacks a leakage model such as based on the Hamming weight of a value can often provide a sufficently accurate approximation for successful attacks, the value leakage model is used in this paper. This simply considers that the value itself leaks information to the side-channel such that different values consume different amounts of power. This model is widely utilised for profiling attacks, as when labelling the traces by some intermediate value for training, its allows the model to learn a broader range of features at different points in time, while a power model such as based on the Hamming weight will only allow the model to learn features at the point the value is processed. A greater number of training traces are required however, as the number of unique labels is 2^m as for an m-bit word. If the distribution of the target value is close to random however, the classes will be balanced leading to more straightforward training methods.

2.1.2 SCA Countermeasures

While a number of SCA countermeasures in both hardware and software have been proposed in the literature (such as dual-rail logic, dummy operations, threshold implementations *etc.*), here we focus on masking as this is the countermeasure that is applied to the traces under consideration. Masking (including higher-order variants) apply one or several random masks to the sensitive data such as the input of the S-Box or the S-Box itself, respectively forming 1st-order or higher-order masking schemes [RBN+15]. Masking methods can be divided into two types: additive and multiplicative, and the sensitive data can be protected with one or several independent random or fixed masks. There is a trade-off to be determined in relation to the protection required against the additional processing and randomness requirements of the masking scheme. It is relatively straightforward to mask data for linear operations, however non-linear functions can be trickier to implement securely.

A large number of masking techniques to protect AES have been proposed, such as a rotating masking scheme introduced in [NSGD12] which frequently changes the masked S-Box among 16 pre-computed ones, the algorithm introduced in [Cor14] for masking a lookup table of block-cipher at any order, or the secret sharing scheme approach of [GM11] to name but a few. In our experiment, the target implementation has a higher-order masking scheme as described in [PSB+18], in which the plaintext and S-Box are masked by two independent masks as shown in Equations 1 and 2.

Higher-order masking for AES implementations can be seen as masking of plaintext by:

$$\overline{p_i} = p_i \oplus m_i \tag{1}$$

and masking the S-Box for value $i \in [0 \ldots 255]$, which can be prepared in advance:

$$\overline{\text{S-Box}\,(x)} = \text{S-Box}\,(x \oplus m_{i,in}) \oplus m_{i,out} \tag{2}$$

in which p and m are plaintext and mask respectively in Equation 1, and x is the masked input for S-Box, $m_{i,in}$ and $m_{i,out}$ the input/output masks in Equation 2. Equation 1 ensures that the linear AddRoundKey operation which follows the S-Box works as expected on the masked data, but no unmasked data is processed. Equation 2 ensures that the non-linear S-Box operation will be masked by the unknown pair values $m_{i,in}$ and $m_{i,out}$ so that on every execution different data will be processed regardless of the input value.

2.1.3 The Role of Plaintext

It is clear that the plaintext (or ciphertext) is an important factor in building the leakage models as it is XORed with the key prior to the S-Box in the first round. The output of this operation can be used for labelling traces when training DL models.

$$y = \text{S-Box}\,(p_i \oplus k_i) \tag{3}$$

in which p and k are plaintext and key respectively.

Regardless of the countermeasure utilised, the designer needs to modify the plaintext in some way in order to hide the sensitive value input to the S-Box.

The plaintext will be directly used with the key where no countermeasure is implemented, or masked by one or several masks via XOR to hide the sensitive data, breaking the relationship between the sensitive data in the S-Box computation and the attackers ability to hypothesis on power consumption. Whatever the designers do to protect the sensitive data from side-channel leakage, they will need to modify some signal or variable related to plaintext, and this processing will leave either first or higher-order leakage. There is potential for this leakage to be automatically detected via DL methods in order to find additional points of interest. Hence, the plaintext (or ciphertext) should be considered as an important feature for training DL models. Surprisingly, to the authors knowledge, there are only two reports on using plaintext or ciphertext as additional input feature to increase the efficiency of profiling attacks in general, in which [HJZ12] used the ciphertext bit values to modify amplitude of the traces when training a SVM, and [HGG18] used plaintext as additional knowledge for training and attacking.

2.2 Convolutional Neural Networks

CNNs are a DL architecture that incorporates several different types of layers for detecting features for classification. Our work relies on *convolutional*, *MaxPooling*, *dropout*, *dense* (or *fully-connected*) and *softmax* layers, and the use of the non-linear activation functions (*ReLu* and *Softmax*) [SLJ$^+$14, Kar19, Sah18, Das17].

1. *Convolutional* layers are based on a convolution process, where we take a small array of small size, *e.g.* 5, 7, 13, 19 *etc.*, called the kernel or filter f, and pass it over the input sequence signal (*i.e.* in our case the trace t). After placing the filter over a selected area of sequence input signal, each value from the filter is point-wise multiplied with the corresponding signal point, and summed up for the filtered result of one feature map (FM^i). In detail, each point m of the output feature map is computed by the following transformation (or forward propagation) based on the values from corresponding filter f^i:

$$FM^i[m] = (t * f^i)[m] = \sum_j f^i[j] \times t[m-j] \tag{4}$$

   The filter runs or *strides* over the traces with a step so that if the same feature (*i.e.* leakage) appears in a different position, it can still be detected. *Convolutional* layers can be stacked for finding higher abstractions of feature maps. Additionally, many different filters with the same or different sizes can be applied for corresponding feature detection. The filter parameters are learned through the training process with back-propagation hence are tailored to the specific training dataset.

2. A pooling layer is often used in conjunction with the *convolutional* layer to reduce the size of the parameters to be learned and the subsequent computational requirements. In our model, *MaxPooling* layers are added, where the input trace is divided into different regions, and then the maximum filled value inside that region is considered as the most important feature of that region. This resembles the trace reduction methods in SCA where, for example, only the maximum point per clock cycle is retained as representative of the power consumption of that cycle as a whole.

3. A *fully-connected* layer is flattens the output of the previous layer, combining all previous nodes (or input features $in[i]$) together by summing the bias $b[j]$ and multiplying those input nodes $in[i]$ with corresponding weights $w[j][i]$ to calculate each output node $out[j]$. Each *fully-connected* layer has a number M of output nodes $out[j]$. Correspondingly, an equal number M of bias, $b[j]$, and weight, $w[j][i]$, values

are present. The *fully-connected* layer computes each output $out[j]$ with:

$$out[j] = \sum_i (b[j] + in[i] \times w[j][i]); \qquad j \in [1..M] \tag{5}$$

4. Dropout is known as a simple way to prevent neural networks from over-fitting (predicting the secret key of the training data very accurately, but generalising to an unseen set of traces poorly). Dropout is implemented by randomly disconnecting a percentage of neurons in the network, preventing any particular feature having an disproportionate effect on the model as a whole. A fixed number of nodes are randomly selected and disconnected from the network at each epoch during the learning phase. The number of nodes is selected as a hyper-parameter, and is 40% in our experiment[1].

5. Rectified Linear units (ReLu) is used in as the activation function in our intermediate layers, *i.e.* the *convolutional* and *fully-connected* layers. This function simply replaces a value by itself or else zero where the value is smaller than zero. This helps mitigate the *vanishing gradients* problem which can happen in large networks.

$$ReLu(x) = max(0, x) \tag{6}$$

6. *Softmax* is used for the activation function used in the last *fully-connected* layer. This function is also known as a normalised exponential function, which converts the raw logits scores output of the last *fully-connected* layer into probabilities, where the probabilities of all classes sum to one. This allows key recovery by utilising the probabilities acquired from multiple target traces.

$$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}} \tag{7}$$

3 ASCAD Database and Reference ML Models

An open-source database of side-channel traces, along with some profiling attack models is available at [PSB+18]. The traces are acquired from an ATMega embedded device running AES with higher order masking implemented, with details of the algorithm given in [PSB+18]. The reference attack models provided include TA, multilayer perceptrons (MLPs) and pre-trained CNN models.

3.1 ASCAD database

The ASCAD database targets a software protected AES implementation running on an 8-bit AVR ATMega8515. The software is implemented using assembly for maximum control over register usage. The linear section of the AES algorithm is protected by 16 different random masks, while the non-linear S-Box is protected by a pair of input and output masks. The side-channel information is recorded via the electromagnetic radiation emitted by the device, and is sampled at $2GSs^{-1}$. The target S-Box is the third S-Box operation in the first round as data for that S-Box is provided for both test and training traces. Two datasets are provided, each split into a training and test sets. The test set for both datasets contains a fixed key, while one training set has a fixed key for all traces, while the other training set has a variable key which is randomly generated for each trace. The measurement for the variable key dataset also differ in a viewpoint of signal quality and the number of PoI provided [BCD+17].

[1]Some single PoI size models with 30% dropout without *batch normalization* layers are also tested and achieved similar results.

ASCAD fixed-key dataset: This contains a training dataset group with 50,000 training traces (samples) and test dataset group with $10,000$ attack traces ([PSB+18]), each with a uniformly random plaintext for each trace, and constant key. Two random masks per byte are used, one for masking the plaintext, and the other for masking the S-Box with a different mask for every iteration. Each trace has a dimension of 700 (sample points), and is labeled by the expected value of the *unmasked* output of the 3rd S-Box in the first round. The 3rd S-Box in the first round is used as the ASCAD database with fixed key provides full data for this S-Box for both training and testing data. Even though the masks for S-Box are provided, they are not used in either our training or attacking steps.

This dataset has three variants, with two additional datasets generated from the original by randomly shifting the data. These datasets are referred to as $\text{ASCAD}^{fixed}_{sync}$, $\text{ASCAD}^{fixed}_{desync50}$, and $\text{ASCAD}^{fixed}_{desync100}$. $\text{ASCAD}^{fixed}_{sync}$ contains the original data with synchronised traces, while for $\text{ASCAD}^{fixed}_{desync50}$ and $\text{ASCAD}^{fixed}_{desync100}$ each trace is randomly shifted to the left an amount of $\delta \in [0..50]$ and $\delta \in [0..100]$ respectively. In this paper, we will evaluate our CNNP models on $\text{ASCAD}^{fixed}_{sync}$ and show why our proposed CNNP model would achieve the same result on the other $\text{ASCAD}^{fixed}_{desync}$ datasets.

ASCAD variable-key dataset: This is a set of $200,000$ traces for training and $100,000$ traces for testing. The training traces have random variable keys, as well as random plaintext and mask values. The $100,000$ testing traces have the same key with random plaintext and mask values. Each trace has $1,400$ features and is again labelled by the output of the 3rd S-Box in the first round, giving ≈ 781 traces per label. Similarly with the fixed-key database, the variable-key database has 3 transforms of $\text{ASCAD}^{variable}_{sync}$, $\text{ASCAD}^{variable}_{desync50}$, and $\text{ASCAD}^{variable}_{desync100}$, and we evaluate our proposed CNNP models on $\text{ASCAD}^{variable}_{sync}$ and $\text{ASCAD}^{variable}_{desync50}$ variants to allow comparison with the provided reference. Even though no reference is provided for $\text{ASCAD}^{variable}_{desync100}$ database, additional experimentation is conducted on this also to understand how well our CNNP works with a highly de-synchronized database.

3.2 Reference Comparison Models

In this paper, we compare our profiling results against four publicly available models trained and tested against the same dataset. Those models are provided as part of the ASCAD database [PSB+18].

TA (on first order leakage): The ASCAD provided *simple_model* contains a model for running a template attack, which relies on computing a simple cross tabulation (*pd.crosstab*) of the mask and the masked S-Box output factors, so that the effectiveness of the mask is removed, making it equivalent to a template attack on first-order masking. This model is referred as the *simple_model* in subsequent sections of our paper as its original name in the $\text{ASCAD}^{fixed}_{sync}$ database.

MLP: A MLP model which includes 5 *hidden layers* with 50 neurons each and one final *softmax* layer with 256 units, one for each class is also provided. In addition, *batch-normalization* is applied to the input traces before getting to the first *hidden layer*. This model will be referred as *other_mlp_model* in subsequent sections.

MLP (on first order leakage): The provided reference model MLP_{best} model has 5 *hidden layers* with 700 neurons in the first one, and 200 neurons in each of the remaining. The final *softmax* layer has 256 units, one for each class [PSB+18]. In case of first-order attack using MLP_{best} model, it is assumed that an adversary knows the masked output of the AES S-Box, hence can compute the actual S-Box value that is returned:

$$\overline{\text{S-Box}\,(x[3])} = \text{S-Box}(p_i[3] \oplus k_i[3]) \oplus m_{i,out}[3] \tag{8}$$

where $p_i[3]$, $k_i[3]$, and $m_{i,out}$ are the third plaintext, key, and mask bytes from trace t_i respectively. S-Box is the original substitution box in the AES algorithm and $\overline{\text{S-Box}}$ is the actual masked value from precomputed memory. In this weak context, this reference MLP model only requires 4 traces to recover secret key bytes as noted in [PSB+18]. In our paper, this MLP_{best} model is verified with high-order attack and denoted as *Prouff MLP* in comparison figure.

ASCAD CNN models: The ASCAD reference CNN model is based on the *VGG-16 CNN* model proposed in [SZ15], with some additional modifications in the CNN configuration to keep the global volume and amount of information treated by the different layers as constant as possible [PSB+18]. This model has 5 *convolutional* layers with a differing number of filters of the same kernel size of 11, two 4096 unit *fully-connected* layers and a final *softmax* layer with 256 units. The reference model is effective at key recovery in the presence of higher-order masking, and is referred to as *Prouff CNN* in our paper.

4 Plaintext Features for DL Based SCA

While plaintexts (or ciphertexts) are assumed known values to an attacker in SCA, and are generally the first data that are masked in a protected implementation, to the authors knowledge there is minimal related work considering plaintexts or ciphertexts as an additional feature for training models in profiling attacks [HGG18]. Effective proposals to improve the performance of DL models include data augmentation (*i.e.* adding noise to the traces [KPH+19]), or profiling data from multiple physical devices in order to generalise the model.

While these methods should also improve the performance of our model architecture, they are not considered in this work. We propose that the plaintext be incorporated as an important feature for building more accurate DL models in order to better extract sensitive data and features for revealing the secret key. From a data aspect, three inputs will effect a power-trace: plaintext, mask(s), and key. Providing the plaintext (or ciphertext) helps the DL model reduce the number of ambiguous factors and allows it to better learn features related to the unknown mask(s) and key for classification.

4.1 Plaintext Feature Encoding

The plaintext values are incorporated to the model as additional features. While a number of different encoding methods can be used, we focus on two. There are two encoding methods for merging plaintext features, integer and one-hot encodings, as follows:

Integer encoding: In this encoding method, the plaintext feature is considered as a single additional neuron in the model, in which the value of plaintext byte is inserted. This node or plaintext feature is inserted at one of the *convolutional* or the *fully-connected* layers. This addition has a minimal effect on the number of additional parameters to be learned, however increases the sensitivity of the model to the points where the plaintext is being processed. However, when using the integer encoding method the CNN model may interpret the additional feature to have some kind of order or hierarchy, and care must also be taken to ensure the feature does not have a disproportionate effect on the learning process where the absolute values are considerably larger than other feature values in that layer.

One-hot encoding: This method expands the plaintext byte into a 256 element sparse feature vector, in which the element indexed by the plaintext value is set to one and all other elements are set to zero. This feature vector is again added to one of the *convolutional* or *fully-connected* layers as before. While one-hot encoding increases the number of parameters that need to be learned in the training process, it overcomes some of the potential issues with integer encoding.

In order to support for this plaintext feature extension, we introduce an additional ***Plaintext extension*** function, that connects the additional plaintext feature vector (either with a single or 256 elements) in our CNNP models.

4.2 CNNP model for SCA

4.2.1 Attacker Assumptions

When building a model of the power consumption for the target device, the following assumptions hold:

- The attacker cannot access the design or values used for random number generation.
- The attacker has access to the plaintext and/or ciphertext values for both training and attack stages.
- The attacker can profile keys on the device.
- The attacker does not know any implementation details except the algorithm being targeted, but can understand that the designer may or may not have applied countermeasures such as time shifting, 1^{st} or higher-order masking, or secret sharing schemes.

4.2.2 Attack Model

The CNNP model attacks AES implementations based on recovering the unmasked value of S-Box output (even where that value is masked) which allows key recovery given knowledge of the plaintext. In particular, we trained the proposed CNNP model by labelling the traces by the 3^{rd} S-Box byte, which is computed from values known to the attacker during the profiling process as given in Equation 3. This labelling method follows that as used by the reference models on ASCAD databases. In order to attack the remaining key bytes, the traces would have to be relabelled for each S-Box output and the model re-trained. However the same power traces can be used to atttack the other key bytes.

