Differential Cryptanalysis Tutorial

1 Abstract

This tutorial is a detailed and interactive introduction to differential cryptanalysis, a subtle, beautiful, and powerful method of breaking some of the world's strongest ciphers. Unlike other currently available tutorials, it avoids needless math notation, makes very few assumptions about prior knowledge, does not gloss over details, and is written to be understood by novices with an interest in cryptography.

Before continuing, know that there are two major prerequisites:

- You must be comfortable with binary and hexadecimal notation, especially how to convert between them
- We will be using differential cryptanalysis to break symmetric-key block ciphers. You should be familiar with this family of ciphers before learning a technique to break them

2 Introduction

Differential cryptanalysis was first discovered in the 1970s by the designers of the Data Encryption Standard (DES) at IBM. After a consultation with the NSA, the DES designers opted to keep the technique a secret, because its release "would weaken the competitive advantage the United States enjoyed over other countries in the field of cryptography" [1]. It remained classified until it was independently rediscovered and published in the early 1990s by Eli Biham and Adi Shamir, who are now credited with its discovery [2].

Before getting to the definition, it is important to note that differential cryptanalysis is what's known as a *chosen plaintext* attack. This means that we, as attackers, may choose which plaintexts, and how many, we want encrypted by the cipher using the same secret key. This may not always be true in the real world since the same key often isn't used for a very long time, and during that time it can be difficult or impossible to encrypt whichever plaintext you wish. Nevertheless, we must make this assumption for the attack to work.

Note that the final goal of differential cryptanalysis is not simply to decrypt a secret message. We want to get the secret key that was used to encrypt the message so that we can decrypt any other message sent using the same key.

Block ciphers work by repeatedly applying a set of transformations (key addition, substitution, permutation, etc.) to a plaintext until a ciphertext is obtained. The security of the cipher depends on the attacker never knowing/guessing any intermediate value during encryption; we can only see the plaintext (remember, chosen plaintext) and ciphertext, nothing more. As you will see in section 4 of this tutorial, if an intermediate value could be guessed, the security of the cipher would be immediately compromised.

However, as we will explain thoroughly in this tutorial, we *can* know/guess the intermediate value of the *difference of two plaintexts* during encryption (this is why the technique is called *differential* cryptanalysis). This is a very subtle point and one that can be difficult to understand initially, but is crucial to understanding how differential cryptanalysis can recover the secret key and therefore break the cipher.

As you will see, this "difference" is measured via a simple bitwise XOR of the two plaintexts.

Now that we have the basics, let's get to breaking our first cipher!

3 Components of Block Ciphers

This section is a short refresher on the components which make up the block ciphers we'll be breaking. If you feel confident in your knowledge of key addition, substitution, and permutation, you are encouraged to skip this section.

3.1 Block Ciphers

Block ciphers are symmetric key encryption algorithms that operate on a fixed-length *block* of bits. They take a fixed-length block as input and output an encrypted block of the same length. Decryption is done by running the encrypted block backward through the cipher. There are many ways to construct block ciphers, but many are made up of a repeated sequence of three simple transformations: key addition, substitution, and permutation.

3.2 Key Addition

Key addition is the step in which we introduce the secret/symmetric key. We do this by a bit-wise XOR of the key with the block. Remember that the XOR of two bits is defined by this function:

b_1	b_2	$b_1 \oplus b_2$
0	0	0
0	1	1
1	0	1
1	1	0

If, for example, our secret key was the bit string 11010010 and we had a block of bits, which *before* the key addition step had a value of 01101110 (block size is one byte), the value of the block *after* the key addition would be:

$$\begin{array}{r} 01101110 \\ \oplus\ 11010010 \\ \hline 10111100 \end{array}$$

3.3 Substitution

Substitution is the step in which some bits are directly substituted for others. We define how these bits are substituted with a substitution-box (S-box). Let's define a simple 4-bit S-box:

S-box															
Input:	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
Output:	e	4	d	1	2	f	b	8	3	a	6	c	5	9	0	7

Note that both the input and output are represented using hexadecimal notation. As an example, if the input to this S-box is 1011 (0xb), then we'll get an output of 1100 (0xc) by simply looking it up in the table.

