Advanced Topics In Cybersecurity 2

Task 2

Federico Dittaro, 12345773

Declaration

I declare that I am the sole author of this submission. I take full responsibility for all content. I confirm that I have either not used AI based tools at all, or, that I have only used them for the purposes of checking grammar and spelling.

1 Profiling method

The objective of this request is to apply either a profiling method or leakage detection on a dataset different from that used in Task 1. For my approach, I chose to implement a profiling method using the fixed-key dataset "ATMEGA_AES_V1". This dataset consists of synchronized traces where all acquisitions share the same fixed key. The ASCAD database contains raw traces with 700 samples of interest, structured as follows:

- A 50,000 traces profiling dataset.
- A 10,000 traces attack dataset.

To apply the profiling method to this dataset I started from the python file *profile.py* available on the Moodle page of the course which computes the signal, the SNR, the Pearson correlation and the estimated mean for each HW for each trace point.
The first step involved running the code with the *ASCAD.h5* file instead of *snrtraces.h5*. To achieve this, I modified the initial part of the code to correctly process the traces within the file. Specifically, the necessary adjustments were made to the section of the code shown below.

```
1    dset = fhandle['traces']
2    inputs = np.array(dset[:, 0], dtype='int')
3    traces = dset[:, 1:6]
```

Listing 1: original code

To find the correct keys I used the command **list(fhandle.keys())** which gives as result the pair ['Attack_traces', 'Profiling_traces']. At this point it is possible to modify the code.

```
    fhandle = h5py.File(filename)
    profiling_traces = np.array(fhandle['Profiling_traces']['traces'
        ][:, :num_points])
    profiling_labels = np.array(fhandle['Profiling_traces']['labels'],
        dtype='int')

    attack_traces = np.array(fhandle['Attack_traces']['traces'][:, :
        num_points])
    attack_labels = np.array(fhandle['Attack_traces']['labels'], dtype=
        'int')
```

Listing 2: modified code

Now to run the script with the profiling traces it is just necessary to correctly replace the new variables in the existing code. The original variable *input* is replaced with *profiling_labels* and the original variable *traces* with *profiling_traces*.
Now to run a template attack it is necessary to:

1. Create the templates from the profiling traces.
2. Use the templates with the attack traces.

1.1 Creating the Gaussian templates

A template is a set of probability distributions that describe what the power traces look like for many different keys, this information can be used to find subtle differences between power traces and to make very good key guesses for a single power trace. The related code is reported in the box below.

```
    print("[+] Creating Gaussian Templates")
    mean_vectors = []
    covariance_matrices = []

    for h in range(9):
        indices = np.nonzero(HW[profiling_labels] == h)
        group_traces = profiling_traces[indices, :].squeeze()

        mean_vector = np.mean(group_traces, axis=0)
        covariance_matrix = np.cov(group_traces.T)

        mean_vectors.append(mean_vector)
        covariance_matrices.append(covariance_matrix)

    print("[+] Gaussian Templates Created")
```

Listing 3: creating the Gaussian templates

The code initializes lists to store mean vectors and covariance matrices. It then iterates over each Hamming weight (from 0 to 8), extracts the power traces corresponding to the current Hamming weight, and computes the mean vector and covariance matrix for that group of traces. It is important to notice that the covariance is computed on the traces transpose matrix because of how `numpy.cov()` interprets input data. In the code, the group traces have a shape of (num_traces, num_points), where rows represent traces (observations) and columns represent time points (features). However, `numpy.cov()` assumes that rows correspond to features and columns to observations. Therefore, it is necessary to transpose the matrix to match this expected structure, resulting in a shape of (num_points, num_traces), where each row now represents a time point across multiple traces. Now based on this 9 templates it is possible to run the attack.

1.2 Using the templates

The attack utilizes 1,000 traces from the attack dataset and initializes an array to store the recovered AES key guesses. It iterates through all 16 bytes of the AES key, performing the attack on each one. For each byte, the 1,000 traces are used to compute the likelihood of all 256 possible key guesses. Each guess is XORed with the attack label to compute the S-box output, and the Hamming weight of the result is determined using the HW lookup table. Based on the Hamming weight, the corresponding mean vector and covariance matrix are selected. Using these elements, the multivariate normal probability density function (PDF) is computed for the attack trace to assess the likelihood of each key guess. The guess with the highest probability for the current trace is then selected and stored in the predicted keys array. At the end of execution, the code prints the 16 predicted key bytes contained in the predicted keys array. An important statement of this code is in the likelihood computation where it is necessary to set the variable "allow_singular" to true since the covariance matrix is singular. To check its singularity it is sufficient to print the determinants of the matrix which are

- Determinant of covariance matrix for HW 0: -0.0
- Determinant of covariance matrix for HW 1: 0.0
- Determinant of covariance matrix for HW 2: 0.0
- Determinant of covariance matrix for HW 3: 0.0
- Determinant of covariance matrix for HW 4: 0.0
- Determinant of covariance matrix for HW 5: 0.0
- Determinant of covariance matrix for HW 6: 0.0
- Determinant of covariance matrix for HW 7: 0.0
- Determinant of covariance matrix for HW 8: -0.0

To run the code, it is necessary to import the AES S-box and the multivariate normal library (`from scipy.stats import norm, multivariate_normal`). The full attack code is provided in the section below. I tried running the code to get the result, but after 8 hours, it was still testing the key for byte 0, so I stopped the execution. My PC couldn't run this script for such a long time, but I believe the underlying idea should be correct.

```
    num_traces = 1000
    attack_traces = attack_traces[:num_traces]
    attack_labels = attack_labels[:num_traces]

    print("[+] Testing Templates with Attack Traces")
    recovered_key = []

    for key_byte_index in range(16):
        print(f"Testing for key byte index: {key_byte_index}")
        predicted_keys = []

        for trace in attack_traces:
            key_likelihoods = []

            for key_guess in range(256):
                sbox_output = key_guess ^ attack_labels[key_byte_index]
                hamming_weight = HW[sbox_output]

                mean_vector = mean_vectors[hamming_weight]
                covariance_matrix = covariance_matrices[hamming_weight]

                likelihood = multivariate_normal.pdf(trace, mean=
                    mean_vector, cov=covariance_matrix, allow_singular=
                    True)
                key_likelihoods.append(likelihood)

            best_key = np.argmax(key_likelihoods)
            predicted_keys.append(best_key)

        predicted_key_byte = np.argmax(np.bincount(predicted_keys))
        recovered_key.append(predicted_key_byte)

    recovered_key = np.array(recovered_key)
    print("[+] Recovered AES Key:")
    print(recovered_key)
```

Listing 4: using the Gaussian templates