Physical Attacks on Secure Systems (NWI-IMC068)

Tutorial 3: Template Attacks

Goals: After completing this tutorial successfully, you should understand how Template Attacks work and be able to perform such attacks.

Before you start: To complete this tutorial, please download the notebook for this tutorial! You can collect the necessary traces yourself (your teacher will provide a package containing the Chipwhisper-lite board) or download the traces linked in this tutorial.

1 Template Attacks of an AES-128 software implementation

In this section, we first collect the traces for building the templates and the matching phase of the attack. We go through the template creation tasks and will create our templates based on the collected traces. Then we will use the created templates for the attacks phase!

1.1 Overview Template attacks

A template captures characteristics of a probability distribution corresponding to a specific key. Effectively, a template says *if you are going to use key k, your power trace will look like,* $p_k(\mathbf{X})$, where $\mathbf{X}$ is a random variable that represents power consumption. We can use this information to find subtle differences between power traces and to make good key guesses for a single power trace.

One of the downsides of template attacks is that they require many traces to be preprocessed before the attack can begin. This is mainly for statistical reasons. To come up with a realistic distribution to model the power traces for every key, we need a large number of traces for every possible key. For example, if we attack a single byte of subkey of AES-128, we need to create 256 power consumption models (one for every key hypothesis from 0 to 255). Note that we might need tens of thousands of traces to get enough data to make good models. This will be exhaustive!

An alternative is to model a sensitive key part, for example, the S-box output in AES. We could reduce the number of required traces if we make a model for possible Hamming Weight (HW), then we would end up with 9 models (why?), which is an order of magnitude the smaller number of models. However, we cannot recover the key from a single attack trace! We need more information to recover the secret key (why?).

The AES S-box is defined in the provided notebook. As before, you calculate the Hamming Weight (HW) of the S-box output. You can use the code you wrote in the previous tutorial.

1.2 PoI selection algorithm

We aim to create multivariate probability density functions that describe the power traces for every possible key with the same HW. If we use all 5 000 samples in the trace, we will create a 5 000-dimension distribution. This is not practical! Thankfully, not every point on the power trace is important. There are two main reasons for this:

1. Our choice of key does not affect the entire power trace. It is likely that the subkeys only influence the power consumption at a few critical samples, so we can ignore most of the samples not related to this.

2. We might be taking more than one sample per clock cycle. The Chipwhisperer ADC often runs four times faster than the target device. There's no real reason to use all these samples - we can get as much information from a single sample at the right time.

Therefore, we can usually live with a handful (3-5) Points of Interest (PoI). If we can choose such points and write down a model using these samples, we can use a 3D or 5D distribution! A significant improvement over the original 5 000D model.

How can we select these PoIs? There are several ways to pick the most important points in each trace. The goal is to find points that vary strongly between different operations (subkeys or HW). The simplest method is the sum of differences method. We will use this method here! The algorithm for the sum of difference method is:

1. For every operation k and every sample i, find the average power $M_{k,i}$. For instance, if there are T_k traces where we performed operation k, then this average is $M_{k,i} = \frac{1}{T_k} \sum_{j=1}^{T_k} t_{j,i}$, where $t_{j,i}$ represents the sample i in the j^{th} trace t.

2. After finding all the means, calculate their absolute pairwise differences. Add these up. This will give one "trace", which has peaks where the samples are usually different. The calculation looks like: $D_i = \sum_{\text{for all } (k_p,k_q)} \mid M_{k_p,i} - M_{k_q,i} \mid$, where k_p and k_q represent two different operations.

3. The peaks of D_i show the most important points. We need to pick some peaks that are not too close. One algorithm to do this is:

   - Pick the highest point in D_i and save this value of i as a point of interest. (i.e. $i = \arg\max(D_i)$)
   - Throw out the nearest N points (where N is the minimum spacing between POIs).
   - Repeat until enough POIs have been selected.

Suppose we have choose I PoIs, which are at samples $s_i (0 \leq i < I)$. Then, our goal is to find a mean and covariance matrix for every operation (every choice of subkey or intermediate HW). Let's say that there are K of these operations (maybe 256 subkeys or 9 possible HW).

