Advanced Topics In Cybersecurity 2

Lab Session: Task 2

Imen sghaier, 12346138

Abstract

In the field of cryptography, ensuring the security of cryptographic algorithms is of paramount importance. However, even well-established algorithms such as the Advanced Encryption Standard (AES) can be vulnerable to side-channel attacks (SCA). These attacks exploit unintentional information leaks during the physical execution of a cryptographic algorithm, such as power consumption, electromagnetic emissions, or timing information. One of the most powerful approaches to defending against side-channel attacks is the use of machine learning techniques, particularly deep learning models, which can analyze and predict the secret key used in cryptographic operations by learning from the observed side-channel traces.

1 Introduction:

The objective of this project is to implement a deep learning-based solution using a CNN model inspired by the VGG16 architecture, a proven model in the field of image processing, adapted here to work with one-dimensional side-channel traces. The primary task is to train the model on profiling traces to learn the relationship between side-channel measurements and the cryptographic key. Once trained, the model is then used to predict the secret AES key byte by byte from attack traces, effectively performing a side-channel attack.

By utilizing the ASCAD dataset and a VGG16-inspired CNN, this project aims to demonstrate the feasibility and effectiveness of deep learning techniques for key recovery in side-channel analysis.

2 Implementation:

2.1 Data Preparation (Loading the Datasets):

the first critical step in implementing any machine learning model is the proper preparation of the dataset. In this project, i tried first to understand and analyses the structure of the datasets used for training and testing the Convocational Neural Network (CNN) which contains power traces corresponding to AES encryption operations. These traces are of paramount importance because they contain the information that the model will use to predict the AES key.

1. The load_dataset() function: is responsible for loading the entire dataset, including both profiling and attack traces. The ASCAD database is stored in an HDF5 file format, and the function opens this file and extracts the necessary components: profiling traces, profiling labels, attack traces, and attack labels.

2. Data Preprocessing: Normalization and Reshaping;

   Once the traces are loaded, they undergo normalization and reshaping. These preprocessing steps are crucial for ensuring that the data is in a suitable form for input into the CNN model.

   (a) Normalization:The raw power traces often have varying ranges and magnitudes due to different conditions during collection, such as environmental factors or measurement noise. To mitigate this variability, the traces are normalized by dividing them by their maximum absolute value. This scales all values between -1 and 1, ensuring that the model does not become biased by larger or smaller values in the data and that it can more easily learn the underlying patterns.

   (b) Reshaping: The CNN model expects the input data to be in a specific format. The traces, initially stored as 2D arrays (with shape ‘(num_traces, num_samples)‘), need to be reshaped to include a channel dimension, which is a common requirement for CNNs. This reshaping transforms the data into a 3D array of shape ‘(num_traces, num_samples, 1)‘, where ‘1‘ represents a single channel, allowing the convolutional layers to process the data effectively.

   NOTE: That was an initial script to load the entire data and understand its contents however in the coming steps the loaded traces are treated differently depending on the use .

2.2 Model Training (train_model.py):

The train_model.py script is responsible for training a Convolutional Neural Network (CNN) model to recover the AES key by analyzing the profiling traces, which contain side-channel information captured during cryptographic operations. The script leverages TensorFlow to train the model efficiently and utilizes GPU acceleration for faster computation, given the complex nature of deep learning tasks.

1. Data Preparation :
   The training data is extracted from the metadata within the profiling traces. Specifically, the profiling traces contain actual information about the AES key bytes and their corresponding plaintexts. Using the load_profiling_dataset() function, the script extracts the relevant key byte (from the target byte in the AES key) from the

metadata, which is then used to generate labels for training. The corresponding traces are normalized and reshaped to match the input requirements of the CNN. **NOTE:**i included a script called t**inspect labels .py** that helped me to understand how the labels are organized in the dataset so i could acess properly the information related to each keybyte by label

2. CNN Architecture:

   The architecture of the model is based on the VGG16 network, known for its deep convolutional layers that effectively extract hierarchical features from the input data. While originally designed for image classification, the VGG16 architecture was adapted to one-dimensional input, as the profiling traces are essentially 1D signals.