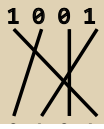
The S-box that a cipher uses is almost always released to the public. This is done in accordance with Kerckhoffs's principle which says that a cryptosystem should be secure even if everything about it—except for the secret keys—is public knowledge. Just because the S-box is public and therefore easily reversible does not mean that substitution is pointless or insignificant. In fact, it is often the only part of a cipher that directly protects against differential cryptanalysis. Consequently, we'll discuss substitution in much more depth throughout this tutorial.

3.4 Permutation

Permutation is a step in which we change the position of the bits. We define how these bits are permutated with a permutation-box (P-box). Let's define a simple 4-bit P-box:

P-box				
Input:	0	1	2	3
Output:	3	0	2	1

This P-box says that the bit at position 0 will move to position 3, the bit at position 1 will move to position 0, the bit at position 2 will stay at position 2, and the bit at position 3 will move to position 1. For example, if we input the nibble 1001 into the P-box, we will get an output of 0101. P-boxes, unlike S-boxes, are easy to represent visually:



```
0 1 0 1
```

Just like S-boxes, P-boxes are released to the public. Though S-boxes serve as the main protection against differential cryptanalysis, P-boxes also play a vital role which we will discuss later.

4 Toy Cipher

> *If you can't solve a problem, then there is an easier problem you can't solve: find it.*
>
> —Pólya/Conway

Let's begin by breaking the simplest block cipher worth breaking which we'll call the "toy cipher." The toy cipher is *just* complicated enough to make differential cryptanalysis a valid method to break it, but no more. Note that it is trivial to break the toy cipher by brute force. Nevertheless, it will serve as a great introduction to the main concepts behind differential cryptanalysis.

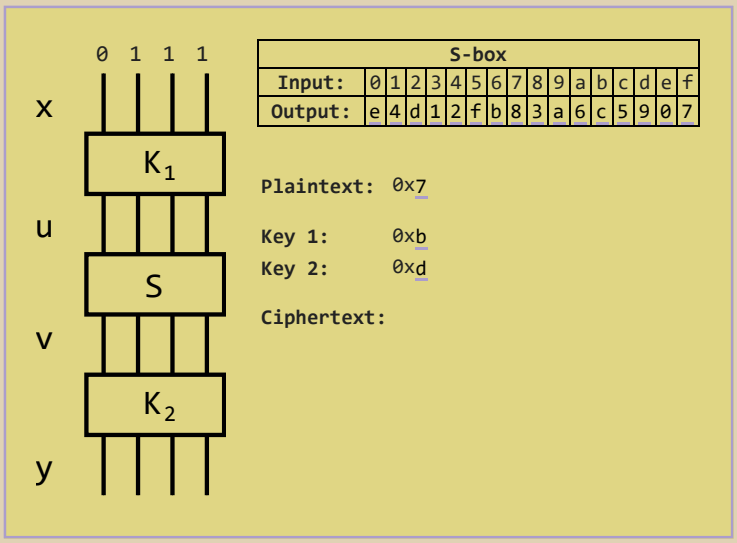
4.1 Toy Cipher Definition

The toy cipher has a block size of 4 bits (also known as a nibble), meaning that we'll input 4 bits into the cipher and it will return 4 bits of ciphertext. The toy cipher has a 4-bit S-box and two separate 4-bit keys (k_1 and k_2).

