Advanced Topics in Cybersecurity 2 Elisabeth Oswald

Structural Attacks

2024

1 Introduction

You must read the introduction of every lab sheet. It explains if you must install something, and if so, what you must install. It also explains the first steps to get going with a lab.

1.1 Requirements

All example code that I provide here has been tested under Python 3, using the PyCharm community edition UI (`https://www.jetbrains.com/pycharm/download/`). The Community Edition (scroll on the linked website) is free and runs reliably under all major OSs. If you do not have Python installed on your system, please do so. I can not support other programming languages in this course. If you elect to use a different (G)UI, then you do this at your own risk (in the sense that I cannot offer help and advice for anything but PyCharm).

If you are unfamiliar with Python or PyCharm, use the wondrous wild web to find one of the various tutorials. PyCharm helpfully enables you to create different "virtual environments" for different projects: in case you mess something up in one project, provided that you don't reuse the same venv for all projects, your other projects won't be affected.

1.2 What you should get out of this.

The lab gives you a taster of how differential cryptanalysis works, and a hint as to how structural attacks inspired cryptanalysts to use additional information in order to create strong, practically feasible attack vectors.

1.3 Getting going

Create a project under PyCharm (the default options from PyCharm should be fine, i.e. Python 3) and then add the supplied files for this lab (they are on Moodle) to your

project. It is up to you whether or not you connect/set up PyCharm with any Version control system that you might be using (it natively integrates with Github), or whether you do version control outside of PyCharm. But ensure you have some form of version control, to avoid the "I lost all my files before the deadline." embarrassment.

The file `toy_ciphers.py` implements various toy ciphers, and supplies access to substitution boxes. By toy ciphers I mean that the ciphers are deliberately weak, both in terms of specific choices of substitution boxes, key length, and number of rounds. They are easy to break in all respects, to the point, where you can break them with paper and pencil. The reason for this is so that you can, if necessary, use paper and pencil to get to grips with the idea behind differential cryptanalysis.

> Initially you should not look at the contents of the file `toy_ciphers.py`.

The file `diff_cryptanalysis.py` implements some functionality to facilitate differential cryptanalysis, and then, in the main function, implements the steps necessary to conduct a differential cryptanalysis of the simplest of the toy ciphers. For this purpose it calls `toy_ciphers.py` as an oracle. You can look at the contents of `diff_cryptanalysis.py`, but the lab sheet will examine this file closely in the next section.

Test if you can run `diff_cryptanalysis.py`. You may have to create a configuration for running it, or PyCharm will just do it for you. Running it multiple times should result in different outputs. If it runs successfully, you are ready to roll!

2 Differential Cryptanalysis of a Single Round Cipher

When attempting to understand complex ideas, it is often helpful to apply them to an example that is as simple as possible. Thus we will now look at exploiting differentials in the context of a very simple cipher.

2.1 Target cipher

Our initial target is a cipher that features a (randomly chosen) 3-bit substitution box, that is applied only once (thus there is a single "round"). To keep things as simple as possible, there is no permutation layer, bits go "straight through" the round and after an initial key addition, into the substitution box. There is a second key addition after

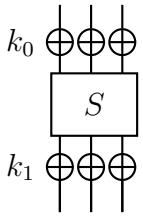

Figure 1: Single round toy cipher: encryption as implemented in `encrypt_sub_1r()`.

the substitution. Thus the encryption and decryption rules are:

$$Enc : c = S(m \oplus k_0) \oplus k_1$$

$$Dec : m = S^{-1}(c \oplus k_1) \oplus k_0$$

The substitution box S has been arbitrarily chosen (by me) as:

x	0	1	2	3	4	5	6	7
$S(x)$	6	4	5	0	7	1	3	2

In the file `toy_ciphers.py` there is an implementation of this cipher. For instance, the encryption function can be called via `toy_ciphers.encrypt_sub_1r(x)` for a plaintext $x \in \{0, \ldots, 7\}$. The file also defines the secret key that is used during encryption.

> For the sake of making this exercise "feel real", I suggest that you do not look into `toy_ciphers.py` for now (so you do not know the secret key used in the example).

2.2 Differential cryptanalysis

We briefly looked at the principle of differential cryptanalysis briefly. Now we shall explore its' working principle based a bit further. A core observation is that two inputs to an encryption that differ by an input difference $\delta_m(m_1, m_2) = m_1 \oplus m_2$ produce outputs with a difference of $\delta_c(c_1, c_2) = c_1 \oplus c_2$; and there are pairs of input-output differences that occur with a higher probability than other pairs. Such a pair is called a **differential** in the context of input-outputs to a substitution box, and a **differential characteristic** in the context of a cipher (I will not care about the distinction in this lab).