   NOTE: all the norms and the values of the architecture are based on the PDF "Study of Deep Learning Techniques for Side-Channel Analysis and Introduction to ASCAD Database" in the section the best CNNs

   The VGG16 model is structured as follows:

   - **Convolutional layers**: These layers learn to extract features from the input traces. Each block consists of multiple Conv1D layers followed by Batch Normalization and MaxPooling operations.
   - **MaxPooling layers**: These layers downsample the data, reducing its dimensionality while retaining key features.
   - **Dropout layers**: Dropout is applied to prevent overfitting by randomly dropping some neurons during training, forcing the model to learn more robust features.
   - **Fully connected layers**: The model ends with fully connected (Dense) layers, which ultimately predict the 256 possible values for the key byte using a softmax activation.

3. GPU Acceleration with TensorFlow:
   Given the computational demands of training such a deep CNN, the script is configured to take advantage of GPU acceleration using TensorFlow and CUDA libraries. TensorFlow is set to utilize the GPU for model training, which significantly speeds up the process compared to using a CPU. The script configures the necessary paths for CUDA and cuDNN libraries to enable TensorFlow to efficiently allocate GPU memory.
   Then By leveraging the GPU, the model can process the data much faster and

handle the complexities of training without running into memory limitations.

NOTE : a small script shows that the tensorflow is working properly by inside the file **check tensorflow.py** is included in my directory that comes with those result

```
C:\Users\sghai\PythonProject1\.venv\Scripts\python.exe "C:\Users\sghai\PythonPro
Updated PATH: C:\Users\sghai\PythonProject1\.venv\Scripts;C:\Program Files\NVIDI
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.2\bin\ptxas.exe
ptxas in PATH: 0
TensorFlow Version: 2.10.0
Built with CUDA: True
Num GPUs Available:  1
2025-01-28 14:36:25.370556: I tensorflow/core/platform/cpu_feature_guard.cc:193]
To enable them in other operations, rebuild TensorFlow with the appropriate comp
2025-01-28 14:36:25.998662: W tensorflow/core/common_runtime/gpu/gpu_bfc_allocat
2025-01-28 14:36:25.998859: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1
2025-01-28 14:36:26.854319: I tensorflow/stream_executor/cuda/cuda_blas.cc:1614]
Matrix a:
 [[1. 2.]
 [3. 4.]]
Matrix b:
 [[1. 1.]
 [0. 1.]]
Matrix c (a * b):
 [[1. 3.]
 [3. 7.]]

Process finished with exit code 0
```

4. Training Process :
   During the training process, the script loops through all 16 bytes of the AES key. For each byte, a separate model is trained using the profiling traces and their corresponding labels extracted from the metadata. The model is optimized using the RMSprop optimizer, with a small learning rate (1e-5) to ensure stable convergence.

   To avoid overfitting, the script utilizes EarlyStopping, which halts training if the validation loss does not improve after a set number of epochs (patience of 10). It also employs ModelCheckpoint, which saves the model whenever it achieves the best performance on the validation data, ensuring that only the best model is kept.

5. Outcomes:
   The expected outcome of this script is a trained CNN model for each of the 16

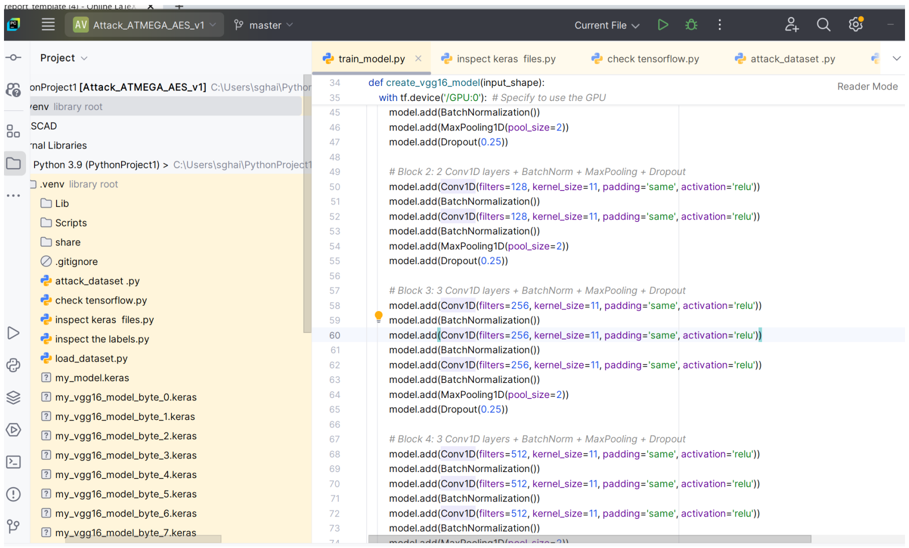
key bytes in the AES key registed as keras files. Each model will have learned to predict one byte of the AES key based on the profiling traces.
These models then are used in the attack_model.py script to perform key recovery attacks by analyzing attack traces.
Note: The training process took around 4 hours from me to get 16 keras files with a size of 625 kb each this is why i didn't included it into my zipped folder then you have to run first the train model with the specified requirements to get the results of the attack .