Now let's look at how we can encrypt a nibble with the toy cipher. There are three steps:

1. We XOR the plaintext with k_1
2. Feed that result into the S-box
3. XOR the output of the S-box with k_2

These simple steps give us the final ciphertext. The cipher is much easier to understand visually so look to the figure below. The values underlined with purple can be changed, so play around with them to see how the cipher is affected. The red lines represent the 1s and the black lines the 0s as the nibble makes its way through the cipher. Also note the labels x, u, v, and y on the left (x is the value of the plaintext, u is the value after the first key addition, etc.). Those labels will make it easier to discuss the cipher when we get to differential cryptanalysis so keep them in mind.



4.2 Differential Cryptanalysis of the Toy Cipher

Much of the following is based on King's webpage: "Differential Cryptanalysis Tutorial" [9] as well as Stinson's textbook *Cryptography: Theory and Practice* [10].

We must begin by selecting which S-box we'll be using for the toy cipher. Remember that S-boxes are public knowledge and that, once the cipher is defined, they do not change. So, we'll pick an arbitrary S-box for our toy cipher.

S-box															
Input:	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
Output:	e	4	d	1	2	f	b	8	3	a	6	c	5	9	0	7

Differential cryptanalysis stems from the analysis of the XOR of two plaintexts as they travel through the cipher. XOR is used because it tells us where two plaintexts differ. If their XOR at some bit position is 1, then those two plaintexts differ at that bit position.

For now, instead of focusing on the cipher as a whole, we'll simply analyze the S-box. Let's first choose an XOR value to analyze: 1011. We can make a table of all possible pairs of S-box inputs, call them x and x^*, which XOR to 1011 (in other words, all inputs which differ in the first, third, and fourth bit positions).

x	x^*	$x \oplus x^*$
0000	1011	1011
0001	1010	1011
0010	1001	1011
0011	1000	1011
0100	1111	1011
0101	1110	1011
0110	1101	1011
0111	1100	1011
1000	0011	1011
1001	0010	1011
1010	0001	1011
1011	0000	1011
1100	0111	1011
1101	0110	1011
1110	0101	1011
1111	0100	1011

Now, let's consider the XOR of the S-box *outputs* of both x and x^*. Let's name those outputs y and y^* respectively. In other words, if we put the input x through the S-box we defined earlier, we'll get y, and if we input x^*, we'll get y^*.

x	x^*	$x \oplus x^*$	y	y^*	$y \oplus y^*$
0000	1011	1011	1110	1100	0010
0001	1010	1011	0100	0110	0010
0010	1001	1011	1101	1010	0111
0011	1000	1011	0001	0011	0010
0100	1111	1011	0010	0111	0101
0101	1110	1011	1111	0000	1111
0110	1101	1011	1011	1001	0010
0111	1100	1011	1000	0101	1101
1000	0011	1011	0011	0001	0010
1001	0010	1011	1010	1101	0111
1010	0001	1011	0110	0100	0010
1011	0000	1011	1100	1110	0010
1100	0111	1011	0101	1000	1101
1101	0110	1011	1001	1011	0010
1110	0101	1011	0000	1111	1111
1111	0100	1011	0111	0010	0101

Here we see the heart of differential cryptanalysis. Examine the values of $y \oplus y^*$ and notice the repetitions. Let's create another table to see how many times each value appears in the S-box output XORs.

Output XOR	Appearances
0000	0
0001	0
0010	8
0011	0
0100	0
0101	2
0110	0
0111	2
1000	0
1001	0
1010	0
1011	0
1100	0
1101	2

1110	0
1111	2

Consider what this table is really saying. If we pick two random plaintexts which XOR to 1011, put each of them through the S-box, and then XOR those S-box outputs, 8 out of 16 times, they will XOR to 0010. They will XOR to 0111 2 out of 16 times. They will never XOR to 1010. This bias towards certain XORs is what makes differential cryptanalysis possible.

So far we've only considered the input XOR of 1011. We can repeat the same process for every possible input XOR and build what is called a *difference distribution table* for the given S-box. The rows in this table represent the S-box input XOR, the columns represent the S-box output XOR, and the cell values are the number of appearances of that output XOR given the input XOR. Note that the XORs are represented in hexadecimal instead of binary (e.g. you can find the previous example of 1011 in the row labeled b).