We will now look at a single operation $k (0 \leq k < K)$. The steps are:

1. Find every power trace t that falls under the category of "operation k". (for example: find every power trace where we used a subkey of `0x01`.) We will say that there are T_k of these, so t_{j,s_i} means the value at trace j and POI s_i.
2. Find the average power μ_i at every PoI. This calculation will look like: $\mu_i = \frac{1}{T_k} \sum_{j=1}^{T_k} t_{j,s_i}$.
3. Find the variance v_i of the power at each PoI. One way of calculating this is: $v_i = \frac{1}{T_k} \sum_{j=1}^{T_k} (t_{j,s_i} - \mu_{s_i})^2$.
4. Find the covariance c_{i,i^*} between the power at every pair of POIs (s_i and s_{i^*}). One way of calculating this is: $c_{i,i^*} = \frac{1}{T_k} \sum_{j=1}^{T_k} (t_{j,s_i} - \mu_{s_i})(t_{j,s_{i^*}} - \mu_{s_{i^*}})$.
5. Now, we need to put together the mean and covariance matrices as:

$$\mu = \begin{bmatrix} \mu_1 \\ \mu_2 \\ \mu_3 \\ \vdots \end{bmatrix}$$

$$\sum = \begin{bmatrix} v_1 & c_{1,2} & c_{1,3} & \dots \\ c_{2,1} & v_2 & c_{2,3} & \dots \\ c_{3,1} & c_{3,2} & v_3 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

These steps must be done for every operation k. At the end of this preprocessing, we will have K mean vectors and covariance matrices, modeling each K different operation that the target can do. To verify your understanding, consider the following questions:

- Are the mean vectors and covariance matrices enough to describe the template or the model? Why?
- Is there a Python command to calculate the covariance matrix efficiently?

1.3 How to Use the Created Templates

For executing the attack, we need a smaller number of traces. Let's say that we have A traces. The sample values will be labeled a_{j,s_i} $(1 \leq j < A)$.

1.3.1 Template Matching

First, let's apply the template matching to a single trace. We want to compute the probability that each key produces the trace. We need to do the following:

1. Put our trace values at the POIs into a vector. This vector will be

$$\mathbf{a_j} = \begin{bmatrix} a_{j,1} \\ a_{j,2} \\ a_{j,3} \\ \vdots \end{bmatrix}$$

2. Calculate the PDF for every key guess and save these for later. This might look like: $p_{k,j} = p_k(\mathbf{a_j})$.

3. Repeat these two steps for all of the attack traces.

This process gives us an array of $p_{k,j}$, which says: "Looking at trace j, how likely is it that key k is the correct one?" Now we need to combine these results! The last step is to combine our $p_{k,j}$ values to decide which key best fits. The easiest way to do this is to combine them as,

$$P_k = \prod_{j=1}^{A} p_{k,j}$$

For example, if we guessed that a subkey was equal to 0x00 and our PDF results in 3 traces were (0.9, 0.8, 0.95), our overall result would be 0.684. Having one trace that does not match the template can cause this number to drop quickly, helping us eliminate the wrong guesses. Finally, we can pick the highest value of P_k, which tells us which guess fits the templates the best, and we are done!

This method of combining our per-trace results can suffer from precision issues. After multiplying many large or too small numbers, we could end up with too large or small to fit into a floating point variable. An easy fix is to work with logarithms. Instead of using P_k directly, we can calculate

$$\log P_k = \sum_{j=1}^{A} \log p_{k,j}$$

Comparing these logarithms will give us the same results without the precision issues.

1.4 [Optional] Collect Traces with Chipwhisperer

In the first tutorial, you successfully installed Chipwhisperer and collected some traces. Here we will redo the trace collection task, and then we will attack the software implementation of AES using the traces.

The first capturing task is for the templates. This phase requires full control of the device, key, and plaintext. We will use a random key, meaning the key will change on every encryption. To do this, add the following code to your trace collection block in Jupyter notebook,

```
ktp.fixed_key = False
```

The reason is that if we used a fixed key during the profile, there is a strong chance that we will "overfit" the templates. The templates will only be valid for the fixed key we used during the profiling phase. This will not be possible in practice since we do not know the key in use! Use TinyAES and collect 6000 traces with Chipwhisperer to build the templates upon. Make sure that each trace has 5000 samples. Save the collected traces, the key used, and the plaintexts in a file. You can use the following code.

```
# Capture Traces
from tqdm import tnrange
import chipwhisperer as cw

# Make a project to save template traces
project = cw.create_project("projects/tutorial_template_templatedata.cwp",
            overwrite = True)

ktp = cw.ktp.Basic()
ktp.fixed_key = False # RANDOM KEY in addition to RANDOM TEXT

N = 6000  # Number of traces
for i in tnrange(N, desc='Capturing traces'):
    key, text = ktp.next()
    trace = cw.capture_trace(scope, target, text, key)
    if trace is None:
        continue
    project.traces.append(trace)

# Save the project
project.save()
```