A separate set of traces, unused in the training process, is used for testing the model verification in the attack stage, with an output probability returned for each possible S-Box value. As the correct key and plaintext values are known, these can be used to rank the S-Box predictions. Both individual trace probabilities and maximum likelihood scores can be used for evaluation. For the experimental results in this paper, comparisons are based on the maximum likelihood score as while CNNPs have the power to recover key values within a single trace when attacking the ASCAD^{fixed} dataset, the maximum likelihood scores are needed to combine multiple trace values when attacking the ASCAD^{variable} dataset.

4.2.3 Maximum Likelihood Score

This evaluation step utilises maximum likelihood scores for combining results of attacking multiple different traces with the same key as outlined in [SMY09]. It estimates the *likelihood* of each hypothesis key by multiplying the classification probability given by independent traces to create a *scores vector*. Values in this *scores vector* are then sorted according to probability for ranking. Accuracy is considered as the location of the correct key in the ranked *scores vector*. The closer the rank of the correct key is to 1, the higher the accuracy of the evaluated model.

4.2.4 CNNP Models with a Single Convolutional Filter Kernel Size

Taking into account the three considerations outlined in Subsection 1.1 for building the model, a CNNP model with a single *convolutional* filter kernel size is given. In order to benchmark the more complex model given later, an initial CNNP model with a single *convolutional* filter size is first presented. It is divided into four main modules consisting of feature convolution, plaintext extension, plaintext feature embedding via the *fully-connected* layers before the final *softmax* layer for classification.

Figure 1 shows our proposed CNNP model implementation with single *convolutional* filter kernel size (CNNP$_{\text{single}}$). The plaintext feature extension can be implemented with either integer or one-hot encoding method. Corresponding models are named as CNNP$^{integer}_{single}$ and CNNP$^{one-hot}_{single}$ respectively, in which $_{single}$ here means a single *convolutional* filter kernel size, which can be replaced by the actual sizes used during experimentation, such as $[3, 5, 7, 13, 19]$. Two *convolutional* architectures are given. In these, each *convolutional* layer has the same filter kernel size but the number of *convolution* filters in each layer (the dimensionality of the output space) are different. The first structure, the 3-layer CNN version in Figure 1a has number of *convolutional* filters reduced from 512 to 256 and then 128 while the second one, the 4-layer CNN version in Figure 1b has those numbers increase from 64 to 128, then 258 and finally 512. Those *convolutional* layers are connected through *MaxPooling* layers with *pool_size* of 3 and a *stride* of 3. The resulting features after the last *MaxPooling* layer are then extended by adding the plaintext feature vector before input to the 1024 neuron *fully-connected* layer. Adding the plaintext feature here allows the model to combine this feature with the filtered trace features. This *fully-connected* layer combines all those features together with different weights on each node, which are learnt by the training algorithm. Those features are then processed in another *fully-connected* layer[2] before applying a dropout rate of 40% in order to reduce overfitting of the model on the training data. The model is completed with three more *fully-connected* layers with the same number of nodes and a *softmax* layer for classification.

The model for CNNP with a single *convolutional* filter kernel size can be given by the following formula:

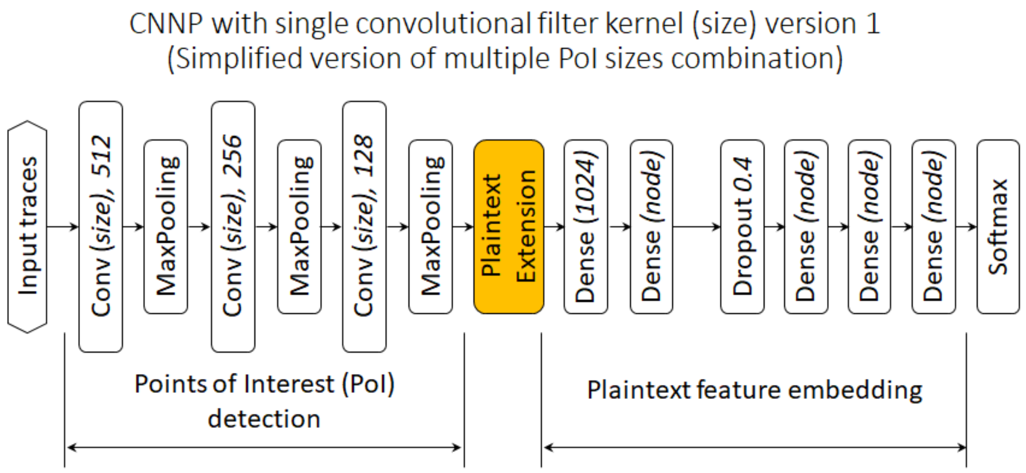
$$s \circ [\lambda^3] \circ \beta \circ [\lambda^2] \circ [P_{\text{ext}} \circ [\delta \circ [\alpha \circ \gamma_x]]^3] \tag{9}$$

in which, s, λ, β, P_{ext}, δ, α, and γ are the *softmax*, *fully-connected*, *dropout*, *plaintext feature extension*, *MaxPooling*, *activation*, and *convolutional* layers respectively. γ_x is a *convolutional* layer with filter kernel size $x \in [3, 5, 7, 13, 19]$ so that the model accounts for the most interesting PoI during the training phase. As image processing literature often uses small kernel sizes, we chose a range of sizes up to and beyond that used in [PSB+18] which uses a size of 11 when attacking the ASCAD database.

In order to support the *Plaintext extension* function, the computation of the *Dense* 1024 layer needs a slight modification in order to match the increase in the number of inputs. Depending on the size of the input trace (700 points in the ASCAD^{fixed} and 1400 points in the ASCAD^{variable} databases) the number of outputs of the *MaxPooling* layer before *plaintext extension* layer is different, as shown in Figures 10, 11, 12, 13, 14, 15, and 16 in the Appendix.

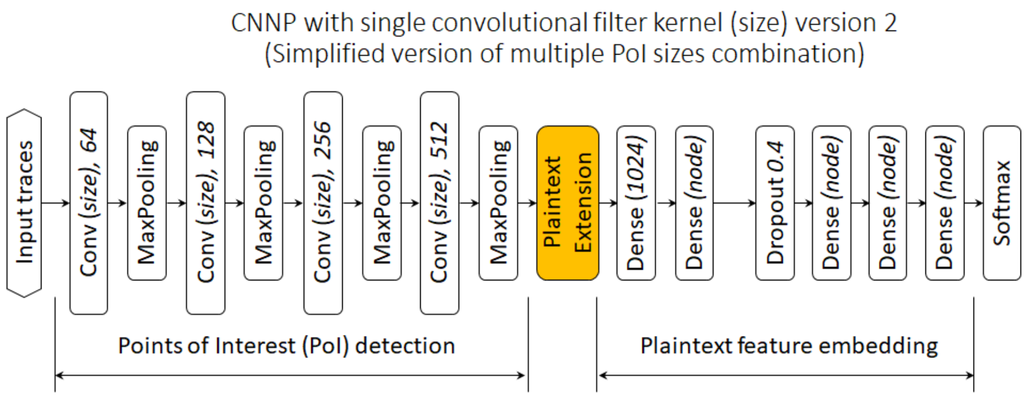
Given N outputs from the last *MaxPooling* layer, in the integer encoding method for *Plaintext extension*, only one neuron is added to the output of the *MaxPooling* layer,

[2]The number of nodes in these fully connected layers can vary but empirically 512 and 1024 nodes have shown good performance.



(a) CNNP$_{\text{single}}$ [a] model with single *convolutional* filter kernel size and 3-layer CNN version.

[a] *Plaintext extension* layer can be located before or after the first *Dense(1024)* layer and are denoted as "Pext loc.1" and "Pext loc.2" in our experiments.



(b) CNNP$_{\text{single}}$ model with single *convolutional* filter kernel size and 4-layer CNN version.

Figure 1: CNNP$_{\text{single}}$ [b] model with single *convolutional* filter kernel size [c]

[b] The *plaintext extension* layer increases the number of neurons by 1 for integer encoding or 256 for one-hot encoding

[c] *batch normalization* layers are included after all other layers for models used with ASCAD^{variable} database but deducted in this figure for simplification.

increasing the input length to the following *Dense*1024 layer to $(N+1)$. The formula for the *fully-connected* layer in Equation 5 with M nodes is now:

$$Dense1024[j] = \sum_{i=1}^{N}(b[j] + in[i] \times w[j][i]) + b[j] + P \times w^P; \qquad j \in [1..M] \qquad (10)$$

in which $Dense1024[j]$ is the jth output of the *fully-connected* layer. P and w^P are the raw values of the plaintext and its corresponding weight.

In the one-hot encoding method for *Plaintext extension*, 256 neurons are appended onto the output of *MaxPooling*, modifying the input to the following *Dense*1024 layer to

$(N + 256)$. The formula for the *fully-connected* layer with M nodes is now:

$$Dense1024[j] = \sum_{i=1}^{N}(b[j]+in[i]\times w[j][i])+\sum_{k=1}^{256}(b[j]+P[k]\times w^P[k]); \qquad j \in [1..M] \quad (11)$$

in which $Dense1024[j]$ is the jth output of the *fully-connected* layer. $P[k]$ is the one-hot encoding of the plaintext value, and is a 256 element all-zero vector except the element indexed by the value of the plaintext, which is set to 1. $w^P[k]$ is a 256-element vector containing the weight value for each individual $P[k]$. The additional part $b[j] + P \times w^P$ in integer encoding or $\sum_{k=1}^{256}(b[j] + P[k] \times w^P[k])$ in one-hot encoding allows each output node to adjust itself dependent on the input value of the plaintext.

4.2.5 CNNP Models with Multiple Convolutional Filter Kernel Sizes

CNNP$_{multi}$ is an extension of the CNNP$_{single}$ model in Figure 1, in which a number of different *convolutional* filter kernel sizes are used together in parallel to deal with unknown factors in the implementation as used in GoogLeNet [SLJ+14]. Additional, parallel, filter sizes are also expected to help the proposed CNNP models to work better against the random delay countermeasure as the length of *convolutional* filter should cover the most interesting PoI [PSB+18, Mag19]. Figure 2 shows the proposed CNNP$_{multi}$ model implementation, combining two different *convolutional* filter kernel sizes of 3 and 5 and the two *convolution* structures shown in Figure 1 together. Other similar models can be built or extended using different *convolutional* filter kernel sizes and numbers. In addition to the *Plaintext extension* layer introduced previously, an additional *feature combination* function is required in order to connect the filtered features from each *convolutional* layer branch. An additional *Dropout* layer set to 40% is used after the *feature combination* layer, before three *fully-connected* layers are used to learn any relationships between the features of the different branches. As before, the final layer consists of a *softmax* layer to return the class probabilities. Two CNNP$_{multi}$ versions are given with *plaintext extension* using integer and one-hot encoding methods, named as CNNP$_{multi}^{integer}$ and CNNP$_{multi}^{one-hot}$, respectively. The CNNP architecture with multiple *convolutional* filter kernel sizes can be described by the following formula:

$$\begin{aligned} s \circ [\lambda^3] \circ \beta \circ [[\lambda^2] \circ [P_{\text{ext}} \circ [\delta \circ [\alpha \circ \gamma_x]]^3]] joint [[\lambda^2] \circ [P_{\text{ext}} \circ [\delta \circ \dots \\ [\alpha \circ \gamma_y]]^3]] joint [[\lambda^2] \circ [P_{\text{ext}} \circ [\delta \circ [\alpha \circ \gamma_z]]^3]] \end{aligned} \quad (12)$$

in which, s, λ, β, P$_{\text{ext}}$, δ, α, γ and *joint* are *softmax*, *fully-connected*, *dropout*, *Plaintext extension*, *MaxPooling*, *activation*, *convolutional*, and *joint* layers respectively. Subscripts x, y, z are different *convolutional* kernel sizes. In our experiments, sizes 5, 7, 13, and 19 are used for fixed key database and sizes 3, 5 are used for variable database.

4.2.6 CNNP Network Structure and Selection of Hyperparameters

The *convolutional* layer to detect PoI is designed taking into account that the larger number of filters we have, the greater number of PoI patterns we can find, therefore our model can detect more important combined leakage features in a trace. Hence, we select the initial filter kernel size as 512, similar to the largest number of filters in [PSB+18] and gradually reduce over each convolution time to size of 64. Different from [PSB+18], we would like to select local PoI features instead of averaging them, hence, *MaxPooling* layers size 3 with stride 2 are applied after each *convolutional* layer. Kernel filter sizes are selected as 3, 5, 7, 11, 13, 19. Kernel filter sizes up to 11 have empirically proven to be sufficient enough for ASCAD database (fixed key) as outlined in [PSB+18], but are extended to 13 and 19 for

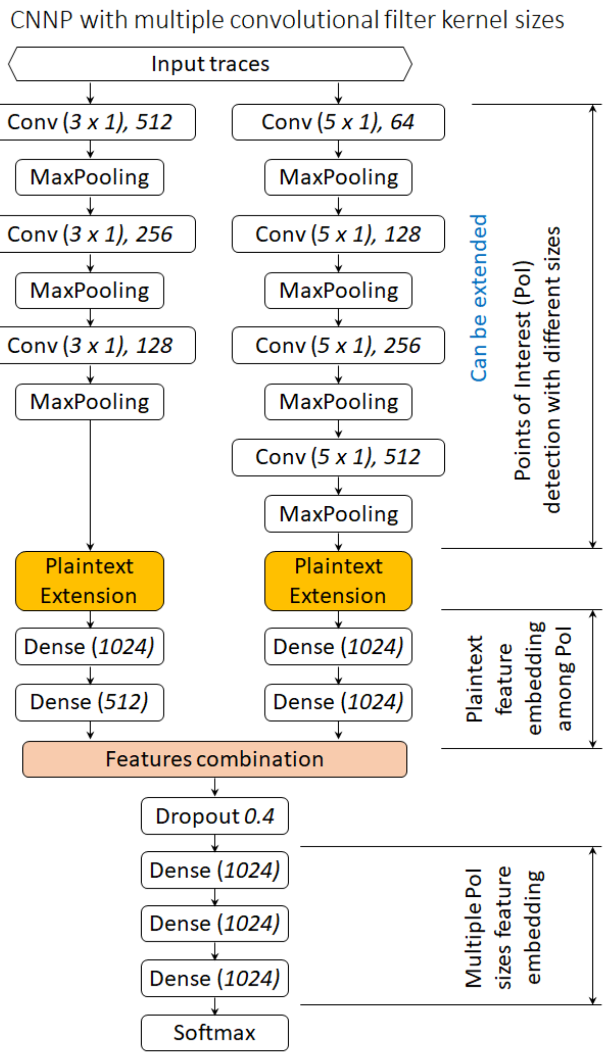


Figure 2: CNNP$_{multi}$ model with multiple *convolutional* filter kernel sizes (3 and 5)[a].

[a] *Batch normalization* layers are included before all *convolutional* and *dense* layers but deducted in this figure for simplification.

the variable key database due to its higher sampling rate combined with the fact that the number of sampling points is doubled.

ReLu is selected as the activation function, optimizer is *RMSprop*, and learning rate is 10^{-5} as a reference from [PSB+18].

While many hyperparameters from [PSB+18] are re-used in this work, the depth (the number of *fully connected (dense)* layers) and width (the number of nodes in a *fully connected* layer) of the model is still a relevant topic of research. DL research from [AJR+18, BC14] imply that deeper networks (which incorporate more layers for better feature embedding) provide better accuracy compared to wider networks with the same number of parameters. We develop a deeper but narrower model with a total of five *fully-connected* layers, two are placed prior to the feature combination, with the remaining three placed afterwards, with a final *softmax* layer for classifying the SBox output, as shown in Figure 1.

It is proposed to place the *plaintext extension* layer after the convolutional layer before either of the first two *fully-connected* layers. In the experiments that follow, the *plaintext extension* layer is placed before the first *fully-connected* layer to allow forward propagation through all *fully-connected* layers while having a direct connection to all outputs of the CNN layer. An additional experiment with the *plaintext extension* layer located at the input of second *fully-connected* layer is also given for comparison.

5 Experimental Results

While a number of variants of the CNNP architecture with different hyperparameters were tested, in the following we present results from five $\text{CNNP}_{\text{single}}$ instances with different *convolutional* filter sizes of $[3, 5, 7, 13, 19]$, and one CNNP_{multi} model with three *convolutional* filter kernel sizes of $[7, 13, 19]$. Each model is tested with both the integer and one-hot encoding for incorporating the plaintext feature extension. The models are trained and evaluated using the ASCAD database traces, on both the fixed-key and variable-key trace sets. For each trace set, the aligned traces and de-synchronised traces in range of $\delta \in [0..50]$ points are used, giving four different databases tested. Based on experimental results, not all models are evaluated on all trace sets.

