SCA_template_attack

From https://wiki.newae.com/Template_Attacks:

Theory from internet

To perform a template attack, the attacker must have access to another copy of the protected device that they can fully control. Then, they must perform a great deal of pre-processing to create the template - in practice, this may take dozens of thousands of power traces. However, the advantages are that template attacks require a very small number of traces from the victim to complete the attack. With enough pre-processing, the key may be able to be recovered from just a single trace.
There are four steps to a template attack:

1. Using a copy of the protected device, record a large number of power traces using many different inputs (plaintexts and keys). Ensure that enough traces are recorded to give us information about each subkey value.
2. Create a template of the device's operation. This template notes a few "points of interest" in the power traces and a multivariate distribution of the power traces at each point.
3. On the victim device, record a small number of power traces. Use multiple plaintexts. (We have no control over the secret key, which is fixed.)
4. Apply the template to the attack traces. For each subkey, track which value is most likely to be the correct subkey. Continue until the key has been recovered.

Sidetrack: Signals, Noise, and Statistics

A template is effectively a multivariate distribution that describes several key samples in the power traces. This section will describe what a multivariate distribution is and how it can be used in this context.

Noise Distributions

Electrical signals are inherently noisy. Any time we take a voltage measurement, we do not expect to see a perfect, constant level. For example, measuring V_{DD} 4 times might give things like $4.95, 5.03, \ldots$
Therefore, the following model could be used to model this measurement:

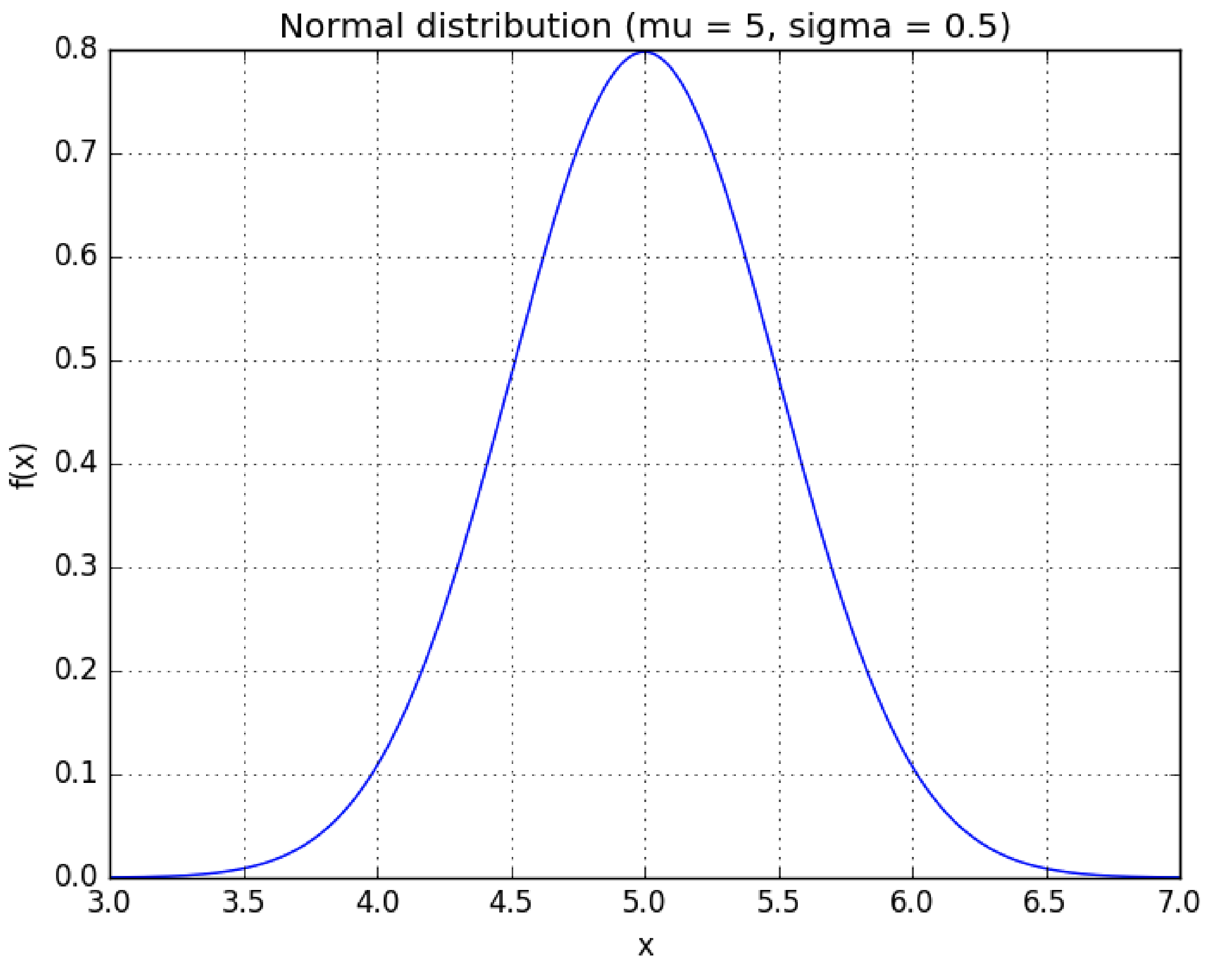
$$\mathbf{X} = X_{actual} + \mathbf{N}$$