Input XOR	Output XOR															
	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	0	0	0	2	0	2	4	0	4	2	0	0
2	0	0	0	2	0	6	2	2	0	2	0	0	0	0	2	0
3	0	0	2	0	2	0	0	0	0	4	2	0	2	0	0	4
4	0	0	0	2	0	0	6	0	0	2	0	4	2	0	0	0
5	0	4	0	0	0	2	2	0	0	0	4	0	2	0	0	2
6	0	0	0	4	0	4	0	0	0	0	0	0	2	2	2	2
7	0	0	2	2	2	0	2	0	0	2	2	0	0	0	0	4
8	0	0	0	0	0	0	2	2	0	0	0	4	0	4	2	2
9	0	2	0	0	2	0	0	4	2	0	2	2	2	0	0	0
a	0	2	2	0	0	0	0	0	6	0	0	2	0	0	4	0
b	0	0	8	0	0	2	0	2	0	0	0	0	0	2	0	2
c	0	2	0	0	2	2	2	0	0	0	0	2	0	6	0	0
d	0	4	0	0	0	0	0	4	2	0	2	0	2	0	2	0
e	0	0	2	4	2	0	0	0	6	0	0	0	0	0	2	0
f	0	2	0	0	6	0	0	0	0	4	0	2	0	0	2	0

This table gives us probabilities for what happens to the XOR of two plaintexts as they individually travel through the S-box. However, besides the S-box, the cipher also includes key addition. So, before we get to breaking the keys, we must consider what happens to the XOR of two plaintexts as we put them through key addition.

Let's consider two plaintexts:

$x = 0111$

$x^* = 1100$

$x \oplus x^* = 1011$

Now let's choose a key that we'll add to both:

$k = 1010$

$x \oplus k = 0111 \oplus 1010 = 1101$

$x^* \oplus k = 1100 \oplus 1010 = 0110$

Now let's see what value we get when we XOR the outputs of this key addition:

$(x \oplus k) \oplus (x^* \oplus k) = 1101 \oplus 0110 = 1011$

You should see that both $x \oplus x^*$ and $(x \oplus k) \oplus (x^* \oplus k)$ come out to the same value. This should be apparent because of a few mathematical facts: XOR is associative, XOR is commutative, a value XORed by itself is zero, and a value XORed by zero is itself. We can therefore prove the previous statement like so:

$(x \oplus k) \oplus (x^* \oplus k)$

$= x \oplus x^* \oplus k \oplus k$

$= x \oplus x^* \oplus 0$

$= x \oplus x^*$

This is an incredibly useful realization. Given the XOR of two plaintexts, we can predict with 100% certainty what their XOR will be after key addition—it will always stay the same. Remember that with the S-box, the best prediction we have is only 50%, and many are much worse than that.

The fact that keys become essentially useless when we use this attack should convince us that we're on the right track. This is also why S-boxes are an absolutely *essential* part of many ciphers. You will see in the next section that even when we also consider permutations, S-boxes are still the only protection against attacks like differential cryptanalysis.