For the fixed key trace dataset, our experiment shows that 64 training epochs[3] is sufficient for our CNNP models to recover the key from only one or two attack traces. A greater number training epochs is required for training CNNP models on the variable key ASCAD database. The number of training epochs can be selected by analysing the loss and accuracy scores of the validation dataset. Our models were trained with 1024 epochs, with the state saved every 16 epochs. In the main, we randomly select 10,000 traces from the attack (or test) dataset as validation data for verification of the correct key rank at each saved epoch state. Training is performed on a VMware virtual machine, with access to virtual NVIDIA GRID M60-8Q and M40-4Q GPUs with 8GB and 4GB memory respectively.

In experiments using the ASCAD database, the *convolutional* filter kernel sizes, the number of training epochs and time for each model, together with the rank of the correct key are reported. Our comparison method is straightforward, in that we train our CNNP models with the *training dataset* group and evaluate the trained models on the separate *test dataset* group provided by ASCAD. At each *run*, a number of traces are randomly selected from the test dataset for the attack phase, with the maximum likelihood score of each hypothesis key calculated as a function of the number of traces. These maximum likelihood scores are then sorted after each run and the rank of the correct key is recorded. N runs are evaluated and mean of the correct key rank is computed. We evaluate our models using 10,000 traces from test dataset. Depending on how fast the key rank converges, the number of traces for each run is different; it is set to 20 for the ASCAD^{fixed} and 100 for the $\text{ASCAD}^{variable}$ datasets. Hence, N is set to 500 and 100 respectively. In the evaluation with desynchronized data for variable key dataset, the number of traces for each run is 1,000, hence $N = 10$. The better performing models are the ones that have a lower key rank with similar or less traces required.

5.1 Evaluation on ASCAD Fixed Key Datasets

We now present results for our CNNP models on the $\text{ASCAD}^{fixed}_{sync}$ and $\text{ASCAD}^{fixed}_{desync50}$ databases. Each model is trained on the synchronised profiling data and tested on the synchronised and desynchronised attack datasets. Each model is also trained on desynchronised profiling data[4] and tested on the synchronised and desynchronised attack datasets.

[3]One epoch is when the entire set of training samples have been used for training.

[4]This can be viewed as somewhat similar to the augmented data approach in [KPH+19]

Results are compared with that of the reference models as outlined in Subsection 3.2. Table 1 shows the names and parameter settings for the different models and their running time (on NVIDIA GRID M60-8Q) in minutes for reference. In addition to the CNNP models, results for the model named CNN*$_{(5)}$ are provided, which is the modification of CNNP model in Figure 1 with the *plaintext extension* layer removed, and a *convolutional* filter size of 5. This model works as a reference to highlight the impact of the *plaintext extension* layer. An additional experiment with the ASCAD fixed database for training CNNP using random values for the traces is also given as "CNNP*$^{one-hot}_{(5)}$" model. The purpose of this model is to detail exactly why the model has the excellent performance as shown.

Table 1: Evaluated models on the ASCAD^{fixed} databases.

	Model names	Conv. kernel size	Training on ASCAD$^{fixed}_{sync}$		Training on ASCAD$^{fixed}_{desync50}$	
			No. of epochs	Time (min)	No. of epochs	Time (min)
One-hot encoding	CNNP$^{one-hot}_{(7,13,19)}$	7, 13, 19	14	20.32	16	23.49
	CNNP$^{one-hot}_{(5)}$	5	22	8.87	20	8.07
	CNNP$^{one-hot}_{(7)}$	7	20	8.39	22	9.23
	CNNP$^{one-hot}_{(13)}$	13	18	8.71	20	10.00
	CNNP$^{one-hot}_{(19)}$	19	16	9.29	16	9.29
Integer encoding	CNNP$^{integer}_{(7,13,19)}$	7, 13, 19	62	91.00	62	91.00
	CNNP$^{integer}_{(5)}$	5	64	25.81	58	24.32
	CNNP$^{integer}_{(7)}$	7	62	26.00	52	21.81
	CNNP$^{integer}_{(13)}$	13	60	29.03	60	30.00
	CNNP$^{integer}_{(19)}$	19	52	30.19	58	33.68
References	simple model	-	-	-	-	-
	other mlp 01	-	-	-	-	-
	Prouff MLP	-	-	-	-	-
	Prouff CNN	11	-	-	-	-
	CNN*$_{(5)}$[5]	5	64	25.81	-	-
	CNNP*$^{one-hot}_{(5)}$[6]	5	48	-	-	-

CNN*$_{(5)}$ model is the modification of CNNP$^{one-hot}_{(5)}$ model by removing plaintext feature extension layer.

CNNP*$^{one-hot}_{(5)}$ is the CNNP$^{one-hot}_{(5)}$ model but random values are used as traces when training.

5.1.1 Evaluation on ASCAD Fixed Key Synchronised Dataset

Figure 3 shows the experimental results of our 10 proposed CNNP$^{\text{integer}}$ and CNNP$^{\text{one-hot}}$ models on the ASCAD$^{\text{fixed}}$ database using the metric defined in Subsubsection 4.2.2. Each of the CNNP$^{\text{integer}}$ and CNNP$^{\text{one-hot}}$ variants include 5 sub-models with different *convolutional* filter kernel sizes as outlined in Table 1.

The effectiveness of the *plaintext extension* layer against this dataset can be seen by comparing the attack results of the CNNP$^{one-hot}_{(5)}$ and CNNP$^{integer}_{(5)}$ models with that of CNN*$_{(5)}$. They have the same network structure as described in Figure 1, except that

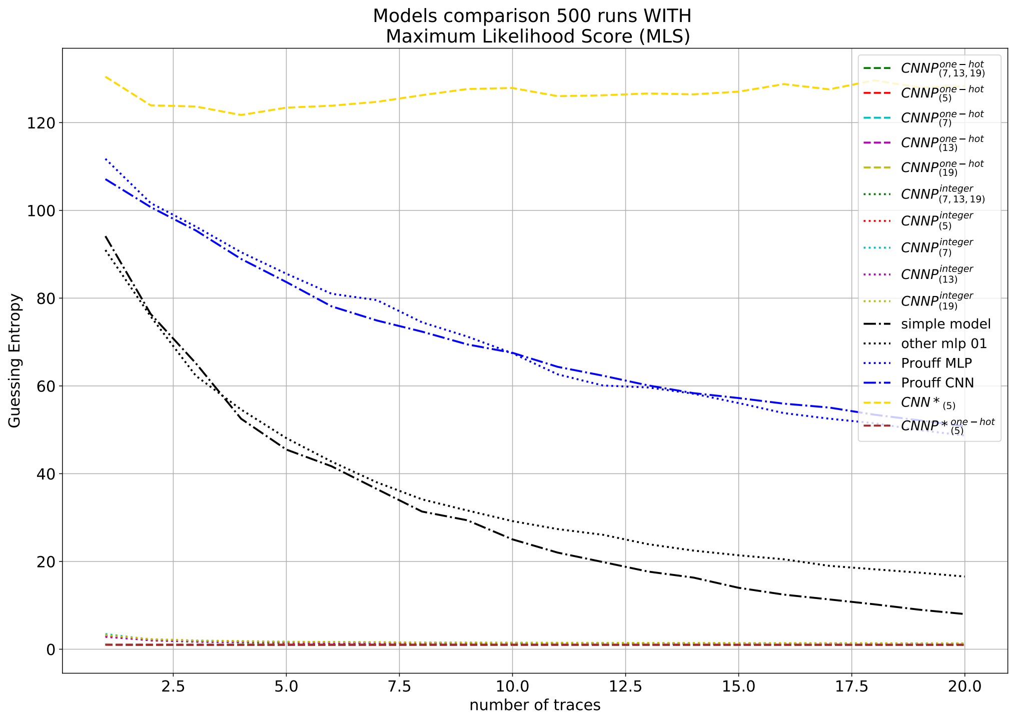


Figure 3: Comparison of CNNP models with *convolutional* filter kernel sizes of $[5, 7, 13, 19]$. Keys are ranked by maximum likelihood score as a function of the number of attack traces. The blue and black lines are reference models from the ASCAD database as described in Subsection 3.2.

plaintext extension layer is removed from the CNN$*_{(5)}$ model. Without this, the CNN$*_{(5)}$ model essential achieves result no different to random guessing.

It can be seen that our CNNP$^{\text{integer}}$ and CNNP$^{\text{one-hot}}$ models greatly outperform the reference models available as part of the ASCAD database. Our CNNP$^{\text{integer}}$ models achieve an expected key rank of 3 after a **single** trace when using additional integer encoded plaintext features. This leads to potential key recovery where only a single encryption trace is available, *e.g.* some challenge-response protocols. If two traces are available, then it is expected that successful key recovery will occur without any subsequent key enumeration stage. Only minor differences are visible in the likelihood scores between the different *convolutional* filter kernel sizes. Using the one-hot encoding method, only a single trace is required for key recovery, and the training stage also requires considerably less computation. These results were obtained after 62 epochs in CNNP$^{\text{integer}}$ and 14 epochs in CNNP$^{\text{one-hot}}$.

5.1.2 Explanation of Fixed Key Dataset Results

In order to understand why adding the plaintext increases the accuracy of the attack on the ASCAD^{fixed} dataset, an additional experiment is executed by replacing all traces in the training process by random values and is named CNNP$*^{one-hot}_{(5)}$ in Figure 3. Despite using random traces, the model can still correctly predict the key value. This can be explained as since the key K is fixed, and the trace labels are $S[P \oplus K]$, giving the plaintext P as a feature implies that the network only needs to learn the bijection $S[(.) \oplus K]$ and considers the input traces as noise. Hence, no additional experiments are conducted with the ASCAD^{fixed} dataset.

5.2 Evaluation on ASCAD Variable Key Datasets

In this section, we evaluate our CNNP models on the ASCAD$^{variable}_{sync}$, ASCAD$^{variable}_{desync50}$ and ASCAD$^{variable}_{desync100}$ databases. Each model is trained on the synchronised training data and tested on the synchronised and de-synchronised testing data, as well as trained on the de-synchronised training data and tested on the synchronised and de-synchronised testing data. Two sub-groups of 10,000 traces are randomly selected from testing dataset for "epoch selection" and "model testing" purposes. The models are each trained with 1,024 epochs and stored every 16 epochs, giving 64 instances of each model. The model instance used for evaluating the test set, is that with the lowest mean key rank after 50 traces when evaluated on the "epoch selection" set. As in an attack scenario the correct key is not known a-priori, a separate data set (i.e. the validation data) is required to know how long the model runs for. Table 2 shows the model names, their *convolutional* filter kernel sizes and number of nodes per neural network layer, plaintext feature extension, and the number of training epochs selected as described. The training time of each model is not provided as the models are trained using different GPUs (M60-8Q and M40-4Q). As different model hyperparameters, such as the number of nodes per *fully connected (dense)* layer effects the model accuracy, we compare results among our best models with other references in the following sections. Full comparisons among our proposed CNNP models can be found in Figure 9 of the Appendix.

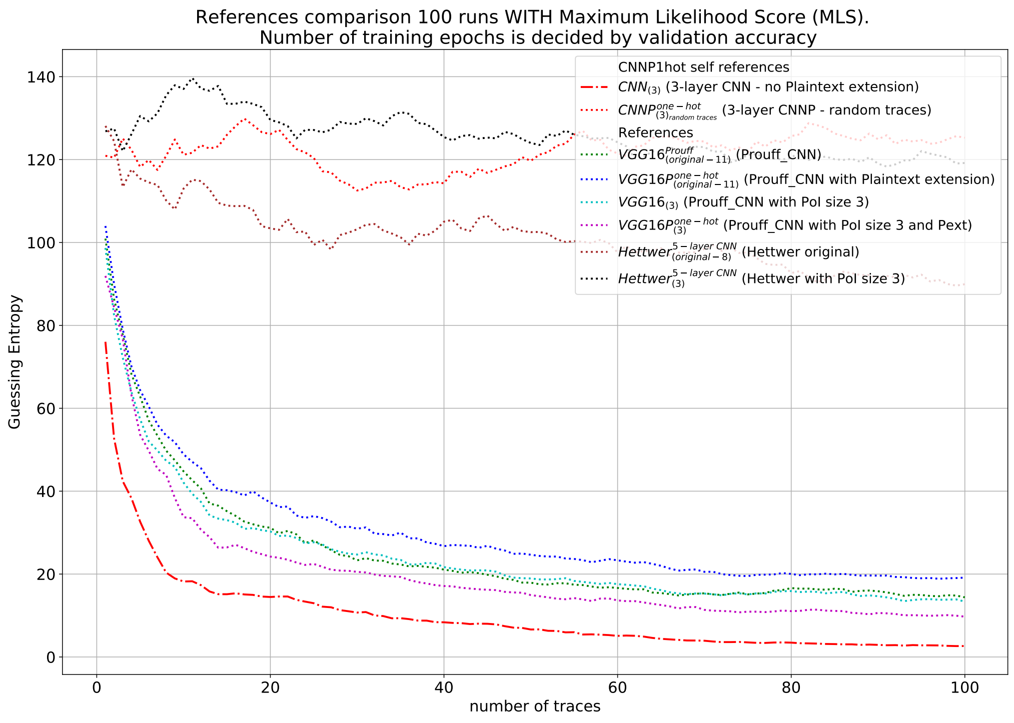
5.2.1 Evaluation on ASCAD Variable Key Synchronised Dataset

Figure 4 shows the key ranking comparison between our proposed CNNP$^{\text{one-hot}}$ model and other similar works like the work of *Hettwer* [PSB+18] and *Prouff CNN* [HGG18]. Additional experiments like adding plaintext into the input and modifying the *convolutional* filter kernel sizes of these references are also given. We do not give a full set of comparisons as the ASCAD^{fixed} database results show that given sufficient training data, the larger sizes of the filter kernels in the *convolutional* layer tend to perform worse than the smaller ones, and that CNNP$^{\text{integer}}$ models perform worse than CNNP$^{\text{one-hot}}$ models. We give results for the 10 CNNP models together with references as outlined in Table 2.

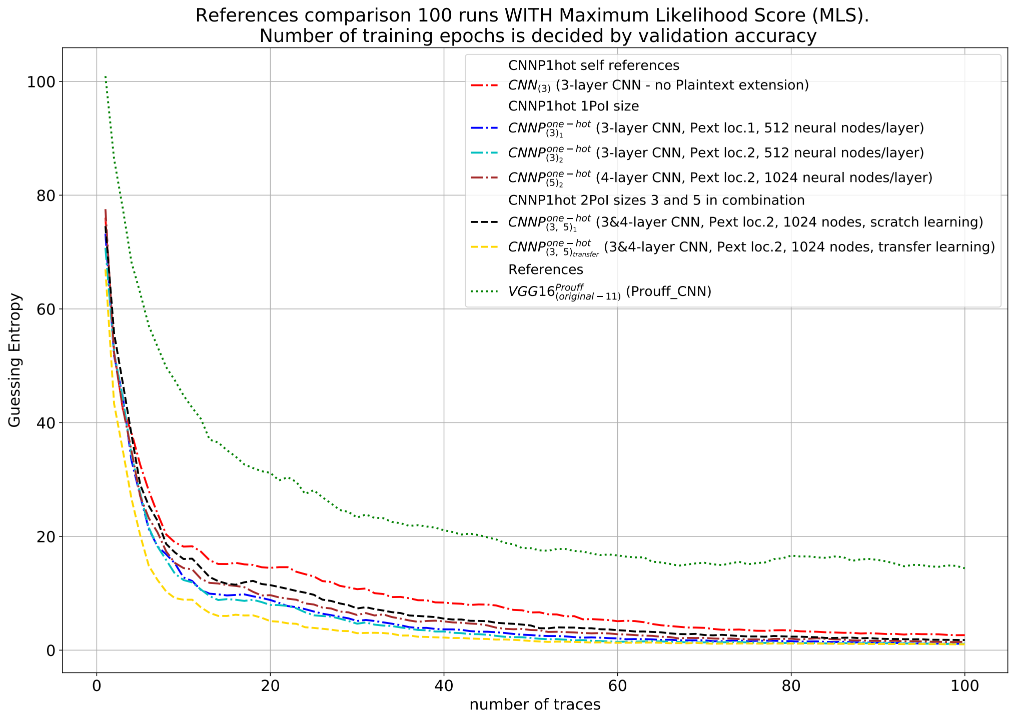
As can be seen in Figure 4a, our proposed structure with *convolutional* filter size 3 (PoI size 3) and *without* the plaintext extension outperforms all the references of *Prouff CNN* and *Hettwer 5-layer CNN*. With 100 attack traces, our proposed CNN structure achieves a key rank of 2 while *Prouff CNN* gets key rank 18 and *Hettwer 5-layer CNN* obtains a result slighty better than a random guess. This shows that a generic neural network structure applicable to many datasets is a non-trivial problem as the *Hettwer 5-layer CNN* was not designed with a view to attacking the ASCAD^{variable} dataset. Experiments with the CNNP model, in which plaintexts and random traces are considered as training inputs, also returns a key rank similar to a random guess unlike in Subsubsection 5.1.2 with the ASCAD^{fixed} dataset. This further re-enforces that the model was simply learning a bijection due to the fixed key previously.