- X_{actual} is the noise-free level and $\mathbf{N}$ is the additional noise.
  - $X_{actual} = 5$
  - $\mathbf{N}$ is a random variable
    - Every time you take a measurement, you expect to see a different value
    - thus $\mathbf{X}$ also is a random variable

A simple model for these random variables uses a Gaussian distribution (bell curve). The probability density function (PDF) of a Gaussian distribution is:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma^2)}$$

- μ is the mean
- σ standard deviation



- PDF is used to calculate how likely a certain measurement it:
  - $f(5.1) \approx .7821$
  - $f(7.0) \approx .0003$

We'll use this to our advantage in this attack: if $f(x)$ is very small for one of our subkey guesses, it's probably a wrong guess.

Multivariate Statistics

The 1-variable Gaussian distribution works well for one measurement. What if we're working with more than one random variable?

- Imagine at two points in a circuit you are measuring voltages, $\mathbf{X}, \mathbf{Y}$
  - You could model them individually with Gaussian distribution
  - But you are assuming then they are independent
  - Multivariate distributions let us model multiple random variables that may or may not be correlated.
  - In a multivariate distribution, instead of writing down a single variance σ, you keep track of a whole matrix of covariances.

  Example: Three random variables

$$\mathbf{\Sigma} = \begin{bmatrix} Var(\mathbf{X}) & Cov(\mathbf{X},\mathbf{Y}) & Cov(\mathbf{X},\mathbf{Z}) \\ Cov(\mathbf{Y},\mathbf{X}) & Var(\mathbf{Y}) & Cov(\mathbf{Y},\mathbf{Z}) \\ Cov(\mathbf{Z},\mathbf{X}) & Cov(\mathbf{Z},\mathbf{Y}) & Var(\mathbf{Z}) \end{bmatrix}$$

- This distribution needs to have a mean for each random variable:

$$\mu = [\mu_X \ \mu_Y \ \mu_z]$$

- PDF of this distribution uses a vector with all of the variables is input

Creating the Template

```
A template is a set of probability distributions that describe what the
power traces look like for many different keys. Effectively, a template
says:
```

- If you are going to use key k, your power trace will look like the distribution $f_k(\mathbf{x})$
  This information can be used to find subtle differences between power traces and to make very good key guesses for a single power trace.

Number of traces

One of the downsides of template attacks is that they require a great number of traces to be pre-processed before the attack can begin. This is mainly for statistical reasons. In order to come up with a good distribution to model the power traces for *every key*, we need a large number of traces for *every key*.
For example: To attack a single subkey of AES-128, you would need to create 256 power consumption models (one for every number from 0 to 255)

Note that we don't have to model every single key. One good alternative is to model a sensitive part of the algorithm, like the substitution box in AES. We can get away with a much smaller number of traces here; if we make a model for every possible Hamming weight, then we would end up with 9 models, which is an order of magnitude smaller. However, then we can't recover the key from a single attack trace - we need more information to recover the secret key.

Points of Interest

Our goal is to create a multivariate probability describing the power traces for every possible key.

If we modeled the entire power trace this way (with, say, 3000 samples), then we would need a 3000-dimension distribution. This is insane, so we'll find an alternative.

Thankfully, not every point on the power trace is important to us.

- In the lab text we are going to take the sample which has the largest pearson corr coefficient (A way of picking POI).
  - Or take the two largest, or x largest then you would end up with xD distribution, a great improvement over the original nb_samplesD model.
- Our choice of key doesn't affect the entire power trace. It's likely that the subkeys only influence the power consumption at a few critical times. If we can pick these important times, then we can ignore most of the samples.