since we cannot directly open the keras files as a regular text file since it is a binary file, usually in HDF5 format in the SavedModel format (with the .keras extension), i included a script in the file **inspect keras file.py** to load and inspect the model contents as the structure, weights, and other attributes of a saved Keras model file, and that was a part from the result :

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv1d (Conv1D)             (None, 700, 64)           768

 batch_normalization (BatchN  (None, 700, 64)          256
 ormalization)

 conv1d_1 (Conv1D)           (None, 700, 64)           45120

 batch_normalization_1 (Batc  (None, 700, 64)          256
 hNormalization)

 max_pooling1d (MaxPooling1D  (None, 350, 64)          0
 )

 dropout (Dropout)           (None, 350, 64)           0

 conv1d_2 (Conv1D)           (None, 350, 128)          90240

 batch_normalization_2 (Batc  (None, 350, 128)         512
 hNormalization)

 conv1d_3 (Conv1D)           (None, 350, 128)          180352

 batch_normalization_3 (Batc  (None, 350, 128)         512
 hNormalization)
```

```
max_pooling1d_1 (MaxPooling  (None, 175, 128)         0
1D)

dropout_1 (Dropout)         (None, 175, 128)         0

conv1d_4 (Conv1D)           (None, 175, 256)         360704

batch_normalization_4 (Batc  (None, 175, 256)         1024
hNormalization)

conv1d_5 (Conv1D)           (None, 175, 256)         721152

batch_normalization_5 (Batc  (None, 175, 256)         1024
hNormalization)

conv1d_6 (Conv1D)           (None, 175, 256)         721152

batch_normalization_6 (Batc  (None, 175, 256)         1024
hNormalization)

max_pooling1d_2 (MaxPooling  (None, 87, 256)          0
1D)

dropout_2 (Dropout)         (None, 87, 256)          0

conv1d_7 (Conv1D)           (None, 87, 512)          1442304

batch_normalization_7 (Batc  (None, 87, 512)          2048
hNormalization)

conv1d_8 (Conv1D)           (None, 87, 512)          2884096

batch_normalization_8 (Batc  (None, 87, 512)          2048
hNormalization)

conv1d_9 (Conv1D)           (None, 87, 512)          2884096

batch_normalization_9 (Batc  (None, 87, 512)          2048
hNormalization)

max_pooling1d_3 (MaxPooling  (None, 43, 512)          0
 1D)

 dropout_3 (Dropout)         (None, 43, 512)           0

 conv1d_10 (Conv1D)          (None, 43, 512)           2884096

 batch_normalization_10 (Bat  (None, 43, 512)          2048
 chNormalization)

 conv1d_11 (Conv1D)          (None, 43, 512)           2884096

 batch_normalization_11 (Bat  (None, 43, 512)          2048
 chNormalization)

 conv1d_12 (Conv1D)          (None, 43, 512)           2884096

 batch_normalization_12 (Bat  (None, 43, 512)          2048
 chNormalization)

 max_pooling1d_4 (MaxPooling  (None, 21, 512)          0
 1D)

 dropout_4 (Dropout)         (None, 21, 512)           0

 flatten (Flatten)           (None, 10752)             0

 dense (Dense)               (None, 4096)              44044288

 dense_1 (Dense)             (None, 4096)              16781312

 dense_2 (Dense)             (None, 256)               1048832

=================================================================
Total params: 79,873,600
Trainable params: 79,865,152
Non-trainable params: 8,448
```

this screenshot of my local directory to show the structure of the trained models :