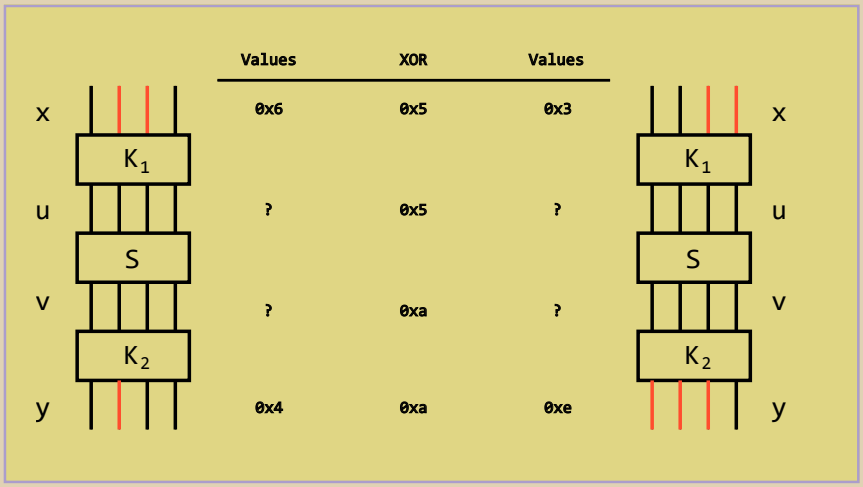
Now we can begin breaking the keys.

Because differential cryptanalysis is a chosen plaintext attack we'll need to encrypt plaintexts in order to break the keys. For that, however, we need to choose which keys will be used for encryption. For now let's assume that k_1 = 0xb and k_2 = 0xd. Of course, when differential cryptanalysis is used in the real world the keys are unknown to the attackers. But some keys *need* to be chosen and this is just a tutorial so we might as well know them beforehand.

The first step is to pick a *differential characteristic* (this is just an input and output XOR pair). Let's just choose a random one that has a non-zero value in the difference distribution table: 0x5 → 0xa.

Next, we must find a pair of plaintexts that XOR to 0x5 and whose associated ciphertexts XOR to 0xa. Such a pair is called a *good pair* for that given differential characteristic. You will see how a good pair is useful in a second.

Assuming k_1 = 0xb and k_2 = 0xd, one good pair is 0x6 and 0x3. They XOR to 0x5. They encrypt to 0x4 and 0xe respectively, and 0x4 ⊕ 0xe = 0xa. This matches our differential characteristic. You can confirm this using the toy cipher visualization at the beginning of this section or by hand.



Now let's see how this good pair will travel through the cipher. Above is a figure showing what is being explained here. We'll be referencing the letters on the sides of the figure.

We know that at x the value on the left is 0x6, the value on the right is 0x3, and their XOR is 0x5. We also know that at y the value on the left is 0x4, the value on the right is 0xe, and their XOR is 0xa. These are all values that we've handpicked. Now, what is the XOR at u? As explained earlier, the XOR doesn't change over key addition, so it remains at 0x5. Since the XOR at y is 0xa, we can follow the same logic and conclude that the XOR at v must also be 0xa.

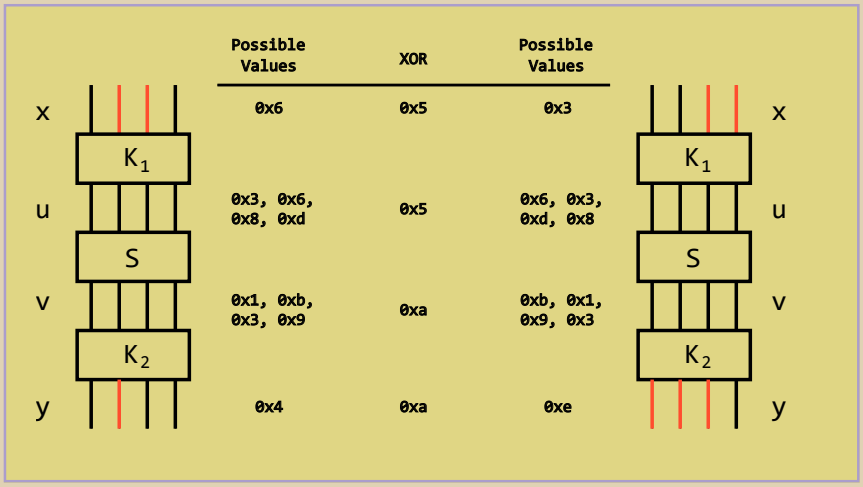
But what are the values at u and v?