Figure 4b shows the comparison among different CNNP models and the best reference, the *Prouff CNN* model[7]. It can be seen that all our proposed CNNP models with different kernel sizes and neural architectures achieve better results than the reference. The models obtain an estimated key rank of 3 after 60 traces, meaning that to recover the entire key from a secure AES implementation with high-order masking, the brute force key search should take at least $3^{16} = 43,046,721$. Among the provided CNNP models, the model that combines two different *convolutional* filter sizes (3 and 5) and structures in Figure 2

[7] Figure 4b shows results from the best performing models. A full comparison is shown in the Appendix, Figure 9.



(a) Key ranking comparison between proposed CNN network structures (no plaintext extension) and other references on $\mathrm{ASCAD}_{sync}^{variable}$ databases.



(b) Key ranking comparison among proposed CNNP structures on $\mathrm{ASCAD}_{sync}^{variable}$ databases.

Figure 4: Key ranking comparison on $\mathrm{ASCAD}_{sync}^{variable}$ databases.

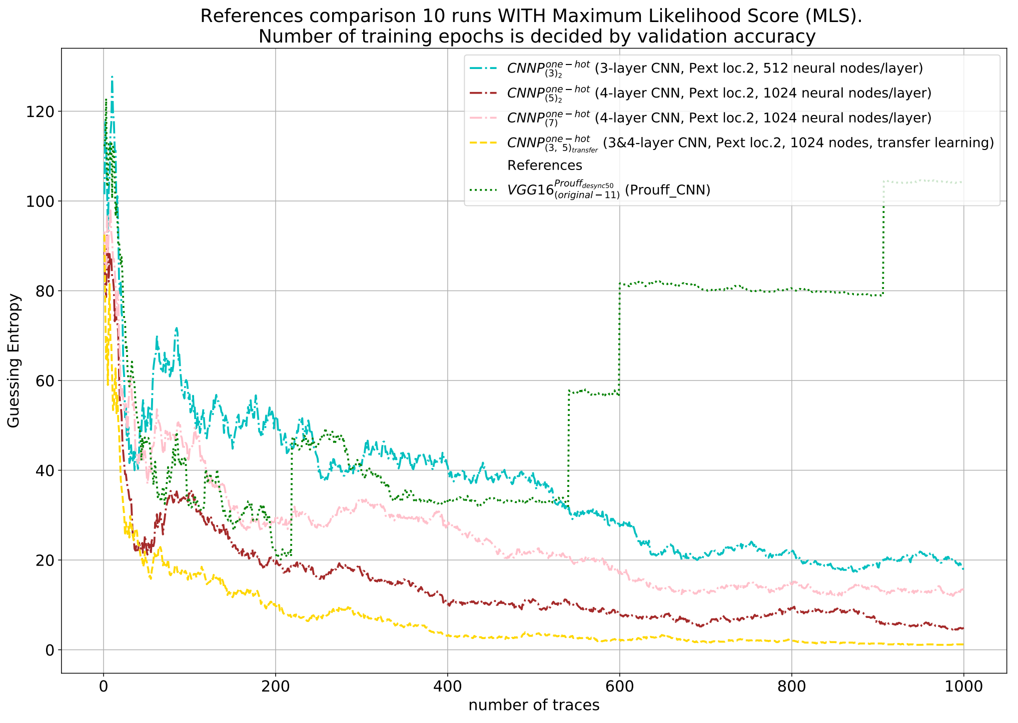
Table 2: Evaluated models on ASCAD^{variable} databases.

	Model names	Conv. kernel size	Nodes per *Dense* (node)	Plaintext extension	Number of training epochs on ASCAD^{variable} databases		
					with *sync*	with *desync50*	with *desync100*
Self ref.	CNN $_{(3)}$	3	512	No	416	-	-
	CNNP$^{one-hot}_{(3)_{random\ traces}}$ [a]	3	512	Yes	80	-	-
Proposed CNNP	CNNP$^{one-hot}_{(3)_1}$	3	512	Yes	544	-	-
	CNNP$^{one-hot}_{(3)_2}$	3	512	Yes	384	368	976
	CNNP$^{one-hot}_{(3)_3}$	3	1024	Yes	240	-	-
	CNNP$^{one-hot}_{(3)_4}$	3	1024	Yes	368	-	-
	CNNP$^{one-hot}_{(5)_1}$	5	512	Yes	352	-	-
	CNNP$^{one-hot}_{(5)_2}$	5	1024	Yes	256	192	464
	CNNP$^{one-hot}_{(7)}$	7	1024	Yes	-	384	336
	CNNP$^{one-hot}_{(3,\ 5)_1}$	3, 5	1024	Yes	224	-	-
	CNNP$^{one-hot}_{(3,\ 5)_2}$	3, 5	1536	Yes	80	-	-
	CNNP$^{one-hot}_{(3,\ 5)_{transfer}}$ [b]	3, 5	1024	Yes	70	20	16
References	VGG16$^{Prouff}_{(original-11)}$	11	4,096	No	64	-	-
	VGG16P$^{one-hot}_{(original-11)}$	11	4,096	Yes	128	-	-
	VGG16$_{(3)}$	3	4,096	No	64	-	-
	VGG16P$^{one-hot}_{(3)}$	3	4,096	Yes	192	-	-
	Hettwer$^{5-layer\ CNN}_{(original-8)}$	8	400	Yes	16	-	-
	Hettwer$^{5-layer\ CNN}_{(3)}$	3	400	Yes	16	-	-

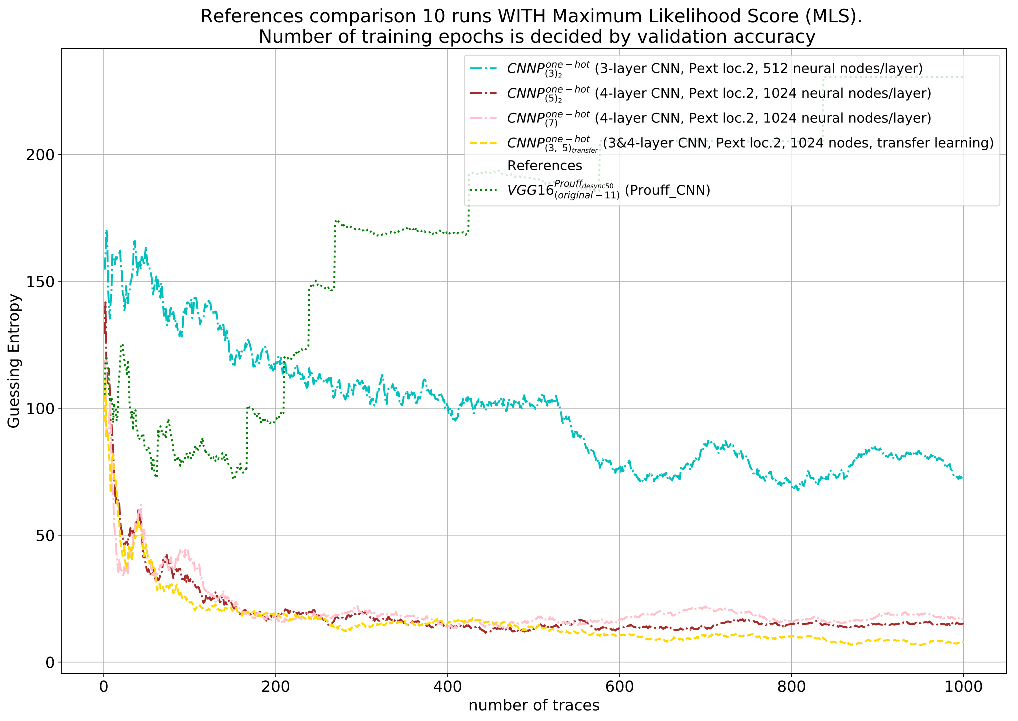
[a] We develop some model versions with the same *convolutional* filter kernel sizes (filter sizes) but different location of *plaintext extension* module and number per *dense* layer. Hence, model version is included after filter sizes in subsubscript format, making the model name to $Model\ name_{(filter\ sizes)_{version}}$.

[b] *convolutional* filters from models CNNP$^{one-hot}_{(3)_2}$ and CNNP$^{one-hot}_{(5)_2}$ are reused for transfer learning model shown in Figure 2.

with transfer learning from CNNP$^{one-hot}_{(3)_2}$ and CNNP$^{one-hot}_{(5)_2}$ models achieves the best attacking result. It achieves rank 2 of the correct key with only 40 traces. Promising attack results are also received from the CNNP models with *convolutional* filter kernel size 3 with 3-layer CNN, which achieve a key rank of 3 after only with 40 traces. We can see that there is nearly no difference in the attack result of those models, where the *plaintext extension* layer located before or after the first *fully-connected* layer. The worst result among proposed CNNP models occurs with a *convolutional* filter kernel size of 5, which achieves key rank 5 and 3 with 40 and 60 traces, respectively. Empirically, it is found that results get worse with an increase in kernel size, hence experiments with a kernel size of 7 on the ASCAD$^{variable}_{sync}$ database is given in Figure 9 in Section 7 only as reference.



(a) Key ranking comparison between proposed CNNP models and reference from ASCAD$^{variable}_{desync50}$ databases.



(b) Key ranking comparison among proposed CNNP models on attacking ASCAD$^{variable}_{desync100}$ databases.

Figure 5: Key ranking comparison for CNNP models trained on the ASCAD$^{variable}_{desync}$ database with three *convolutional* filter kernel sizes 3, 5 and 7 [a]. Keys are ranked by maximum likelihood score. The green line are reference models from the ASCAD database as described in Subsection 3.2.

[a]As the model with *convolutional* filter kernel size 5 shows better result than that with *convolutional* filter kernel size 3 on attacking ASCAD$^{variable}_{desync}$, CNNP model with *convolutional* filter kernel size 7 is added to this experiment.

5.2.2 Evaluation on ASCAD Variable Key De-Synchronised Dataset

Figure 5 shows the results of our proposed CNNP models with kernel sizes of 3, 5 and 7 when attacking ASCAD$_{desync}^{variable}$ data. Experiments were extended to CNNP models with kernel size 7 as experiments showed an advantage of size 5 over size 3, due the the desynchronisation of the traces. Figure 5a shows key ranking of our models in comparison with that given in [PSB+18] in attacking ASCAD$_{desync50}^{variable}$ data. It can be seen that our proposed CNNP models significantly outperform the reference one. The best result is achieved by the CNNP model with kernel sizes 3 and 5 in parallel, in which rank 4 of the correct key can be achieved with 500 traces, reducing to 2 after 1,000 traces. CNNP models with kernel sizes 5 and 7 are similar in terms of key rank in general, achieving key ranks 4 and 5 with 1,000 traces, respectively. The model with a kernel of sizes 3 does not work as well as previously with the synchronized data, only achieving a key rank of 20 after 1,000 traces.

As an additional benchmark for future works, we evaluate the models CNNP$_{(3)}^{one-hot}$, CNNP$_{(5)}^{one-hot}$, CNNP$_{(7)}^{one-hot}$ and CNNP$_{(3,\ 5)}^{one-hot}$ on the ASCAD$_{desync100}^{variable}$ database. The results shown in Figure 5b show that our proposed models can work on data with greater desynchronisation also. Outperform of CNNP model with *convolutional* filter kernel size 7 over other CNNP models in attacking ASCAD$_{desync100}^{variable}$ data suggests that bigger *convolutional* filter kernel size should be used to over come the deduction in key ranking caused by high de-synchronization.

5.2.3 Cross Evaluation on ASCAD Variable Key Datasets

Figure 6 shows the key ranking comparison between our proposed CNNP modes and reference of [PSB+18] in training on synchronized traces and attacking on desynchronized traces. Here the reference model works better than ours as it can obtain a key rank of 10 while the best result for our models in the same situation is key rank 50. This is likely due to the larger kernel size of 11 that is used.

As shown in Figure 7, the situation has changed when all models are trained on ASCAD$_{desync50}^{variable}$ and attack on the synchronized traces. Our combined CNNP model is compatible with the reference. It is able to reveal the key with around 300 traces. In addition, all our CNNP models outperform the reference in attacking *desync100* traces, in which the CNNP model with two parellel convolutional stages achieves a key rank of 4 with 800 *desync100* attack traces. The figure also shows that in this case, our proposed CNNP models with kernel sizes of 5 and 7 work better than that of size 3.

Figure 8 shows the cross evaluation for our models trained on ASCAD$_{desync100}^{variable}$ traces. It again confirms that the models with larger kernel sizes such as 5 or 7 work better than that with size 3 with *desync* data. The combined model performs best in cross evaluation, giving better results when attacking both *sync* and *desync50* traces.

6 Discussion

6.1 Effect of Convolutional Layers

As shown in this work and the previous literature, in general *convolutional* layers act as filters for SCA and allow generic detection of features that leak secret information. As they combine a greater number of time series samples as the number of layers increase, they are relatively robust to trace misalignment, similar to the way they allow translation invariance in image recognition problems. Hence, relevant features can be found even when located across the trace, and can subsequently be combined by the *fully-connected* layers. Therefore *convolutional* layers help DL models deal with trace misalignment based

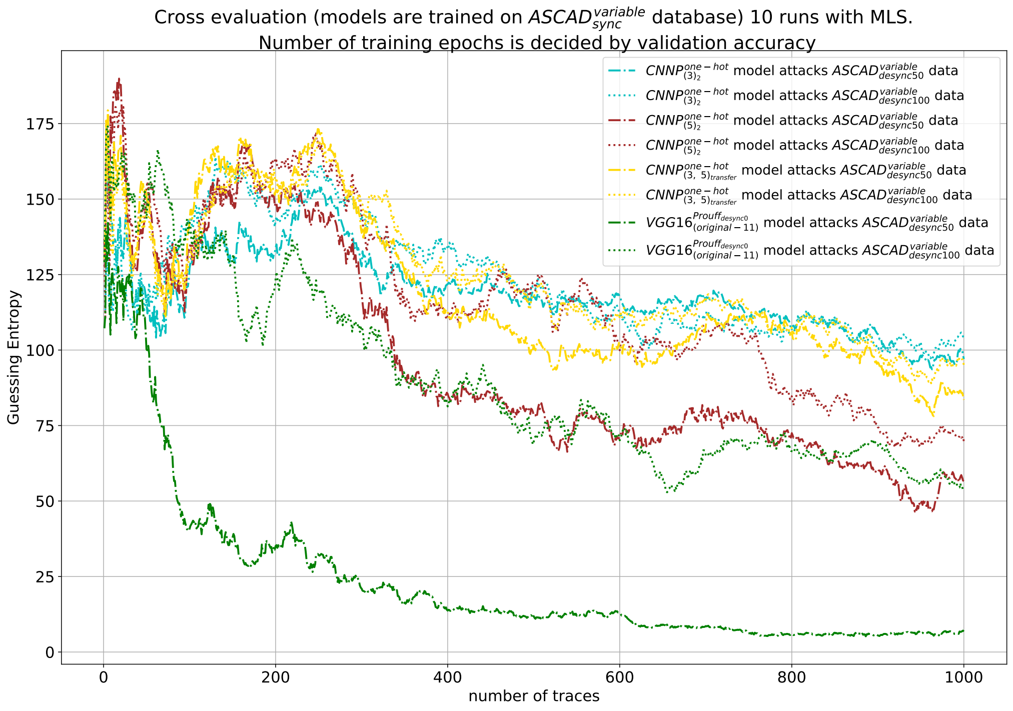


Figure 6: Attacking ASCAD$_{desync50}^{variable}$ and ASCAD$_{desync100}^{variable}$ traces using CNNP models trained on ASCAD$_{sync}^{variable}$ traces.

countermeasures such as unstable clocks, random hardware interrupts, clock stealing (in hardware), random delays insertion, shuffling *etc.*

6.2 Effect of Plaintext Feature Extension and Encoding Methods

Encoding plaintext message data and passing it as an input significantly improves the accuracy of the model. Two different encoding methods were used, with secret key recovery from the ASCAD^{fixed} dataset in 2 or fewer traces when using the synchronised traces with both methods, while on the ASCAD^{variable} dataset using the one-hot plaintext encoding had a significant advantage. Overall, using the one-hot encoding method for plaintext feature extension obtains better results with fewer attack traces. This is as there isn't necessarily a categorical increase in power consumption relative to the input plaintext value. Note this is in line with deep learning works around natural language processing [YHPC18] and word embedding where one-hot encoding has been proven to be an effective encoding method.