Now let's collect the attack traces! You will use the same Chipwhisperer! Remember, in practice, you have two similar devices one is used to build templates, and the other is the one that runs the cipher implementation you want to attack! The code is the same as above! The only difference is that you collect 20 traces, and you do not change the key! You will save it under a new project!

```
project = cw.create_project("projects/tutorial_template_validate.cwp",
            overwrite = True)

ktp = cw.ktp.Basic()

N = 20
for i in tnrange(N, desc='Capturing traces'):
    key, text = ktp.next()
    trace = cw.capture_trace(scope, target, text, key)
    if trace is None:
        continue
    project.traces.append(trace)

project.save()
```

Disconnect from the Chipwhisperer. If you cannot collect traces with Chipwhisperer for some reason, or you forgot to save the collected traces, and you want to finish the tutorial at home, you can download some sample traces from here.

Before starting to work on the attack, you should load the traces. You can do this with the following commands. Just make sure you set the path to the project correctly!

```
import chipwhisperer as cw
project_template = cw.open_project("&&&/tutorial_template_templatedata.cwp")
```

With the for loop provided below you can make sure that the key is changed in each trace collection for building the templates.

```
#Let's confirm that we get random keys
for i in range(0, 4):
    print("%d: "%i + " ".join(["%02x"%project_template.keys[i][j]
                              for j in range(0, 16)]))
```

You can read the plaintexts and traces with the same approach as above. Look at the provided notebook for this tutorial.

The sensitive variable we will use to attack the software implementation of AES is the S-Box output (as usual); see Fig 1. The software implementation of AES that we aim to attack uses 128-bit key.

1.5 Template Building

We first attack the first byte of the secret key. Let's build one template for each possible HW of the value at the output of the S-box. We first select our PoIs using the method we explained in the Sect. 1.1. Follow the instruction below for the first byte (i.e., `target_byte = 0`),

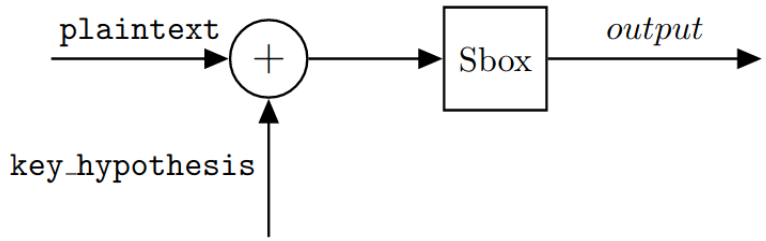

Figure 1: TinyAES Sbox

1. for all plaintexts and keys, calculate the value of the S-box. Then, calculate the HW of the output of the S-box.
2. Make a list that holds 9 empty lists. This list will keep categorizing the traces corresponding to the calculated HW values.
3. You can convert the inner lists to numpy arrays. This will make the manipulation of the dara easier!

Now let's find the PoIs. Continue with the following instructions,

4. We need to calculate the average/mean of the traces in each list/array. You can define a 2D array (`Possible_HW_Values`×`Number_of_Samples`) and keep the mean of traces in each class in it.
5. Write a nested `for` loop to calculate the absolute pairwise differences of the means and add them up into the defined variable `tempSumDiff`. Your nested `for` loop can look like the following,

```
for i in range(9):
    for j in range(i):
```

6. You can draw `tempSumDiff` variable with

```
hv.Curve(tempSumDiff).opts(height=600, width=600)
# Or you can use plt.plot(tempSumDiff)
```

Now we need to select the PoIs. We can use the algorithm from the theory page to pick out some POIs:

7. Make an empty list of POIs. We select 5 PoIs with a minimum spacing of 5 samples.
8. Find the biggest peak in the sum of differences trace and add it to the list of POIs
9. Zero out some of the surrounding points