Assuming the keys are unknown to us, there's no easy way to tell. However, we do know that whatever the values at u are, they must XOR to 0x5 and whatever the values at v are, they must XOR to 0xa. Well, what are the possible inputs to an S-box that XOR to 0x5 and whose corresponding S-box outputs XOR to 0xa? We've already calculated them when making the difference distribution table! The calculation was not shown previously (we only saw the one for an S-box input XOR of 0xb) so let's see it now:

x	x^*	$x \oplus x^*$	y	y^*	$y \oplus y^*$
0000	0101	0101	1110	1111	0001
0001	0100	0101	0100	0010	0110
0010	0111	0101	1101	1000	0101
0011	0110	0101	0001	1011	1010
0100	0001	0101	0010	0100	0110
0101	0000	0101	1111	1110	0001
0110	0011	0101	1011	0001	1010

0111	0010	0101	1000	1101	0101
1000	1101	0101	0011	1001	1010
1001	1100	0101	1010	0101	1111
1010	1111	0101	0110	0111	0001
1011	1110	0101	1100	0000	1100
1100	1001	0101	0101	1010	1111
1101	1000	0101	1001	0011	1010
1110	1011	0101	0000	1100	1100
1111	1010	0101	0111	0110	0001

The highlighted rows give us the four possible S-box input pairs which XOR to 0x5 and whose respective outputs XOR to 0xa. With this information, we can update the previous figure:



If we know the value at x, u, v, and y, we can simply calculate the value of the first and second keys! By definition, we know:

$$x \oplus k_1 = u$$
$$v \oplus k_2 = y$$

Using simple XOR rules we can conclude that:

$$k_1 = x \oplus u$$
$$k_2 = v \oplus y$$

Since there are only four possible values of u and v, we've narrowed down the possibilities for the two keys from 256 to only 4 (in the table, the subscripts l and r refer to the left and right values in the figure above):

x_l	u_l	v_l	y_l	x_r	u_r	v_r	y_r	$x_l \oplus u_l = x_r \oplus u_r = k_1$	$v_l \oplus y_l = v_r \oplus y_r = k_2$
0x6	0x3	0x1	0x4	0x3	0x6	0xb	0xe	0x5	0x5
0x6	0x6	0xb	0x4	0x3	0x3	0x1	0xe	0x0	0xf
0x6	0x8	0x3	0x4	0x3	0xd	0x9	0xe	0xe	0x7
0x6	0xd	0x9	0x4	0x3	0x8	0x3	0xe	0xb	0xd

You can see that one of the possible key guesses is correct! Remember that we set k_1 = 0xb and k_2 = 0xd earlier. Had we not known the keys beforehand, we would have had to loop through these 4 possible key combinations and find the keys that will give us the correct ciphertext for a given plaintext.

Let's do a quick review of the steps we took to get here:

- We started by making the difference distribution table for the given S-box
- We then randomly picked a differential characteristic with a non-zero probability
- We found a plaintext-ciphertext pair that matched that differential characteristic
- Using that good pair we narrowed down the values at u and v to only four possibilities

- Those also narrowed down the possible key combinations to four
- Since we already knew which one was correct, we just picked it out. Had we not known, we would have had to verify all four combinations with some plaintext-ciphertext pairs

All of these steps are implemented in a Python script available in the GitHub repository. You are *highly* encouraged to check it out, as it was made to accompany this tutorial and follows the exact same steps that we've taken. You can change the S-box, key combinations, see some of the implementation details which were not discussed, and play around with the code in order to gain a deeper understanding of this section.

Though the toy cipher served as a great introduction to many of the concepts and vocabulary of differential cryptanalysis, it's called the *toy* cipher for a reason. It is so easily broken by brute force that these steps seem unnecessarily complicated. In the next section, you will see a more realistic application of differential cryptanalysis to break a non-trivial cipher.