Analysing the Data

Suppose we've picked I points of interest, which are at samples $s_i : 0 \leq i \leq I$.
Then our goal is to find a mean and covariance matrix for every operation (every choice of subkey or intermediate Hamming weight). Let's say that there are K of these operations (maybe 256 subkeys or 9 possible Hamming weights)

Let's look at a single operation k:

- Find every power trace t that falls under the category of "operation k" (find every power trace where we used a subkey of 0x01). Say there are T_k of these, so t_{j,s_i} means the value at trace j and POI i.
- Find the average power μ_i at every point of interest:

$$\mu_i = \frac{1}{T_k}\sum_{j=1}^{T_k} t_{j,s_i}$$

- Find the variance v_i of the power at each point of interest

$$v_i = \frac{1}{T_k}\sum_{j=1}^{T_k}(t_{j,s_i} - \mu_i)^2$$

- Find the covariance $c_{i,i*}$ between the power at every pair of POIs (i and i)
  `$$c_{i,i} = \frac{1}{Tk}\sum{j=1}^{Tk}(t{j,si-\mu_i})(t{j,s{i*}}-\mu{i*})$$`
- put together the mean and covariance matrices as

$$\mu = [\mu_1, \mu_2, \ldots]$$
$$\Sigma = [v_1, c_{12}\ldots]$$

  These steps must be done for every operation k. At the end of this preprocessing, we'll have K mean and covariance matrices, modelling each of the K different operations that the target can do.

Using the Template

With A traces and sample values a_{j,s_i} $(1 \leq j \leq A)$

Applying the template

For a single trace. Our job is to decide how likely all of our key guesses are

- Put our trace values at the POIs into a vector

$$\mathbf{a_j} = [a_{j,1}, a_{j,2}, \ldots]$$

- Calculate the PDF for every key guess and save these for later

$$p_{k,j} = f_k(\mathbf{a_j})$$

- Repeat the above 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 it is that key k is the correct one ?

Combining results

The last step is to combine $p_{k,j}$ values to decide which key is the best fit.

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

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), then our overall result would be 0.684.

Having one trace that doesn't match the template can cause this number to drop quickly, helping us eliminate the wrong guesses.

Finally pick the highest value of P_k which tells us which guess fits the templates the best and we are done.

Note:
This method of combining our per-trace results can suffer from precision issues. After multiplying many large or small numbers together, we could end up with numbers that are 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 use logarithm:

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

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

Lab question 3

When an adversary can control a device similar to the one he intends to attack, he can take advantage of it to estimate more precise leakage models and exploit more powerful statistical tools.

Template attacks are a typical example of such stronger attacks. They require the attacker to first run a profiling phase to characterize the distribution of the leakages for each key

and plaintext.
Then, during the online attack, he can apply Bayes' law to compute the likelihood of the different key candidates.

Profiling phase

First, a reasonable (yet heuristic) choice is to use the time sample that gives the maximum Pearson correlation coefficient obtained in the previous exercise.
Second step comprises of building 256 templates corresponding to the 256 possible outputs of the first AES S-box.

- Assuming they follow a normal distribution, each template has the form:

$$\mathcal{N}(l|\mu_i,\sigma_i) = \frac{1}{\sigma_i\sqrt{2\pi}}e^{-\frac{(1-\mu_i)^2}{2\sigma_i^2}}$$

- μ_i is the average leakage value corresponding to the S-box output value i
- σ_i standard deviation related to μ_i

Implement training_phase function

This is the profiling phase function.

```python
def training_phase(index, time_idx, pts, ks, trs):
    """
    index: index of the byte on which to perform the attack.
    time_idx: time sample to consider when building the model
    pts: the plaintexts used in the training set
    ks: the keys used in the training set
    trs: the traces of the training set
    return: the list [us,ss] where
        us: is a numpy array containing the mean of each class
        ss: is a numpy array containing the standard deviation of each
class
    """
    means = np.arange(256)
    stds = np.arange(256)
    nb_traces = trs.shape[0]
    for s_box_out in range(256):
        traces = []
        for key_idx in range(nb_traces):
            key = ks[key_idx,:][index]
            pt = pts[key_idx,:][index]
            if s_box_out == sbox[key ^ pt]:
                traces.append(trs[key_idx,:][time_idx])

        traces = np.asarray(traces)
        mu_i = np.mean(traces,axis=0)
        sigma_i = np.std(traces,axis=0)

        means[s_box_out] = mu_i
        stds[s_box_out] = sigma_i
    return [means, stds]
```

In this function we are returning univariate statistics. For each s-box output, we are going through all the keys and plaintexts. Then we are calculating the actual s-box output. Then we combine the traces which give the s-box output 0, 1, 2, ..., 255 and for each s-box output we calculate the mean and the std at the given time_index (this will be the point of interest in the following section.)