The effectiveness of using the plaintext as an input is dependent on the CNN architecture. In general, in an effective network structure like the proposed CNNP, adding plaintext knowledge into the model can improve the rank and convergence speed of the correct key, particularly for models with small kernel sizes. This can be seen in the improvement in the results from model VGG16P$_{(3)}^{one-hot}$ in comparison with the other VGG16 models in Figure 4a, and in the CNNP$^{one-hot}$ models in comparison with CNN$_{(3)}$ model in Figure 4b, where 40 traces are enough for CNNP$^{one-hot}$ model to reveal the correct key compared to 70 traces required for CNN$_{(3)}$ model. Where a sufficient number of attack traces are available (more than 100), there is little difference in the key rank with and without plaintext knowledge. However, in a narrow but wide CNN model with a large kernel

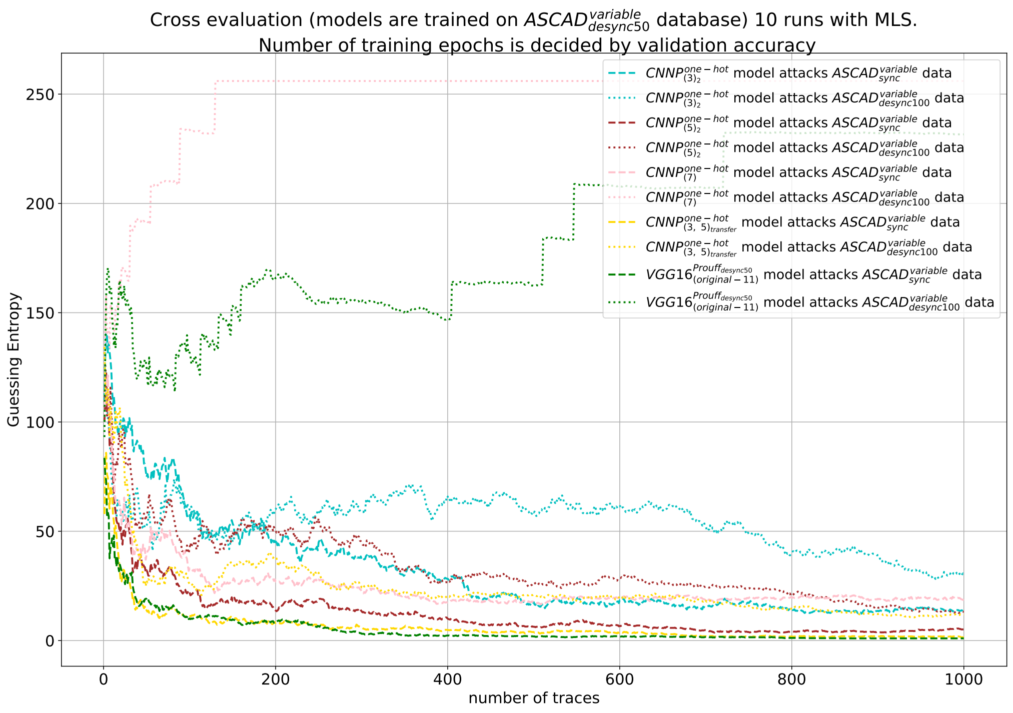


Figure 7: Attacking ASCAD$^{variable}_{sync}$ and ASCAD$^{variable}_{desync100}$ traces using CNNP models trained on ASCAD$^{variable}_{desync50}$ traces.

size like the original VGG16 model, incorporating the *plaintext feature* with the model reduces attacking accuracy. The key rank comparison between VGG16$^{Prouff}_{(original-11)}$ and VGG16P$^{one-hot}_{(original-11)}$, which is from the original work [PSB+18] without/with the *plaintext extension* layer included in Figure 4a, shows that it is not always advantageous to add the plaintext as an additional feature.

6.3 Effect of Convolutional Filter Kernel Sizes

When attacking the traces in the ASCAD^{fixed} database, the *convolutional* filter size, or combining multiple filters in CNNP$_{\mathrm{multi}}$, unsurprisingly has little effect on the results as the model is simply learning the bijection between the plaintext and SBox output given the key is fixed. When targeting the ASCAD$^{variable}_{sync}$ database, the smaller filter size of 3 has an advantage over the longer ones (5 and 7). However, experiments on the artifically desynchronized data shows that the longer the kernel size is, the better the performance. Intuitively this is to be expected, as the relavent features are no longer aligned in time across traces. Both the sampling rate and clock frequency will also have an impact on the optimal kernel size for a given dataset, hence it is advantageous to combine multiple parallel convolutional stages to improve the performance as can be seen in the CNNP$^{one-hot}_{(3,\ 5)_2}$ model.

6.4 Effects of Plaintext Extension Locations and DL Network Structures

In order to further understand about how the depth of network structure, the location of the *plaintext extension* layer, and the combination of various kenel lengths effect the output, we

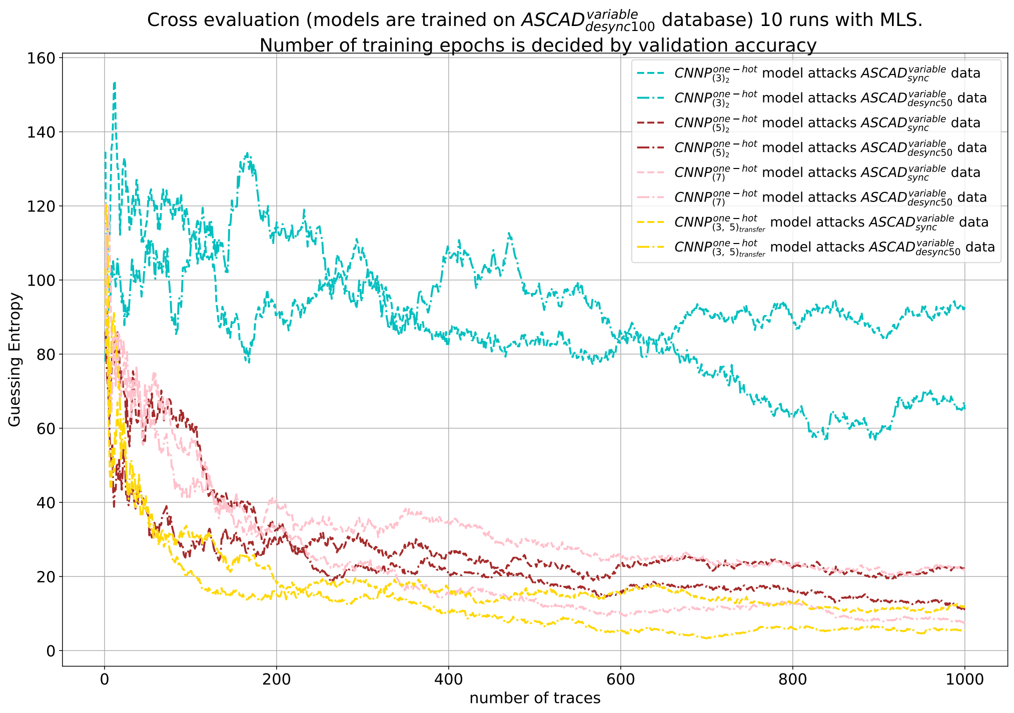


Figure 8: Attacking $\mathrm{ASCAD}^{variable}_{sync}$ and $\mathrm{ASCAD}^{variable}_{desync50}$ traces using CNNP models trained on $\mathrm{ASCAD}^{variable}_{desync100}$ traces.

need a combined analysis of the results given in Figure 4 and Figure 9. The results obtained by the $\mathrm{CNN}_{(3)}$ and $\mathrm{VGG16}^{Prouff}_{(original-11)}$ models are considered the baselines for comparing between our CNN network structures and CNNP, models with other approacheds developed using a similar domain knowledge approach. The network structures can be categorized into three groups: CNNP related (this work), VGG16 related ([PSB$^+$18]), and Hettwer ([HGG18]) models.

Precise location of the *plaintext extension* layer does not appear to be of paramount importance in our network structure, as there is minimal difference in the key rank between the two $\mathrm{CNNP}^{one-hot}_{(3)}$ versions; in which "Pext loc.1" means before the first *fully-connected* layer and "Pext loc.2" means before the second *fully-connected* layer. In addition, increasing the number of nodes per *fully-connected* layer also little effect as no significant improvement can be seen between the results using the $\mathrm{CNNP}^{one-hot}_{(3)_3}$ in comparison with that of $\mathrm{CNNP}^{one-hot}_{(3)_2}$. However, changing the structure of *convolutional* layers does effect the result, seen as a slight decreases in attacking accuracy of $\mathrm{CNNP}^{one-hot}_{(3)_4}$ model.

In term of network structure, when compared with "VGG16" [PSB$^+$18] our proposed CNNP with a deeper but narrower network structure results in a significantly improved mean rank of the correct key as can be seen in $\mathrm{CNN}_{(3)}$ (3-layer CNN - no Plaintext extension). The straightfoward $\mathrm{Hettwer}^{5-layerCNN}_{(original-8)}$ (Hettwer original) model in [HGG18], in which plaintext knowledge is taken into account, and $\mathrm{Hettwer}^{5-layerCNN}_{(3)}$ (with a kernel size of 3) have little success in attacking the higher-order protected AES implementation given by [PSB$^+$18], as can be seen in Figure 4a.

The effectiveness of different network widths when combined with different kernel lengths can be seen by the results of the $\mathrm{CNNP}^{one-hot}_{(3,\ 5)}$ models in Figure 9, which use the same structure in Figure 2 with *convolutional* filter sizes of 3 and 5 but are different in width of 512, 1024, and 1536. Applying neural layers that are too wide (1536 nodes per *fully connected* layer) in the $\text{CNNP}^{one-hot}_{(3,\ 5)_2}$ model reduces the accuracy of the model relative to the narrower (1024) one. The best result is achieved using transfer learning to train the model $\text{CNNP}^{one-hot}_{(3,\ 5)_{transfer}}$ in Figure 2, in which *convolutional* filters are taken from $\text{CNNP}^{one-hot}_{(3)_2}$ and $\text{CNNP}^{one-hot}_{(5)_2}$ models. This result can be interpreted as follows; each dependent *convolutional* branch in the combination model converges with different number of epochs, hence training them together means that the learned weight parameters are sub optimal. Transfer learning allows the models to learn the best weights, while separately training the neural network to combine both convolutional branches and classification.

7 Conclusion and Future Work

This paper has proposed a modified approach for building CNN models for profiling SCAs. A layer called *plaintext feature extension* is presented, with two different implementation methods of one-hot and integer encoding to incorporate plaintext information when training the model. A new network structure, which is deeper but narrower than previous proposals is introduced for improved results over the state of the art. Finally, a modified parallel architecture is proposed to incorporate multiple convolutional kernel filter lengths and layer structures. The proposed CNNP models show significant improvement in training and attacking the open-source ASCAD database, in which key recovery from 40 traces is possible when targeting the variable key database.

While considerable work has been done in the area of ML for SCA, there is still significant scope for improvement in current approaches. The experimentation in this work is conducted on traces acquired from an embedded device with software based countermeasures. Future scope of this work would be an analysis of how the plaintext embedding approach transfers to hardware based countermeasures such as threshold or dual-rail logic approaches.

Acknowledgements

This work has been supported by EPSRC via grant EP/R012288/1, under the RISE (http://www.ukrise.org) programme. The authors would like to thank the anonymous reviewers, as well as Dr Sarah McCarthy, for their valuable comments which helped to improve this work.

References

[AJR+18] Md. Zahangir Alom, Theodore Josue, Md Nayim Rahman, Will Mitchell, Chris Yakopcic, and Tarek M. Taha. Deep versus wide convolutional neural networks for object recognition on neuromorphic system. *CoRR*, abs/1802.02608, 2018.

[BC14] Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2654–2662. Curran Associates, Inc., 2014.

[BCD+17] Ryad Benadjila, Eleonora Cagli, Cécile Dumas, Emmanuel Prouff, Rémi Strullu, and Adrian Thillard. The ATMega8515 SCA traces databases. https://github.com/ANSSI-FR/ASCAD/tree/master/ATMEGA_AES_v1, 2017. [Online; accessed 15-Oct-2019].

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2004.

[Cor14] Jean-Sébastien Coron. Higher order masking of look-up tables. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology - EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 441–458. Springer, 2014.

[CZ06] Zhimin Chen and Yujie Zhou. Dual-rail random switching logic: A countermeasure to reduce side channel leakage. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006*, volume 4249 of *Lecture Notes in Computer Science*, pages 242–254. Springer, 2006.

[Das17] Siddharth Das. CNN Architectures: LeNet, AlexNet, VGG, GoogLeNet, ResNet and more.... https://medium.com/analytics-vidhya/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5, 2017. [Online; accessed 15-Oct-2019].

[GHO15] Richard Gilmore, Neil Hanley, and Máire O'Neill. Neural network based attack on a masked implementation of AES. In *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2015, Washington, DC, USA, 5-7 May, 2015*, pages 106–111. IEEE Computer Society, 2015.

[GHT05] Catherine H. Gebotys, Simon Ho, and C. C. Tiu. EM analysis of rijndael and ECC on a wireless java-based PDA. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2005.

[GM11] Louis Goubin and Ange Martinelli. Protecting AES with shamir's secret sharing scheme. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 79–94. Springer, 2011.

[GT02] Jovan Dj. Golic and Christophe Tymen. Multiplicative masking and power analysis of AES. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 198–212. Springer, 2002.

[HF12] Anh-Tuan Hoang and Takeshi Fujino. Intra-masking dual-rail memory on lut implementation for tamper-resistant aes on fpga. *ACM Transactions on Reconfigurable Technology and Systems*, 7:1–10, 02 2012.

[HGG18] Benjamin Hettwer, Stefan Gehrer, and Tim Güneysu. Profiled power analysis attacks using convolutional neural networks with domain knowledge. In Carlos Cid and Michael J. Jacobson Jr., editors, *Selected Areas in Cryptography - SAC 2018*, volume 11349 of *Lecture Notes in Computer Science*, pages 479–498. Springer, 2018.

[HJZ12] Hera He, Josh Jaffe, and Long Zou. Side Channel Cryptanalysis Using Machine Learning : Using an SVM to recover DES keys from a smart card. http://cs229.stanford.edu/proj2012/HeJaffeZou-SideChannelCryptanalysisUsingMachineLearning.pdf, 2012. [Online; accessed 28-Nov-2013].

[Kar19] Andrej Karpathy. Stanford cs231n - convolutional neural networks for visual recognition. http://cs231n.github.io/convolutional-networks/, 2019. [Online; accessed 14-Oct-2019].

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, pages 388–397, Berlin, Heidelberg, 1999. Springer-Verlag.

[Koc96] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 104–113, London, UK, UK, 1996. Springer-Verlag.

[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 Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):148–179, May 2019.

[LBM15] Liran Lerman, Gianluca Bontempi, and Olivier Markowitch. A machine learning approach against a masked AES - reaching the limit of side-channel attacks with a learning model. *J. Cryptogr. Eng.*, 5(2):123–139, 2015.

[LPB$^+$15] Liran Lerman, Romain Poussier, Gianluca Bontempi, Olivier Markowitch, and François-Xavier Standaert. Template attacks vs. machine learning revisited and the curse of dimensionality in side-channel analysis. In *Revised Selected Papers of the 6th International Workshop on Constructive Side-Channel Analysis and Secure Design - Volume 9064*, COSADE 2015, pages 20–33, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[Mag19] Houssem Maghrebi. Deep learning based side channel attacks in practice. Cryptology ePrint Archive, Report 2019/578, 2019. https://eprint.iacr.org/2019/578.

[MDP19] Loïc Masure, Cécile Dumas, and Emmanuel Prouff. Gradient visualization for general characterization in profiling attacks. In Ilia Polian and Marc Stöttinger, editors, *Constructive Side-Channel Analysis and Secure Design - 10th International Workshop, COSADE 2019, Darmstadt, Germany, April 3-5, 2019, Proceedings*, volume 11421 of *Lecture Notes in Computer Science*, pages 145–167. Springer, 2019.

[MPP16] Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. Breaking cryptographic implementations using deep learning techniques. Cryptology ePrint Archive, Report 2016/921, 2016. https://eprint.iacr.org/2016/921.

[MZVT16] Zdenek Martinasek, Ondrej Zapletal, Kamil Vrba, and Krisztina Trasy. Profiling power analysis attack based on mlp in dpa contest v4.2. In *2016 39th International Conference on Telecommunications and Signal Processing (TSP)*, pages 223–226, June 2016.

[NSGD12] Maxime Nassar, Youssef Souissi, Sylvain Guilley, and Jean-Luc Danger. Rsm: A small and fast countermeasure for aes, secure against 1st and 2nd-order zero-offset scas. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, pages 1173–1178, San Jose, CA, USA, 2012. EDA Consortium.

[PEC19] Guilherme Perin, Baris Ege, and Lukasz Chmielewski. Neural network model assessment for side-channel analysis. Cryptology ePrint Archive, Report 2019/722, 2019. https://eprint.iacr.org/2019/722.