10. Repeat until we have enough POIs

Now that we have found PoIs, we can build our templates. Remember, we will generate one template for each possible HW of the output of the S-box. Each template would be a multivariate normal distribution. To define a normal distribution, one only needs to define its mean vector and the covariance matrix. We have 9 different operations (possible HW values). This means 9 different templates should be derived. With 5 (or `numPOIs`) POIs picked out, we can build our multivariate distributions at each point for each HW. We need to write down two matrices for each Hamming Weight (or operation):

- A mean matrix ($1 \times$ `numPOIs`) which stores the mean at each POI
- A covariance matrix (`numPOIs` $\times$ `numPOIs`) which stores the variances and covariances between each of the POIs

The mean matrix is easy to set up because we have already found the mean at every sample. All we need to do is grab the right points. The covariance matrix is a bit more complex. We need a way to find the covariance between two 1D arrays. Helpfully, NumPy has the cov(a, b) function, which returns the matrix

$$np.cov(a, b) = \begin{bmatrix} cov(a, a), & cov(a, b) \\ cov(b, a), & cov(b, b) \end{bmatrix}$$

We can use this to define our own covariance function,

```
def cov(x, y):
    # Find the covariance between two 1D lists (x and y).
    # Note that var(x) = cov(x, x)
    return np.cov(x, y)[0][1]
```

As mentioned in the comments, this function can also calculate the variance of an array by passing the same array for both x and y. Use this function and build the covariance matrix. Congratulation! Your templates are complete! Now we should apply the derived templates!

1.5.1 Template matching

The very last step is to apply our template to these traces. We want to keep a running total of $\log P_k = \sum_{j=1}^{A} \log p_{k,j}$, so we will make space for our 256 guesses, `P_k = np.zeros(256)` Then, we want to do the following for every attack trace,

11. Grab the samples from the points of interest and store them in a list **a**
12. For all 256 of the subkey guesses
    - Figure out which HW we need, according to our known plaintext and guessed subkey

- Build a `multivariate_normal` object using the relevant mean and covariance matrices
- Calculate the log of the PDF ($\log p((\mathbf{a}))$) and add it to the running total

13. List the best guesses we've seen so far.

Make sure you add the following to the list of your imports,

```
from scipy.stats import multivariate_normal
```

With any luck, you will have found the correct key byte! You can try extending this to attack multiple bytes instead. The traces you already recorded will work just great; you'll need to make the following changes,

- Build a new POI for each byte number
- Build new matrices for each byte number
- Apply those matrices for each byte

Now that you have successfully finished the Template Attack on the software implementation of AES, you are ready to answer the following questions,

2. Get rid of the HW assumption. You can build templates based on each key byte value itself directly, which means a single power trace would result in the secret key! Conduct this attack! Do you succeed? Compare this to the attack you conducted using HW assumption. Do we need more data? Why? Can we use the same plaintext? If yes, does this make our task easier?

1. What other methods do you think can be used to select the PoIs? Select one and try it! What do you see? Can you succeed? What will happen if you increase/decrease the number of PoIs?

2. Try PCA to select the PoIs or decrease dimensionality. How can this be done? Can you succeed? What if you select more/less several principal components?

3. Calculate the HW of the output of the S-box. Do traces equally distributed in each class with different HW? If not, which HW class contains more traces? Does this affect the accuracy of template creation and our Template Attack? If yes, how?

Appendices

A An Introduction to Template Attacks

Template Attacks (TA) are a subset of profiling attacks. In these attacks, the attacker creates a "profile" of a sensitive device and applies this profile to find a victim's secret

information. Template attacks are powerful types of side-channel attacks. However, they require more setup than CPA attacks. To perform a template attack, the attacker must have access to another copy of the device. The attacker should have full control over the copy the built profiles are based on. Then, they must perform a substantial amount of pre-processing to create the templates. In practice, The attacker may need to collect dozens of thousands of power traces. These traces are used to build the templates or the profiles, and the attacker uses the fully controlled device to collect them. However, to conduct successful template attacks, the attacker requires a very small number of traces from the victim device. With enough pre-processing, the sensitive information may be recovered from just a trace!

To conduct a Template Attack to recover a key used in encryption done by a cryptographic algorithm, the following steps are needed,

1. The attacker records a large number of power traces using **many different inputs** (plaintexts and keys) from the fully controlled copy of the device. The number of collected traces should be enough to give us information about each subkey value.