Why does any of this matter? Now comes another core observation: in a typical cipher, we "add", i.e., exclusive-or, key material before (and after) we send the state through a

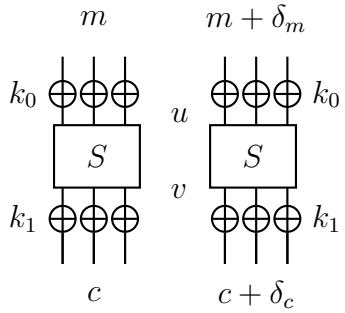

Figure 2: Differential in toy cipher `encrypt_sub_1r()`.

substitution, and the exclusive-or of the key is a commutative operation with regards to the key addition, and thus the differential remains unaffected.

Why does this matter? Consider two inputs $(m, m' = m \oplus \delta_m)$ to the encryption function of our toy cipher:

$$\begin{aligned} c &= S(m \oplus k_0) \oplus k_1 \\ c \oplus \delta_c &= S((m \oplus \delta_m) \oplus k_0) \oplus k_1 \end{aligned}$$

Now suppose that we have fixed a "likely" input-output difference (δ_m, δ_c), which implies that we know the values of the difference; suppose also that we happen to have a pair of messages and ciphertexts that result in this difference.

Because we choose a pair of inputs that is likely to result in a particular differential, we know now, with some probability, the actual inputs to the substitution box, which I denote by u in Fig. 2. In practice u can take several values, and we do not know which one really occur in the encryption (because we don't know k_0). But the number of values u to try might be considerably smaller than the number of keys to try, so this might (theoretically) constitute a break of the cipher.

Hang on, what are we trying out? Consider the cipher once more: you might see that because we know m and we have a small list of possible values for u according to our differential, we can use the relationships:

$$\begin{aligned} u &= m \oplus k_0 \\ v &= c \oplus k_1 \end{aligned}$$

to derive possible values for k_0 and k_1. With a bit of luck (or by trying some more inputs), and by intersecting the resulting values for the round keys, we should, without a brute force key search, be able to derive the keys!

2.2.1 Running the attack

Now that we have looked at the principle in "theory", it makes sense to run such an attack on the toy cipher that has a single round, but uses two independent keys.

Run the file `diff_cryptanalysis.py` without any modifications. You might get a result that is different from your neighbor because the script samples input randomly. Some of you might get a result where the actual cipher key is revealed uniquely, and some of you might get a result where there are four possible keys.

Difference distribution table. Let's look at the output that the file prints to the console. In the first step it prints out the difference distribution table for the substitution box that the cipher uses. The difference distribution table lists, for each input difference (rows) and output difference (column) the frequency by which it occurs.

Try and understand the table

What is the maximum number that you can spot in the table? Does it reach the theoretical maximum? What is the theoretical maximum here?
Why is the first row, as well as the first column so different to the rest of the table?
Why are all numbers in the table even?

Input-output difference to substitution box. Next, the file prints, for a selected difference pair the corresponding input/output pairs of the substitution box. These are the possible pairs of (u, v) that we need to find the keys.

How does the probability of the differential impact on the attack complexity?

Consider the differential that is chosen in the example. How does it impact on the attack complexity?

Input-output pairs. Now the script does something that would **not** happen in an actual attack: it actually lists all pairs of plaintexts and ciphertexts that correspond to the chosen differential.

In practice ...

... this would be done how? Or not at all?

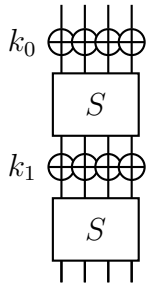

Figure 3: Two round toy cipher `encrypt_sub_2r()`.

The actual attack. The script now uses the first of the good inputs to actually launch the attack. Because we know that we some good probability, we have that (u, v) is our chosen characteristic for the input m and resulting ciphertext c, we can now use the equations from before the derive candidates for k_0, k_1. There are four possible (u, v) pairs, thus, there are four key candidates.

The final step. Because we can, at best, narrow down the key space to four candidates, the script samples a further message at random, and encrypts it with the encryption oracle. It also uses the candidate keys to do an encryption, and checks the results with the results from the oracle. In some cases, this allows to actually identify the secret key tuple (k_0, k_1). If it does not in your case, just rerun the attack, and eventually you will get a single good key tuple.

Now you can check the actual key tuple in `toy_ciphers.py`.

3 Differential Cryptanalysis of a Two Round Cipher

The next step of this lab requires you to extend the existing attack to a cipher that is slightly more complex than the first toy cipher. It is called via `toy_ciphers.encrypt_sub_2r()`. It has a four bit state, and thus, a four bit substitution box. It also has a further substitution layer after the second key addition. Thus the encryption and decryption rules are:

$$Enc : c = S(S(m \oplus k_0) \oplus k_1)$$
$$Dec : m = S^{-1}(S^{-1}(c) \oplus k_1)) \oplus k_0$$

Your turn as a cryptanalyst.