[PM05] Thomas Popp and Stefan Mangard. Masked dual-rail pre-charge logic: Dpa-resistance without routing constraints. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, pages 172–186, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[PSB$^+$18] Emmanuel Prouff, Remi Strullu, Ryad Benadjila, Eleonora Cagli, and Cecile Dumas. Study of deep learning techniques for side-channel analysis and introduction to ascad database. Cryptology ePrint Archive, Report 2018/053, 2018. https://eprint.iacr.org/2018/053.

[PSK$^+$18] Stjepan Picek, Ioannis Petros Samiotis, Jaehun Kim, Annelie Heuser, Shivam Bhasin, and Axel Legay. On the performance of convolutional neural networks for side-channel analysis. In Anupam Chattopadhyay, Chester Rebeiro, and Yuval Yarom, editors, *Security, Privacy, and Applied Cryptography Engineering - 8th International Conference, SPACE 2018*, volume 11348 of *Lecture Notes in Computer Science*, pages 157–176. Springer, 2018.

[RBN$^+$15] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating masking schemes. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015*, volume 9215 of *Lecture Notes in Computer Science*, pages 764–783. Springer, 2015.

[Sah18] Sumit Saha. A comprehensive guide to convolutional neural networks — the eli5 way. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53, 2018. [Online; accessed 14-Oct-2019].

[SLJ$^+$14] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[SMY09] François-Xavier Standaert, Tal G. Malkin, and Moti Yung. A unified framework for the analysis of side-channel key recovery attacks. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009*, pages 443–461, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[SZ15] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

[TAV02] Kris Tiri, Moonmoon Akmal, and Ingrid Verbauwhede. A dynamic and differential cmos logic with signal independent power consumption to withstand differential power analysis on smart cards. In *Proceedings of the 28th European Solid-State Circuits Conference*, pages 403–406, 2002.

[Tim19] Benjamin Timon. Non-profiled deep learning-based side-channel attacks with sensitivity analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(2):107–131, Feb. 2019.

[TV04] Kris Tiri and Ingrid Verbauwhede. A logic level design methodology for a secure dpa resistant asic or fpga implementation. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '04, pages 10246–, Washington, DC, USA, 2004. IEEE Computer Society.

[vdVP19] Daan van der Valk and Stjepan Picek. Bias-variance decomposition in machine learning-based side-channel analysis. Cryptology ePrint Archive, Report 2019/570, 2019. https://eprint.iacr.org/2019/570.

[WMM19] Felix Wegener, Thorben Moos, and Amir Moradi. Dl-la: Deep learning leakage assessment: A modern roadmap for sca evaluations. Cryptology ePrint Archive, Report 2019/505, 2019. https://eprint.iacr.org/2019/505.

[WPB19] Leo Weissbart, Stjepan Picek, and Lejla Batina. One trace is all it takes: Machine learning-based side-channel attack on eddsa. Cryptology ePrint Archive, Report 2019/358, 2019. https://eprint.iacr.org/2019/358.

[YHPC18] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent trends in deep learning based natural language processing [review article]. *IEEE Computational Intelligence Magazine*, 13:55–75, 08 2018.

Appendix

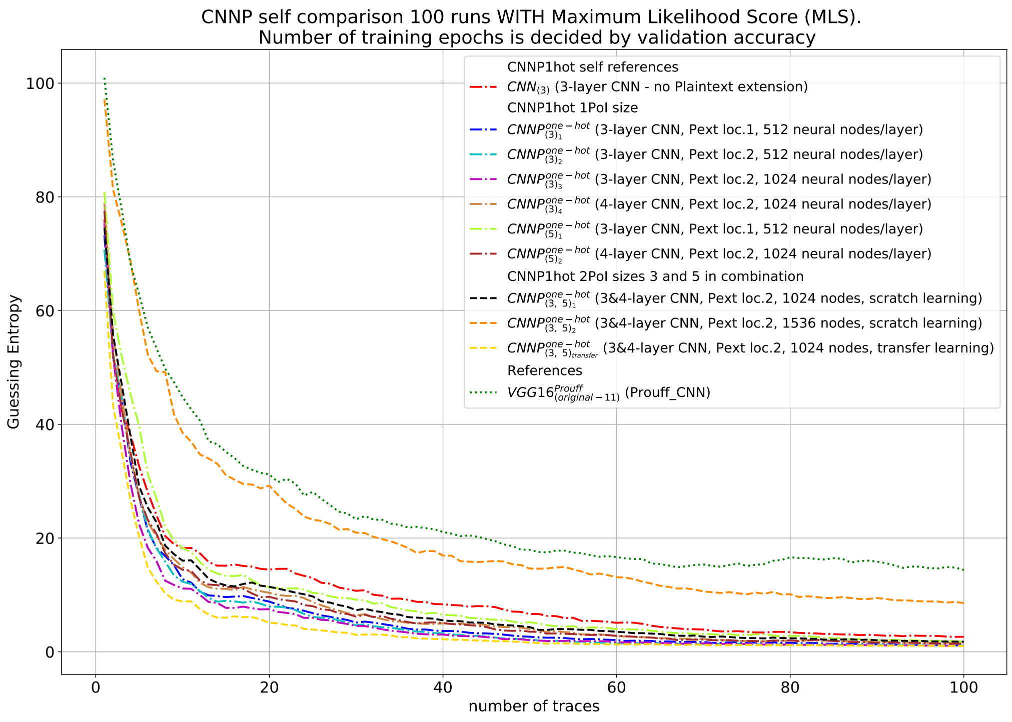


Figure 9: Key ranking comparison among proposed CNNP models on ASCAD$^{variable}_{sync}$ databases.

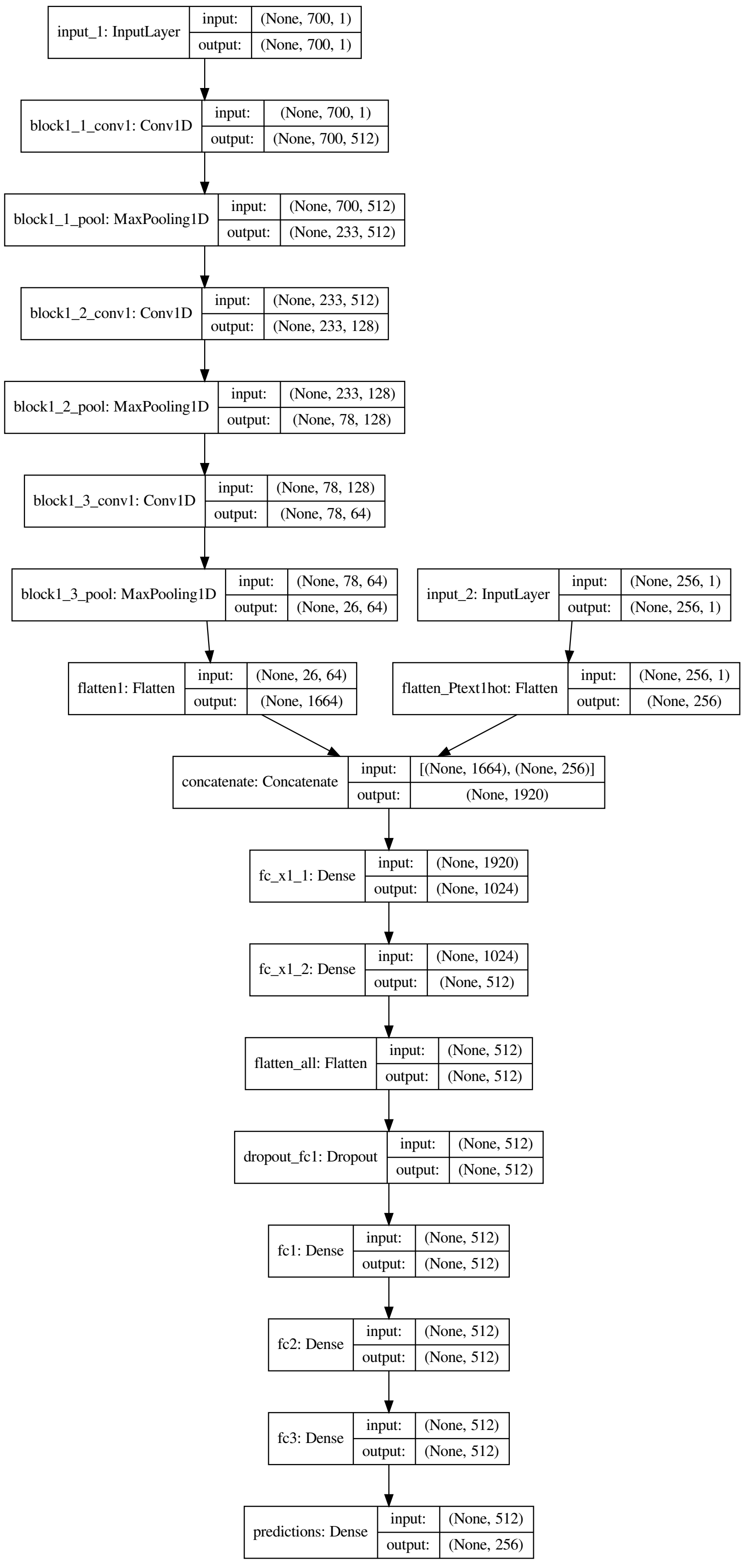

input_1: InputLayer | input: (None, 700, 1) | output: (None, 700, 1)
block1_1_conv1: Conv1D | input: (None, 700, 1) | output: (None, 700, 512)
block1_1_pool: MaxPooling1D | input: (None, 700, 512) | output: (None, 233, 512)
block1_2_conv1: Conv1D | input: (None, 233, 512) | output: (None, 233, 128)
block1_2_pool: MaxPooling1D | input: (None, 233, 128) | output: (None, 78, 128)
block1_3_conv1: Conv1D | input: (None, 78, 128) | output: (None, 78, 64)
block1_3_pool: MaxPooling1D | input: (None, 78, 64) | output: (None, 26, 64)
input_2: InputLayer | input: (None, 256, 1) | output: (None, 256, 1)
flatten1: Flatten | input: (None, 26, 64) | output: (None, 1664)
flatten_Ptext1hot: Flatten | input: (None, 256, 1) | output: (None, 256)
concatenate: Concatenate | input: [(None, 1664), (None, 256)] | output: (None, 1920)
fc_x1_1: Dense | input: (None, 1920) | output: (None, 1024)
fc_x1_2: Dense | input: (None, 1024) | output: (None, 512)
flatten_all: Flatten | input: (None, 512) | output: (None, 512)
dropout_fc1: Dropout | input: (None, 512) | output: (None, 512)
fc1: Dense | input: (None, 512) | output: (None, 512)
fc2: Dense | input: (None, 512) | output: (None, 512)
fc3: Dense | input: (None, 512) | output: (None, 512)
predictions: Dense | input: (None, 512) | output: (None, 256)

Figure 10: Detail of CNNP$^{one-hot}_{(single)}$ models for ASCAD^{fixed} database.

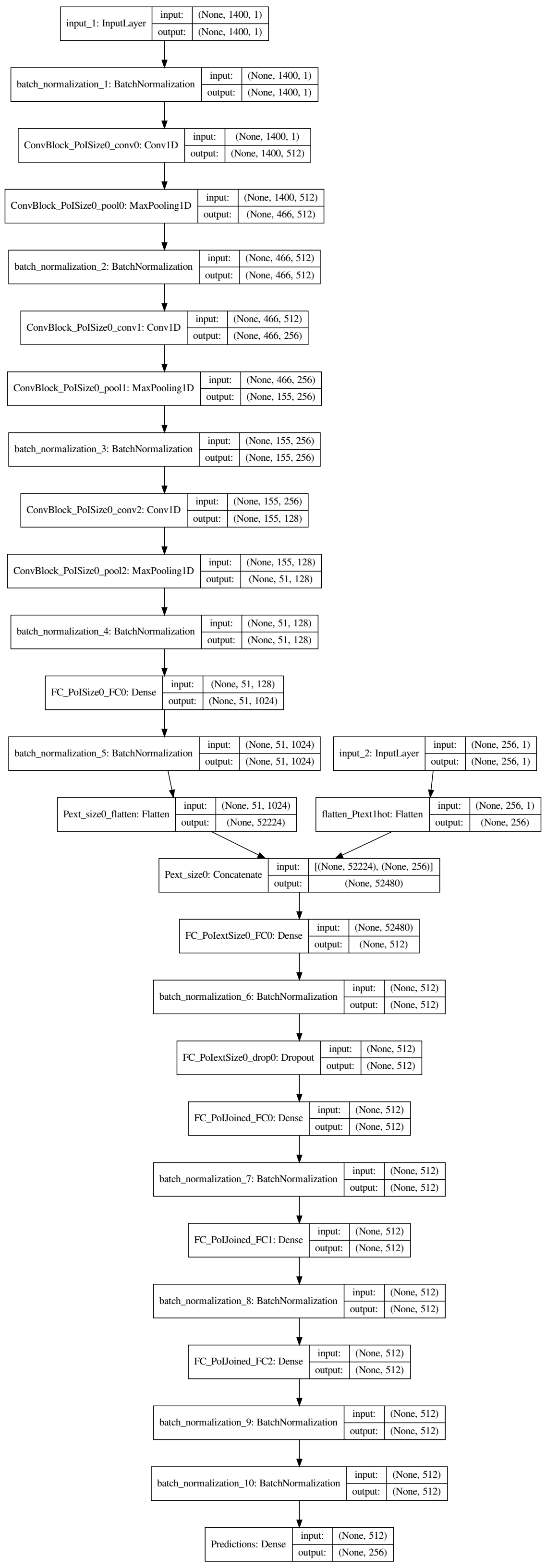

input_1: InputLayer | input: (None, 1400, 1) | output: (None, 1400, 1)
batch_normalization_1: BatchNormalization | input: (None, 1400, 1) | output: (None, 1400, 1)
ConvBlock_PoISize0_conv0: Conv1D | input: (None, 1400, 1) | output: (None, 1400, 512)
ConvBlock_PoISize0_pool0: MaxPooling1D | input: (None, 1400, 512) | output: (None, 466, 512)
batch_normalization_2: BatchNormalization | input: (None, 466, 512) | output: (None, 466, 512)
ConvBlock_PoISize0_conv1: Conv1D | input: (None, 466, 512) | output: (None, 466, 256)
ConvBlock_PoISize0_pool1: MaxPooling1D | input: (None, 466, 256) | output: (None, 155, 256)
batch_normalization_3: BatchNormalization | input: (None, 155, 256) | output: (None, 155, 256)
ConvBlock_PoISize0_conv2: Conv1D | input: (None, 155, 256) | output: (None, 155, 128)
ConvBlock_PoISize0_pool2: MaxPooling1D | input: (None, 155, 128) | output: (None, 51, 128)
batch_normalization_4: BatchNormalization | input: (None, 51, 128) | output: (None, 51, 128)
FC_PoISize0_FC0: Dense | input: (None, 51, 128) | output: (None, 51, 1024)
batch_normalization_5: BatchNormalization | input: (None, 51, 1024) | output: (None, 51, 1024)
input_2: InputLayer | input: (None, 256, 1) | output: (None, 256, 1)
Pext_size0_flatten: Flatten | input: (None, 51, 1024) | output: (None, 52224)
flatten_Ptext1hot: Flatten | input: (None, 256, 1) | output: (None, 256)
Pext_size0: Concatenate | input: [(None, 52224), (None, 256)] | output: (None, 52480)
FC_PoIextSize0_FC0: Dense | input: (None, 52480) | output: (None, 512)
batch_normalization_6: BatchNormalization | input: (None, 512) | output: (None, 512)
FC_PoIextSize0_drop0: Dropout | input: (None, 512) | output: (None, 512)
FC_PoIJoined_FC0: Dense | input: (None, 512) | output: (None, 512)
batch_normalization_7: BatchNormalization | input: (None, 512) | output: (None, 512)
FC_PoIJoined_FC1: Dense | input: (None, 512) | output: (None, 512)
batch_normalization_8: BatchNormalization | input: (None, 512) | output: (None, 512)
FC_PoIJoined_FC2: Dense | input: (None, 512) | output: (None, 512)
batch_normalization_9: BatchNormalization | input: (None, 512) | output: (None, 512)
batch_normalization_10: BatchNormalization | input: (None, 512) | output: (None, 512)
Predictions: Dense | input: (None, 512) | output: (None, 256)

Figure 11: Detail of CNNP$^{one-hot}_{(single)}$ models with 3 *convolutional* layers for ASCAD^{variable} database.