2. The attacker creates templates based on the collected traces. *A template is a multivariate distribution of the power traces at a sample or point.*

3. The attacker records a small number of power traces from the victim device. The attacker uses **multiple plaintexts**, but without access to the **secret key**, which by assumption is **fixed** during trace collection.

4. In the last step, the attacker applies the template to the collected attack traces. The attacker finds the *most likely* value to be the correct subkey. And this procedure is done until the whole key or secret information is recovered!

Now let's move to the next section. It briefly introduces the probability and estimation concepts used in the Template Attacks!

B Estimation and Detection Recap

Imagine having a random variable X equal to $\theta + \omega$. ω is a normally distributed random variable with zero means, i.e., $\mu = 0$, and its variance is σ^2. We say $\omega \sim \mathcal{N}(0, \sigma^2)$. For example, X can be read from a multimeter of an electrical signal. The electrical signals are inherently noisy! As a result, any time we take a voltage measurement, we do not expect to see a perfect, constant level. For example, if we attach a multimeter to a $5V$ source and take 4 measurements, we might expect to see a data set like (4.95, 5.01, 5.06, 4.98). One way of modeling this voltage source is $X = \theta + \omega$, where the noise is represented by ω, which has a *normal distribution*. If we know θ, then X has a normal distribution,

$$p(X \mid \theta) \sim \mathcal{N}(\theta, \sigma^2). \tag{1}$$

In statistics, maximum likelihood estimation (MLE) is a method of estimating the parameters of an assumed probability distribution, given some observed data. So in this

context, we want to estimate parameter X, while we have observed θ. For the normal distribution $\mathcal{N}(\theta, \sigma^2)$ which has the following probability density function,

$$p(X \mid \theta) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(X-\theta)^2}{2\sigma^2}\right),$$

the MLE would be as follow,

$$\begin{aligned} \hat{X} = \arg\max_X \Big(\log p(X \mid \theta)\Big) &= \arg\max_X \log \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(X-\theta)^2}{2\sigma^2}\right) \\ &= \arg\max_X \frac{-1}{2}\frac{(X-\theta)^2}{\sigma^2}. \end{aligned} \tag{2}$$

The only remaining is to solve the maximization problem!

If X is multivariate random variable with dimension k, i.e. $\mathbf{X} = (X_1, \ldots, X_k)^{\mathrm{T}}$, then

$$p(\mathbf{X} \mid \boldsymbol{\theta}) = (2\pi)^{-k/2}\det(\boldsymbol{\Sigma})^{-1/2} \exp\left\{-\frac{1}{2}(\mathbf{X}-\boldsymbol{\theta})^T\boldsymbol{\Sigma}^{-1}(\mathbf{X}-\boldsymbol{\theta})\right\},$$

where $\boldsymbol{\theta}$ is the mean vector

$$\boldsymbol{\theta} = \mathrm{E}[\mathbf{X}] = (\mathrm{E}[X_1], \mathrm{E}[X_2], \ldots, \mathrm{E}[X_k])^{\mathbf{T}} = (\theta_1, \theta_2, \ldots, \theta_k)^{\mathbf{T}},$$

and $\boldsymbol{\Sigma}$ is the $k \times k$ covariance matirx,

$$\Sigma_{i,j} = \mathrm{E}[(X_i - \theta_i)(X_j - \theta_j)] = \mathrm{Cov}[X_i, X_j]$$

$$\begin{aligned} \hat{\mathbf{X}} = \arg\max_{\mathbf{X}} \Big(\log p(\mathbf{X} \mid \boldsymbol{\theta})\Big) &= \arg\max_{\mathbf{X}} \left(\log(2\pi)^{-k/2}\det(\boldsymbol{\Sigma})^{-1/2} \exp\left\{-\frac{1}{2}(\mathbf{X}-\boldsymbol{\theta})^T\boldsymbol{\Sigma}^{-1}(\mathbf{X}-\boldsymbol{\theta})\right\}\right) \\ &= \arg\max_{\mathbf{X}} \left(-\frac{1}{2}(\mathbf{X}-\boldsymbol{\theta})^T\boldsymbol{\Sigma}^{-1}(\mathbf{X}-\boldsymbol{\theta})\right). \end{aligned} \tag{3}$$

Again we need to solve the above "easy" maximization to estimate the $\mathbf{X}$! How do you solve this maximization problem?