Your challenge is now simple: adapt the existing script to perform a differential cryptanalysis on this slightly more complex toy cipher.

I give you a few hints: firstly, most of the script can remain largely unchanged; but remember to adapt any call that is made to the 3 bit substitution box to the 4 bit substitution box, and any call that is made to an encryption function to the encryption function of the two round cipher, which is `encrypt_sub_2r()`. Only a relatively simple observation is needed to deal with the fact that there is another substitution layer in play.

4 Differential Cryptanalysis of a Two Round SPN Cipher

The final step of this lab is somewhat open ended. Real world ciphers feature not only substitution boxes, but also permutations. Let us stick with just two rounds initially, and consider how permutations may influence both attack strategy and attack complexity. There are (at least) two options for a permutation layer. The permutation could be small, and only apply to a small part of the state (thus it is applied to multiple parts of the state independently). The permutation could be large, and perhaps even apply to the entire state of the cipher. The latter situation is visualised via an example for such an encryption function in Fig. 4.

Consider the permutation size

Is a small permutation sensible? What if we chose the size of the permutation in line with the output size of the substitution box?

The file `toy_cipher.py` includes two permutations: one that supports a state of 8 bits (thus two four-bit substitution boxes, like depicted in Fig. 4) and one that supports a state of 16 bits (thus four four-bit substitution boxes). There is also a key supplied for an 8 bit state.

Extend the existing encryption and decryption function

Write a toy cipher that implements the two round SPN structure depicted in Fig. 4. With this new two-round SPN toy cipher, consider how to construct a differential for the cipher. Then, try and implement a differential attack.

Once you are done with this, you could also consider doing this for a 16-bit state, and you could of course increase the number of rounds.

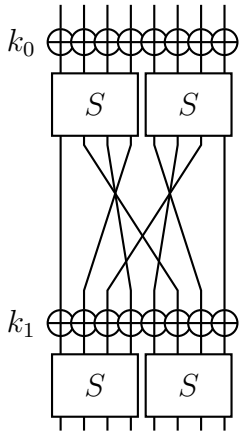

Figure 4: An exemplary two-round SPN toy cipher.

Extend the toy example further

How does the number of rounds impact on the attack strategy and complexity? Imagine (at least, or if you can, implement) a further extension of the cipher that has more than two rounds.

5 Differential Fault Analysis

The previous sections should have helped you to understand that the more complex a cipher gets, and therefore, if the design is good, the more "secure" it gets, the smaller there is the chance of finding a good differential characteristic. Thus for real-world ciphers such as AES or DES, the computational cost for the best known differential attack is either impractically high, or, one can even show that it is lower bounded by something that is impractically high.

Consequently, differential cryptanalysis is not practical, and even variations that are somewhat stronger (or better suited in terms of other structural properties of modern ciphers), are still impractical. This is a somewhat frustrating situation for cryptanalysts.

However, in the mid 1990's a student, had the (unconventional) idea to consider what

would happen if additional information was available to cryptanalysts (leading ultimately to the idea of side channel attacks). Simultaneously, other researchers considered a related but different idea: what if a crypto device could be actively manipulated so that an "error" or "fault" would occur during the computation of some encryption (or decryption). Later in this course we will look at this more generically, but for now let us consider a very concrete example for DES.

Run DES

The scripts for this lab also include DES encryption. I did not write this myself, but utilise the implementation from `https://medium.com/@ziaullahrajpoot/data-encryption-standard-des-dc8610aafdb3` (it looks suitable lay-man so it is easy to read). Get this to run and then extend the implementation so that you can specify to alter a specific bit in $R15$.

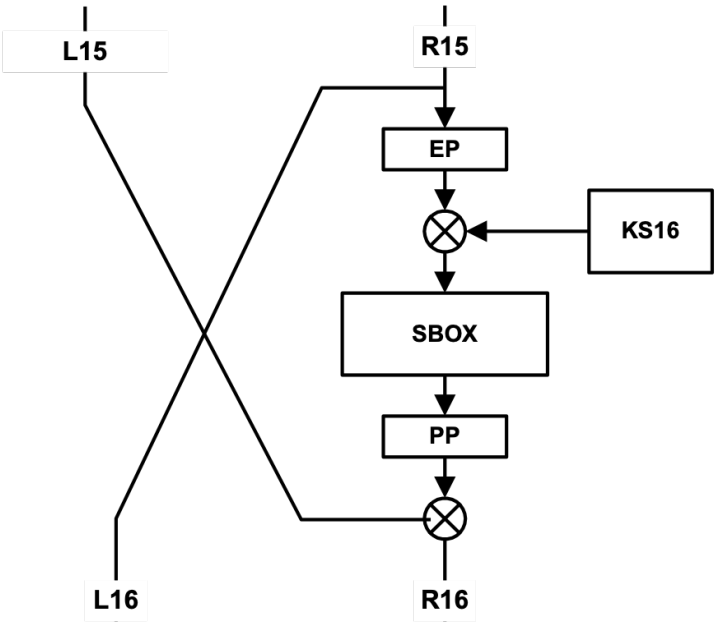

Figure 5: DES final round.