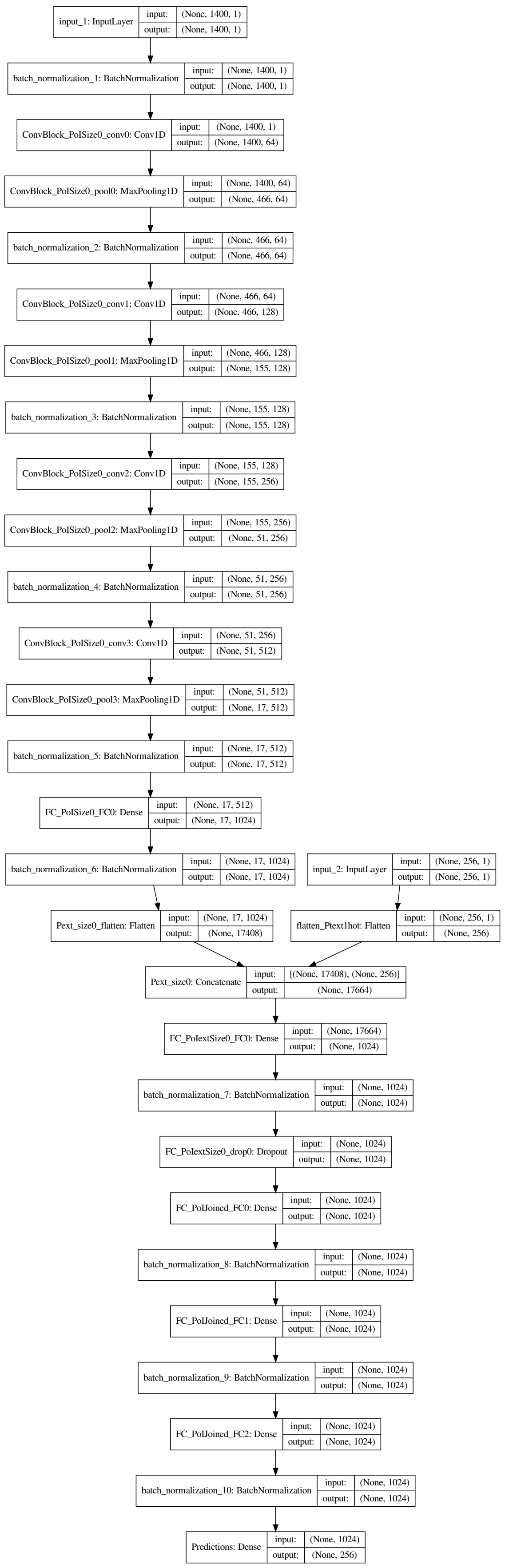

input_1: InputLayer | input: (None, 1400, 1) | output: (None, 1400, 1)
batch_normalization_1: BatchNormalization | input: (None, 1400, 1) | output: (None, 1400, 1)
ConvBlock_PoISize0_conv0: Conv1D | input: (None, 1400, 1) | output: (None, 1400, 64)
ConvBlock_PoISize0_pool0: MaxPooling1D | input: (None, 1400, 64) | output: (None, 466, 64)
batch_normalization_2: BatchNormalization | input: (None, 466, 64) | output: (None, 466, 64)
ConvBlock_PoISize0_conv1: Conv1D | input: (None, 466, 64) | output: (None, 466, 128)
ConvBlock_PoISize0_pool1: MaxPooling1D | input: (None, 466, 128) | output: (None, 155, 128)
batch_normalization_3: BatchNormalization | input: (None, 155, 128) | output: (None, 155, 128)
ConvBlock_PoISize0_conv2: Conv1D | input: (None, 155, 128) | output: (None, 155, 256)
ConvBlock_PoISize0_pool2: MaxPooling1D | input: (None, 155, 256) | output: (None, 51, 256)
batch_normalization_4: BatchNormalization | input: (None, 51, 256) | output: (None, 51, 256)
ConvBlock_PoISize0_conv3: Conv1D | input: (None, 51, 256) | output: (None, 51, 512)
ConvBlock_PoISize0_pool3: MaxPooling1D | input: (None, 51, 512) | output: (None, 17, 512)
batch_normalization_5: BatchNormalization | input: (None, 17, 512) | output: (None, 17, 512)
FC_PoISize0_FC0: Dense | input: (None, 17, 512) | output: (None, 17, 1024)
batch_normalization_6: BatchNormalization | input: (None, 17, 1024) | output: (None, 17, 1024)
input_2: InputLayer | input: (None, 256, 1) | output: (None, 256, 1)
Pext_size0_flatten: Flatten | input: (None, 17, 1024) | output: (None, 17408)
flatten_Ptext1hot: Flatten | input: (None, 256, 1) | output: (None, 256)
Pext_size0: Concatenate | input: [(None, 17408), (None, 256)] | output: (None, 17664)
FC_PoIextSize0_FC0: Dense | input: (None, 17664) | output: (None, 1024)
batch_normalization_7: BatchNormalization | input: (None, 1024) | output: (None, 1024)
FC_PoIextSize0_drop0: Dropout | input: (None, 1024) | output: (None, 1024)
FC_PoIJoined_FC0: Dense | input: (None, 1024) | output: (None, 1024)
batch_normalization_8: BatchNormalization | input: (None, 1024) | output: (None, 1024)
FC_PoIJoined_FC1: Dense | input: (None, 1024) | output: (None, 1024)
batch_normalization_9: BatchNormalization | input: (None, 1024) | output: (None, 1024)
FC_PoIJoined_FC2: Dense | input: (None, 1024) | output: (None, 1024)
batch_normalization_10: BatchNormalization | input: (None, 1024) | output: (None, 1024)
Predictions: Dense | input: (None, 1024) | output: (None, 256)

Figure 12: Detail of CNNP$^{one-hot}_{(single)}$ model with 4 *convolutional* layers for ASCAD^{variable} database.

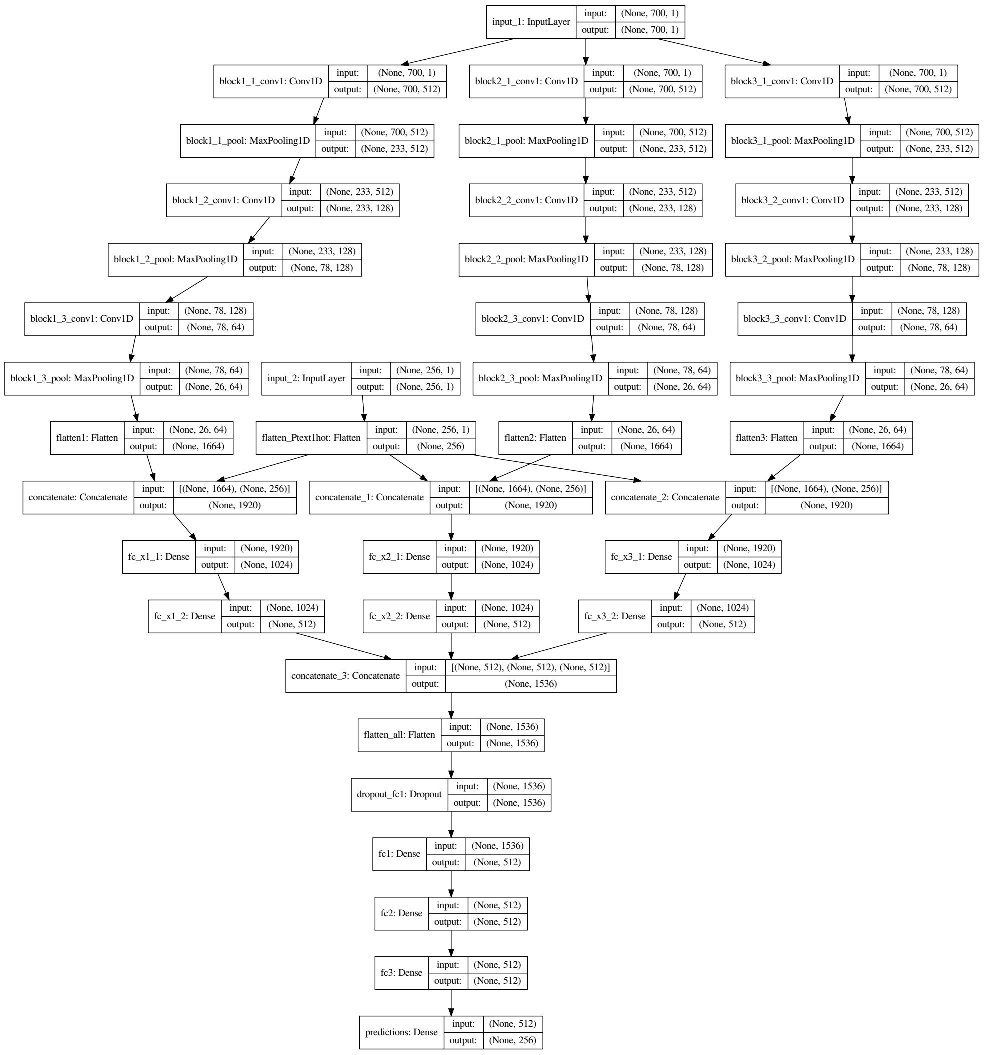

input_1: InputLayer | input: (None, 700, 1) | output: (None, 700, 1)
block1_1_conv1: Conv1D | input: (None, 700, 1) | output: (None, 700, 512)
block2_1_conv1: Conv1D | input: (None, 700, 1) | output: (None, 700, 512)
block3_1_conv1: Conv1D | input: (None, 700, 1) | output: (None, 700, 512)
block1_1_pool: MaxPooling1D | input: (None, 700, 512) | output: (None, 233, 512)
block2_1_pool: MaxPooling1D | input: (None, 700, 512) | output: (None, 233, 512)
block3_1_pool: MaxPooling1D | input: (None, 700, 512) | output: (None, 233, 512)
block1_2_conv1: Conv1D | input: (None, 233, 512) | output: (None, 233, 128)
block2_2_conv1: Conv1D | input: (None, 233, 512) | output: (None, 233, 128)
block3_2_conv1: Conv1D | input: (None, 233, 512) | output: (None, 233, 128)
block1_2_pool: MaxPooling1D | input: (None, 233, 128) | output: (None, 78, 128)
block2_2_pool: MaxPooling1D | input: (None, 233, 128) | output: (None, 78, 128)
block3_2_pool: MaxPooling1D | input: (None, 233, 128) | output: (None, 78, 128)
block1_3_conv1: Conv1D | input: (None, 78, 128) | output: (None, 78, 64)
block2_3_conv1: Conv1D | input: (None, 78, 128) | output: (None, 78, 64)
block3_3_conv1: Conv1D | input: (None, 78, 128) | output: (None, 78, 64)
block1_3_pool: MaxPooling1D | input: (None, 78, 64) | output: (None, 26, 64)
input_2: InputLayer | input: (None, 256, 1) | output: (None, 256, 1)
block2_3_pool: MaxPooling1D | input: (None, 78, 64) | output: (None, 26, 64)
block3_3_pool: MaxPooling1D | input: (None, 78, 64) | output: (None, 26, 64)
flatten1: Flatten | input: (None, 26, 64) | output: (None, 1664)
flatten_Ptext1hot: Flatten | input: (None, 256, 1) | output: (None, 256)
flatten2: Flatten | input: (None, 26, 64) | output: (None, 1664)
flatten3: Flatten | input: (None, 26, 64) | output: (None, 1664)
concatenate: Concatenate | input: [(None, 1664), (None, 256)] | output: (None, 1920)
concatenate_1: Concatenate | input: [(None, 1664), (None, 256)] | output: (None, 1920)
concatenate_2: Concatenate | input: [(None, 1664), (None, 256)] | output: (None, 1920)
fc_x1_1: Dense | input: (None, 1920) | output: (None, 1024)
fc_x2_1: Dense | input: (None, 1920) | output: (None, 1024)
fc_x3_1: Dense | input: (None, 1920) | output: (None, 1024)
fc_x1_2: Dense | input: (None, 1024) | output: (None, 512)
fc_x2_2: Dense | input: (None, 1024) | output: (None, 512)
fc_x3_2: Dense | input: (None, 1024) | output: (None, 512)
concatenate_3: Concatenate | input: [(None, 512), (None, 512), (None, 512)] | output: (None, 1536)
flatten_all: Flatten | input: (None, 1536) | output: (None, 1536)
dropout_fc1: Dropout | input: (None, 1536) | output: (None, 1536)
fc1: Dense | input: (None, 1536) | output: (None, 512)
fc2: Dense | input: (None, 512) | output: (None, 512)
fc3: Dense | input: (None, 512) | output: (None, 512)
predictions: Dense | input: (None, 512) | output: (None, 256)

Figure 13: Detail of CNNP$^{one-hot}_{multi}$ models for ASCAD^{fixed} database.