```
onProject1 [Attack_ATMEGA_AES_v1] C:\Users\sghai\Pythor
venv  library root
SCAD
rnal Libraries
Python 3.9 (PythonProject1) > C:\Users\sghai\PythonProject1
.venv  library root
    Lib
    Scripts
    share
    .gitignore
    attack_dataset .py
    check tensorflow.py
    inspect keras  files.py
    inspect the labels.py
    load_dataset.py
    my_model.keras
    my_vgg16_model_byte_0.keras
    my_vgg16_model_byte_1.keras
    my_vgg16_model_byte_2.keras
    my_vgg16_model_byte_3.keras
    my_vgg16_model_byte_4.keras
    my_vgg16_model_byte_5.keras
    my_vgg16_model_byte_6.keras
    my_vgg16_model_byte_7.keras
```

```python
def create_vgg16_model(input_shape):
    with tf.device('/GPU:0'):  # Specify to use the GPU
        model.add(BatchNormalization())
        model.add(MaxPooling1D(pool_size=2))
        model.add(Dropout(0.25))

        # Block 2: 2 Conv1D layers + BatchNorm + MaxPooling + Dropout
        model.add(Conv1D(filters=128, kernel_size=11, padding='same', activation='relu'))
        model.add(BatchNormalization())
        model.add(Conv1D(filters=128, kernel_size=11, padding='same', activation='relu'))
        model.add(BatchNormalization())
        model.add(MaxPooling1D(pool_size=2))
        model.add(Dropout(0.25))

        # Block 3: 3 Conv1D layers + BatchNorm + MaxPooling + Dropout
        model.add(Conv1D(filters=256, kernel_size=11, padding='same', activation='relu'))
        model.add(BatchNormalization())
        model.add(Conv1D(filters=256, kernel_size=11, padding='same', activation='relu'))
        model.add(BatchNormalization())
        model.add(Conv1D(filters=256, kernel_size=11, padding='same', activation='relu'))
        model.add(BatchNormalization())
        model.add(MaxPooling1D(pool_size=2))
        model.add(Dropout(0.25))

        # Block 4: 3 Conv1D layers + BatchNorm + MaxPooling + Dropout
        model.add(Conv1D(filters=512, kernel_size=11, padding='same', activation='relu'))
        model.add(BatchNormalization())
        model.add(Conv1D(filters=512, kernel_size=11, padding='same', activation='relu'))
        model.add(BatchNormalization())
        model.add(Conv1D(filters=512, kernel_size=11, padding='same', activation='relu'))
        model.add(BatchNormalization())
```

Figure 1: The saved trained models

2.3 Key Recovery (attack_dataset.py)

The attack process leverages the trained Convolutional Neural Network (CNN) models to recover the AES key from side-channel data.
The general workflow is as follows:

1. Loading the Attack Dataset: The attack script begins by loading the dataset containing the side-channel traces and associated metadata. Specifically, it loads the traces of the device under attack along with the corresponding plaintext and key for each byte from the `Attack_traces` section of the dataset. The plaintext and key are extracted for each target byte, and the traces are normalized to a range between -1 and 1 for efficient model input. The traces are reshaped to match the required input format for the CNN model (a 3D array, where the third dimension represents the single-channel data).

2. Model Loading: For each byte of the AES key (16 bytes in total), the pre-trained CNN model specific to that byte is loaded. These models were trained in the previous stage and saved using the Keras framework.

3. Prediction of Key Byte: With the dataset prepared and the model loaded, the traces are fed into the CNN model. The model predicts the probability distribution over all possible key values for each trace. The most likely key byte for each trace is determined by finding the index with the highest predicted probability.

4. Prediction of Key Byte: With the dataset prepared and the model loaded, the traces are fed into the CNN model. The model predicts the probability distribution over

all possible key values for each trace. The most likely key byte for each trace is determined by finding the index with the highest predicted probability.

5. Mode Calculation: To ensure accuracy and reliability, the predicted key bytes for all traces are aggregated, and the mode (the most frequent value) of the predictions is computed. The mode represents the most common predicted key byte across all traces for a specific byte of the AES key.

6. Key Recovery: This process is repeated for all 16 bytes of the AES key, one byte at a time. The recovered key bytes are collected and assembled into the final AES key.

7. Final Outcome: The outcome of the attack is the full AES key, which is recovered by successfully predicting the 16 individual key bytes.