We encrypt a reference message m to get a reference cipher text c. Then we encrypt the same m again and fault the device during encryption, such that a single bit flips in $R15$, leading to a cipher text c'.

The fact that a fault occurred will be visible as a single bit flip in $L16$ of (c') so the adversary knows which bit in $R15$ has been flipped (see Fig. 5).

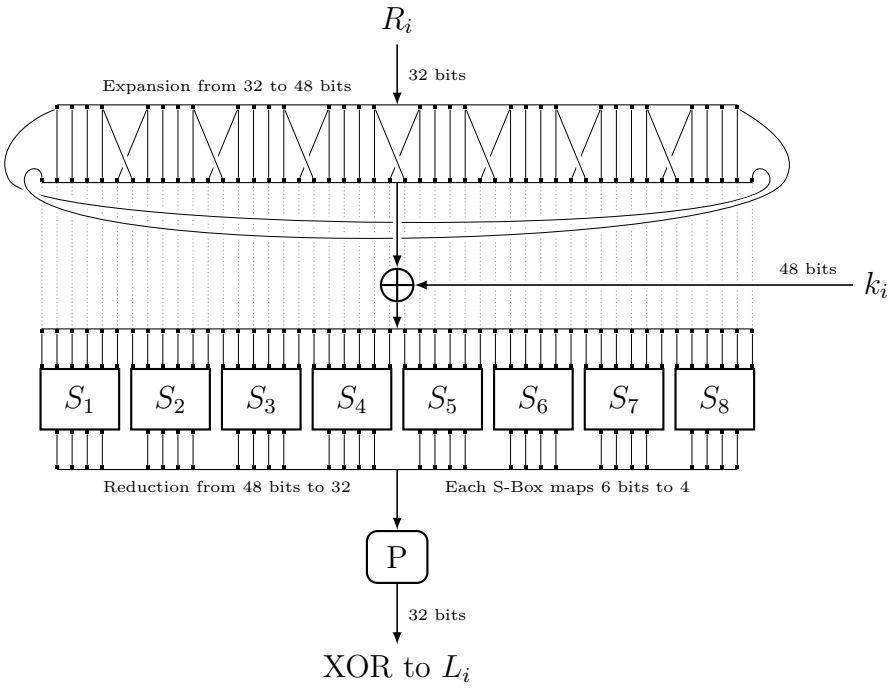

Figure 6: Detailed illustration of DES round function (credits go to Roberto Avanzi)

The bit flip in $R15$ may be expanded into two bit flips because of the expansion permutation. But mostly the input to a single S-Box will have been altered only. This should be clear from the more detailed picture for the DES round function 6 (this illustration shows the actions of the permutations and substitution boxes that apply to the right half of the round function).

Explore the effect of bit flips

Using your modified DES encryption function, play a bit around with flipping different bits in $R15$ and observe the effect via the expansion permutation. Convince yourself that at most two bits can be affected.

Consider the reference ciphertext: $R16 = P(F((R15, K16)) \oplus L15$ and the faulty ciphertext: $R16' = P(F((R15', K16)) \oplus L15$

The adversary knows $R16$ and $R16'$ because they are in the ciphertext. The adversary also knows also $R15$ and $R15'$ because they become the $L16$ (and $L16'$) in the ciphertext.

Remember that the fault affects at most two bits after the expansion permutation, which

means that it can affect at most two S-boxes. But let's assume that just a single S-box gets affected by it for now. The adversary does not know $L15$ (and $L15'$). But now let's think back to differential cryptanalysis: the idea there was that we could check which keys satisfy certain input-output differences that occur with high probability when considering pairs of messages. Our fault attack strategy in fact realises such a differential: we *force* a very specific input difference at the beginning of the final round, and we can then observed the corresponding output difference (for a given but unknown key). Because we ensure a small difference in the input, this difference can only affect a small part of the round key, and it will leave the rest unaffected. Therefore we can brute force test just a small part of the round key, and see which value(s) can produce the observed differential.

In our particular example, the adversary can test which keys satisfy $R16 \oplus R16' = P(F((R15, K16)) \oplus P(F((R15', K16))$.

Implement the fault attack

You now have everything to actually implement the fault attack: use your modifications to flip a single bit in $R15$, and then, based on known what happens because of the expansion permutation, write a script that enumerates 6 bits of the last round key and checks which ones satisfy the above equation.
Consider how often this has to be repeated (with different faults) until the entire round key is known. How do you get the cipher key once you have the final round key?