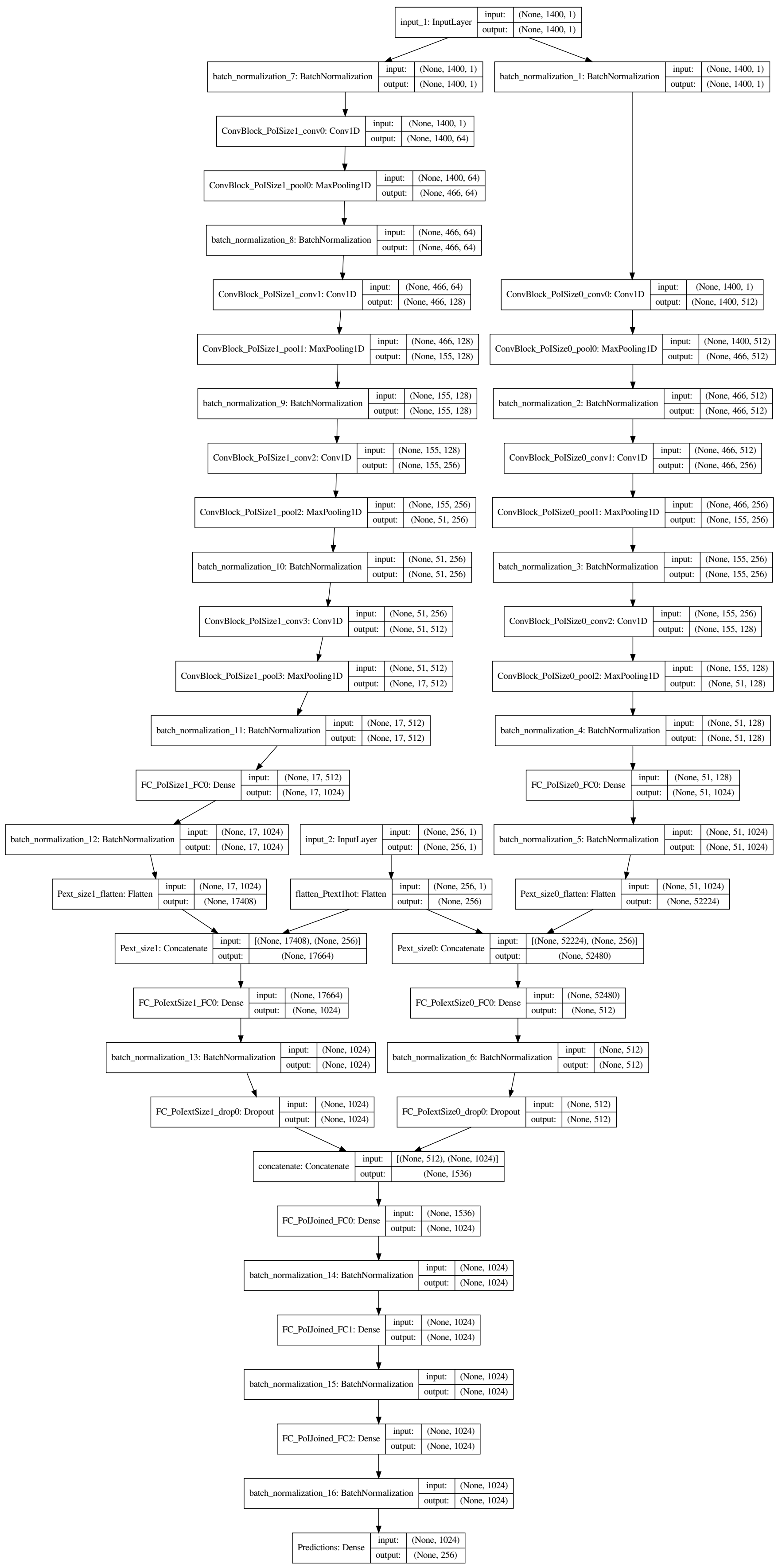

input_1: InputLayer | input: (None, 1400, 1) | output: (None, 1400, 1)
batch_normalization_7: BatchNormalization | input: (None, 1400, 1) | output: (None, 1400, 1)
batch_normalization_1: BatchNormalization | input: (None, 1400, 1) | output: (None, 1400, 1)
ConvBlock_PoISize1_conv0: Conv1D | input: (None, 1400, 1) | output: (None, 1400, 64)
ConvBlock_PoISize1_pool0: MaxPooling1D | input: (None, 1400, 64) | output: (None, 466, 64)
batch_normalization_8: BatchNormalization | input: (None, 466, 64) | output: (None, 466, 64)
ConvBlock_PoISize1_conv1: Conv1D | input: (None, 466, 64) | output: (None, 466, 128)
ConvBlock_PoISize0_conv0: Conv1D | input: (None, 1400, 1) | output: (None, 1400, 512)
ConvBlock_PoISize1_pool1: MaxPooling1D | input: (None, 466, 128) | output: (None, 155, 128)
ConvBlock_PoISize0_pool0: MaxPooling1D | input: (None, 1400, 512) | output: (None, 466, 512)
batch_normalization_9: BatchNormalization | input: (None, 155, 128) | output: (None, 155, 128)
batch_normalization_2: BatchNormalization | input: (None, 466, 512) | output: (None, 466, 512)
ConvBlock_PoISize1_conv2: Conv1D | input: (None, 155, 128) | output: (None, 155, 256)
ConvBlock_PoISize0_conv1: Conv1D | input: (None, 466, 512) | output: (None, 466, 256)
ConvBlock_PoISize1_pool2: MaxPooling1D | input: (None, 155, 256) | output: (None, 51, 256)
ConvBlock_PoISize0_pool1: MaxPooling1D | input: (None, 466, 256) | output: (None, 155, 256)
batch_normalization_10: BatchNormalization | input: (None, 51, 256) | output: (None, 51, 256)
batch_normalization_3: BatchNormalization | input: (None, 155, 256) | output: (None, 155, 256)
ConvBlock_PoISize1_conv3: Conv1D | input: (None, 51, 256) | output: (None, 51, 512)
ConvBlock_PoISize0_conv2: Conv1D | input: (None, 155, 256) | output: (None, 155, 128)
ConvBlock_PoISize1_pool3: MaxPooling1D | input: (None, 51, 512) | output: (None, 17, 512)
ConvBlock_PoISize0_pool2: MaxPooling1D | input: (None, 155, 128) | output: (None, 51, 128)
batch_normalization_11: BatchNormalization | input: (None, 17, 512) | output: (None, 17, 512)
batch_normalization_4: BatchNormalization | input: (None, 51, 128) | output: (None, 51, 128)
FC_PoISize1_FC0: Dense | input: (None, 17, 512) | output: (None, 17, 1024)
FC_PoISize0_FC0: Dense | input: (None, 51, 128) | output: (None, 51, 1024)
batch_normalization_12: BatchNormalization | input: (None, 17, 1024) | output: (None, 17, 1024)
input_2: InputLayer | input: (None, 256, 1) | output: (None, 256, 1)
batch_normalization_5: BatchNormalization | input: (None, 51, 1024) | output: (None, 51, 1024)
Pext_size1_flatten: Flatten | input: (None, 17, 1024) | output: (None, 17408)
flatten_Ptext1hot: Flatten | input: (None, 256, 1) | output: (None, 256)
Pext_size0_flatten: Flatten | input: (None, 51, 1024) | output: (None, 52224)
Pext_size1: Concatenate | input: [(None, 17408), (None, 256)] | output: (None, 17664)
Pext_size0: Concatenate | input: [(None, 52224), (None, 256)] | output: (None, 52480)
FC_PoIextSize1_FC0: Dense | input: (None, 17664) | output: (None, 1024)
FC_PoIextSize0_FC0: Dense | input: (None, 52480) | output: (None, 512)
batch_normalization_13: BatchNormalization | input: (None, 1024) | output: (None, 1024)
batch_normalization_6: BatchNormalization | input: (None, 512) | output: (None, 512)
FC_PoIextSize1_drop0: Dropout | input: (None, 1024) | output: (None, 1024)
FC_PoIextSize0_drop0: Dropout | input: (None, 512) | output: (None, 512)
concatenate: Concatenate | input: [(None, 512), (None, 1024)] | output: (None, 1536)
FC_PoIJoined_FC0: Dense | input: (None, 1536) | output: (None, 1024)
batch_normalization_14: BatchNormalization | input: (None, 1024) | output: (None, 1024)
FC_PoIJoined_FC1: Dense | input: (None, 1024) | output: (None, 1024)
batch_normalization_15: BatchNormalization | input: (None, 1024) | output: (None, 1024)
FC_PoIJoined_FC2: Dense | input: (None, 1024) | output: (None, 1024)
batch_normalization_16: BatchNormalization | input: (None, 1024) | output: (None, 1024)
Predictions: Dense | input: (None, 1024) | output: (None, 256)

Figure 14: Detail of CNNP$^{one-hot}_{(3,\ 5)}$ model for ASCAD^{variable} database.

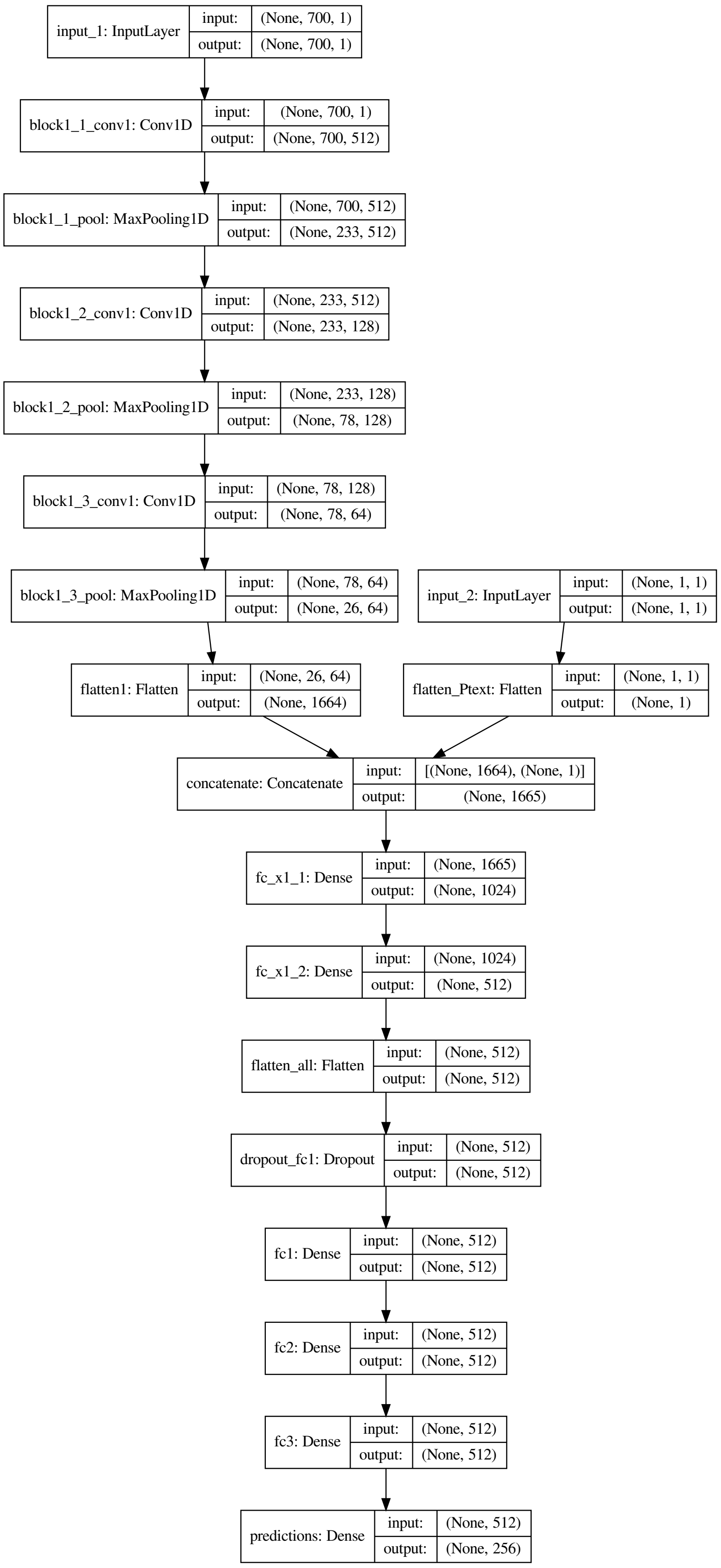

input_1: InputLayer | input: (None, 700, 1) | output: (None, 700, 1)
block1_1_conv1: Conv1D | input: (None, 700, 1) | output: (None, 700, 512)
block1_1_pool: MaxPooling1D | input: (None, 700, 512) | output: (None, 233, 512)
block1_2_conv1: Conv1D | input: (None, 233, 512) | output: (None, 233, 128)
block1_2_pool: MaxPooling1D | input: (None, 233, 128) | output: (None, 78, 128)
block1_3_conv1: Conv1D | input: (None, 78, 128) | output: (None, 78, 64)
block1_3_pool: MaxPooling1D | input: (None, 78, 64) | output: (None, 26, 64)
input_2: InputLayer | input: (None, 1, 1) | output: (None, 1, 1)
flatten1: Flatten | input: (None, 26, 64) | output: (None, 1664)
flatten_Ptext: Flatten | input: (None, 1, 1) | output: (None, 1)
concatenate: Concatenate | input: [(None, 1664), (None, 1)] | output: (None, 1665)
fc_x1_1: Dense | input: (None, 1665) | output: (None, 1024)
fc_x1_2: Dense | input: (None, 1024) | output: (None, 512)
flatten_all: Flatten | input: (None, 512) | output: (None, 512)
dropout_fc1: Dropout | input: (None, 512) | output: (None, 512)
fc1: Dense | input: (None, 512) | output: (None, 512)
fc2: Dense | input: (None, 512) | output: (None, 512)
fc3: Dense | input: (None, 512) | output: (None, 512)
predictions: Dense | input: (None, 512) | output: (None, 256)

Figure 15: Detail of CNNP$_{single}^{integer}$ models for ASCAD^{fixed} database.

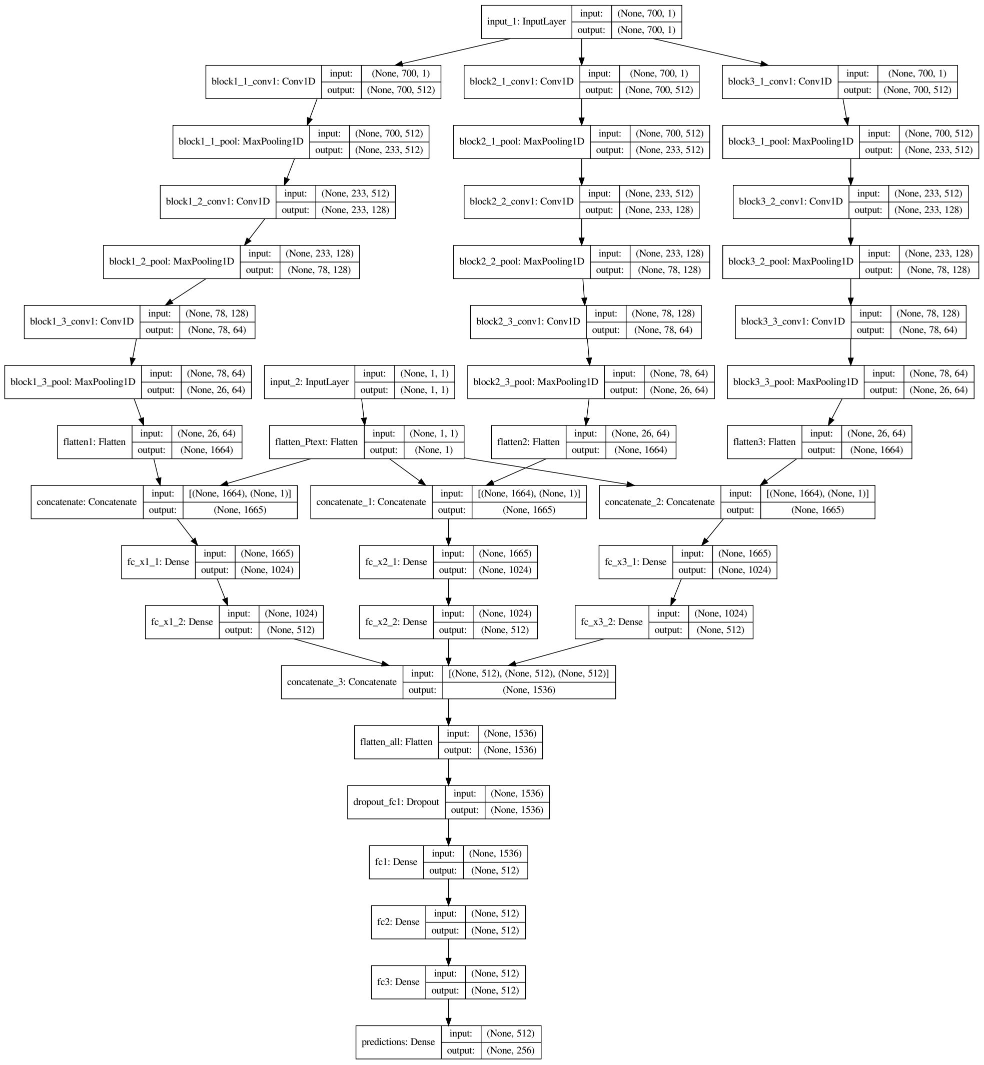

input_1: InputLayer input: (None, 700, 1) output: (None, 700, 1)
block1_1_conv1: Conv1D input: (None, 700, 1) output: (None, 700, 512)
block2_1_conv1: Conv1D input: (None, 700, 1) output: (None, 700, 512)
block3_1_conv1: Conv1D input: (None, 700, 1) output: (None, 700, 512)
block1_1_pool: MaxPooling1D input: (None, 700, 512) output: (None, 233, 512)
block2_1_pool: MaxPooling1D input: (None, 700, 512) output: (None, 233, 512)
block3_1_pool: MaxPooling1D input: (None, 700, 512) output: (None, 233, 512)
block1_2_conv1: Conv1D input: (None, 233, 512) output: (None, 233, 128)
block2_2_conv1: Conv1D input: (None, 233, 512) output: (None, 233, 128)
block3_2_conv1: Conv1D input: (None, 233, 512) output: (None, 233, 128)
block1_2_pool: MaxPooling1D input: (None, 233, 128) output: (None, 78, 128)
block2_2_pool: MaxPooling1D input: (None, 233, 128) output: (None, 78, 128)
block3_2_pool: MaxPooling1D input: (None, 233, 128) output: (None, 78, 128)
block1_3_conv1: Conv1D input: (None, 78, 128) output: (None, 78, 64)
block2_3_conv1: Conv1D input: (None, 78, 128) output: (None, 78, 64)
block3_3_conv1: Conv1D input: (None, 78, 128) output: (None, 78, 64)
block1_3_pool: MaxPooling1D input: (None, 78, 64) output: (None, 26, 64)
input_2: InputLayer input: (None, 1, 1) output: (None, 1, 1)
block2_3_pool: MaxPooling1D input: (None, 78, 64) output: (None, 26, 64)
block3_3_pool: MaxPooling1D input: (None, 78, 64) output: (None, 26, 64)
flatten1: Flatten input: (None, 26, 64) output: (None, 1664)
flatten_Ptext: Flatten input: (None, 1, 1) output: (None, 1)
flatten2: Flatten input: (None, 26, 64) output: (None, 1664)
flatten3: Flatten input: (None, 26, 64) output: (None, 1664)
concatenate: Concatenate input: [(None, 1664), (None, 1)] output: (None, 1665)
concatenate_1: Concatenate input: [(None, 1664), (None, 1)] output: (None, 1665)
concatenate_2: Concatenate input: [(None, 1664), (None, 1)] output: (None, 1665)
fc_x1_1: Dense input: (None, 1665) output: (None, 1024)
fc_x2_1: Dense input: (None, 1665) output: (None, 1024)
fc_x3_1: Dense input: (None, 1665) output: (None, 1024)
fc_x1_2: Dense input: (None, 1024) output: (None, 512)
fc_x2_2: Dense input: (None, 1024) output: (None, 512)
fc_x3_2: Dense input: (None, 1024) output: (None, 512)
concatenate_3: Concatenate input: [(None, 512), (None, 512), (None, 512)] output: (None, 1536)
flatten_all: Flatten input: (None, 1536) output: (None, 1536)
dropout_fc1: Dropout input: (None, 1536) output: (None, 1536)
fc1: Dense input: (None, 1536) output: (None, 512)
fc2: Dense input: (None, 512) output: (None, 512)
fc3: Dense input: (None, 512) output: (None, 512)
predictions: Dense input: (None, 512) output: (None, 256)

Figure 16: Detail of CNNP$_{multi}^{integer}$ models for ASCAD^{fixed} database.