```
C:\Users\sghai\PythonProject1\.venv\Scripts\python.exe
"C:\Users\sghai\PythonProject1\.venv\attack_dataset .py"
Recovering key byte 0...
2025-01-28 14:22:58.675132: I tensorflow/core/platform/cpu_feature_guard.cc:193]
To enable them in other operations, rebuild TensorFlow with the appropriate comp
2025-01-28 14:22:59.321871: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1
2025-01-28 14:23:13.741769: I tensorflow/stream_executor/cuda/cuda_dnn.cc:384] L
2025-01-28 14:23:32.968216: W tensorflow/stream_executor/gpu/redzone_allocator.c
Relying on driver to perform ptx compilation.
Modify $PATH to customize ptxas location.
This message will be only logged once.
2025-01-28 14:23:36.949661: I tensorflow/stream_executor/cuda/cuda_blas.cc:1614]
313/313 [==============================] - 42s 20ms/step
Recovered key byte 0: 7
Recovering key byte 1...
313/313 [==============================] - 7s 21ms/step
Recovered key byte 1: 174
Recovering key byte 2...
313/313 [==============================] - 7s 21ms/step
Recovered key byte 2: 73
Recovering key byte 3...
313/313 [==============================] - 7s 21ms/step
Recovered key byte 3: 127
Recovering key byte 4...
313/313 [==============================] - 7s 21ms/step
Recovered key byte 4: 123
Recovering key byte 5...
313/313 [==============================] - 7s 21ms/step
```

```
Recovered key byte 5: 63
Recovering key byte 6...
313/313 [==============================] - 7s 21ms/step
Recovered key byte 6: 114
Recovering key byte 7...
313/313 [==============================] - 7s 21ms/step
Recovered key byte 7: 39
Recovering key byte 8...
313/313 [==============================] - 7s 21ms/step
Recovered key byte 8: 245
Recovering key byte 9...
313/313 [==============================] - 7s 21ms/step
Recovered key byte 9: 103
Recovering key byte 10...
313/313 [==============================] - 7s 21ms/step
Recovered key byte 10: 244
Recovering key byte 11...
313/313 [==============================] - 7s 21ms/step
Recovered key byte 11: 250
Recovering key byte 12...
313/313 [==============================] - 7s 21ms/step
Recovered key byte 12: 213
Recovering key byte 13...
313/313 [==============================] - 7s 21ms/step
Recovered key byte 13: 177
Recovering key byte 14...
313/313 [==============================] - 7s 21ms/step
Recovered key byte 14: 143
Recovering key byte 15...
313/313 [==============================] - 7s 21ms/step
Recovered key byte 15: 40
Recovered AES key:
[7, 174, 73, 127, 123, 63, 114
39, 245, 103, 244, 250, 213, 177, 143, 40]

Process finished with exit code 0
```

3 Conclusion

In this project, we demonstrated the effectiveness of deep learning techniques, specifically Convolutional Neural Networks (CNNs) inspired by the VGG16 architecture, in recovering AES encryption keys through side-channel analysis. By utilizing the ASCAD dataset, which provides profiling and attack traces from an embedded system, we were able to train models to predict individual key bytes of the AES encryption key.

The process began with the preprocessing of the side-channel traces, followed by the design and training of a CNN model tailored to learn the underlying patterns in the power consumption data during cryptographic operations. The trained models were then used to successfully predict the AES key, byte by byte, leveraging statistical techniques such as mode calculation to improve accuracy.

The results highlight the significant potential of deep learning, especially CNNs, in cryptanalysis, providing a non-invasive and highly efficient method to recover secret keys. The use of GPU acceleration was crucial for handling the large dataset and ensuring fast model training.