Next, the (online) attack phase works as follows. The templates can be used to estimate the probability of each subkey byte. For each power trace measured (with a plaintext byte x), the probability of each subkey byte can be computed as (comes from Bayes rule):

$$Pr[S(x \oplus k) = i] = \frac{\mathcal{N}(l|\mu_i, \sigma_i)}{\sum_j \mathcal{N}(l|\mu_j, \sigma_j)}$$

The probabilities corresponding to different traces can then be combined in order for the correct subkey byte to be detected (which happens when it has maximum likelihood).

For this we need to implement two functions. `online_phase` and `ta_byte`.

```
def online_phase(index, time_idx, models, atck_pts, atck_trs):
    """
    index: index of the byte on which to perform the attack.
    time_idx: time sample to consider when building the model
    models: the list [us, ss] corresponding to the models of each class
    atck_pts: plaintexts used in the attack set
    atck_trs: traces of the attack set
    return: a numpy array containing the probability of each byte value.
    """
    return np.zeros(256)
def ta_byte(index, train_pts, train_ks, train_trs, atck_pts, atck_trs):
    """
    index: index of the byte on which to perform the attack.
    train_pts: the plaintexts used in the training set
    train_ks: the keys used in the training set
    train_trs: the traces of the training set
    atck_pts: plaintexts used in the attack set
    atck_trs: traces of the attack set
    return: a np.array with the key bytes, from highest probable to less
probable
    """
    return np.arange(256)
```

online_phase

```
    nb_traces = atck_trs.shape[0]
    nb_samples = atck_trs.shape[1]
    probabilities = np.zeros(256)
    for i in range(nb_traces):
        l = atck_trs[i,:][time_idx]
        pt = atck_pts[i,:][index]
        for key_guess in range(256):
            s_box_out = sbox[pt ^ key_guess]
            mu_i = models[0][s_box_out]
            sigma_i = models[1][s_box_out]
            d = norm(mu_i, sigma_i)
            probabilities[key_guess] += np.log(d.pdf(l))

    return probabilities
```

In this function the goal is to accumulate probabilities for each key guess going through the traces we would like to attack.
What does that mean now?
At this point, we have models in our hand that we got from training phase. These models basically tell us if you have s-box output of i, the mean and the std is like this and you could calculate with a given key guess your attacking s-box output and see what is the probability that it is in this distribution you have created from the training set.

That is what we do here.

- For each trace we want to attack
    - go through each key guess

    - calculate the s-box output for the given key guess
    - get the mean and std of this s-box output from the models we have crearted.
    - calculate the probability density function of the Gaussian it represents
    - accumulate them at the key_guess index of probabilities.
        - If it fits the distribution we will have high values of probabilities and they will add up for each trace.

ta_byte

We will use the training data where we know the keys, plaintexts, and traces from the device we have. We will use this data to find the highest correlating point in the traces.

Note: We are doing univariate statistics in this attack. Unlike what is normally done with multivariate.

```
nb_traces =  train_trs.shape[0]
nb_samples = train_trs.shape[1]
weights = []
for i in range(nb_traces):
    x_i = train_pts[i,:][index]
    k_i = train_ks[i,:][index]
    v_i = sbox[x_i ^ k_i]
    hw = hamw(v_i)
    weights.append(np.full(nb_samples,hw))
p_corr = pearson_corr(weights,train_trs)
poi = np.argmax(abs(p_corr))
```

This `poi` value gives us the point at which the Pearson correlation coefficient is largest with the training data.

Then using the `poi` we get the models.

```
models = training_phase(index, poi, train_pts, train_ks, train_trs)
```

What is models?

- models[0]: np array of shape (256,)
    - contains the mean of the time index poi at each s-box output

- models[1]: np array of shape (256,)
  - contains the std of the time index poi at each s-box output
- This is the result of our training.
- We basically said that okay,
  - we have these keys and plaintexts
  - then we have the corresponding trace for them
  - use these to create a model for each s-box output

Then we need to use the online attack to finish the attack

```
    probabilities = online_phase(index, poi, models, atck_pts, atck_trs)

    return np.array(sorted(range(len(probabilities)), key=lambda i:
probabilities[i], reverse=True))
```

Multivariate setting

With numPOIs number of POIs we need to build multivariate distributions at each s-box output (or each hamming weight).

We need to write down 2 matrices for each s-box output:

- A mean matrix
  - shape: `(1xnumPOIs)`
  - stores the mean at each POI
- A covariance matrix
  - shape: `numPOIsxnumPOIs`
  - stores the variances and covariances between each of the POIs

Bivariate

Taking the largest two coefficients.