> *Here I pause. If you wish to walk no farther with me, reader, I cannot blame you. It is no easy road.*
>
> —Wolfe

5 SPN

> *Simplicity does not precede complexity, but follows it.*
>
> —Perlis

This section expands on the knowledge gained in the previous. We will discuss how differential cryptanalysis can be applied to more complicated ciphers which resemble the Advanced Encryption Standard (AES). Though the heart of the method will remain the same, the complexity of these ciphers will require the introduction of some new concepts.

5.1 SPN Definition

A substitution-permutation network (SPN) is any kind of cipher that uses both substitution and permutation together with key addition. Because an SPN is so general (it can have any block size, S-box size, number of rounds, etc.) and because differential cryptanalysis depends a lot on the structure of the cipher, we'll be focusing on a particular SPN. Nevertheless, these steps could be taken for any kind of SPN.

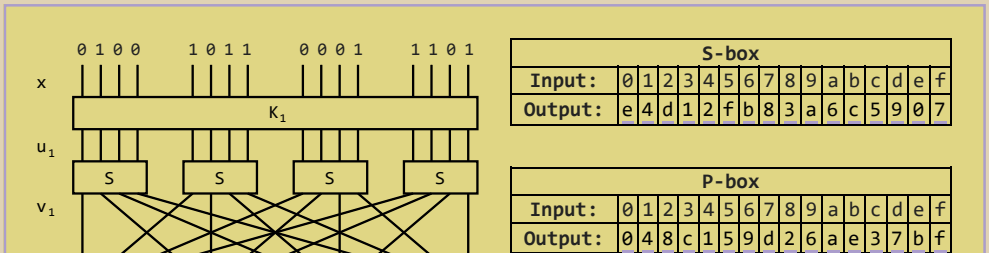
We will be using an SPN with a block size of 16 bits. It will have a total of four rounds. Each round, except for the fourth, will consist of the following operations:

1. Key addition with a 16-bit key
2. We will then consider the 16 bits as 4 nibbles and put each of those nibbles through a 4-bit S-box
3. Lastly, we will permutate the block with a 16-bit P-box

The fourth round will be the same as the previous, but without the permutation (because it provides no additional security). We will also end the cipher with one last key addition.

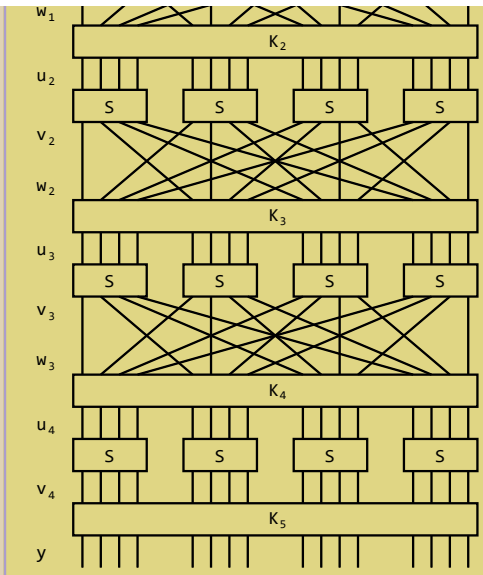
This means that we will need five 16-bit keys. The total number of key bits we need to break is therefore 80. A simple brute force is beyond most computers as this means there are $2^{80} = 1.21 \times 10^{24}$ possible key combinations. It should be apparent that breaking a cipher like this isn't trivial. However, we will show how we can use differential cryptanalysis to break it easily.

As with the toy cipher, this SPN is easier to understand if you look to the figure below. Remember that the values underlined with purple can be modified. The labels on the side are similar to the ones for the toy cipher with a few changes: we use subscripts to indicate the round number and a new label *w* is added for the value after the permutation.



S-box															
Input:	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
Output:	e	4	d	1	2	f	b	8	3	a	6	c	5	9	0	7

P-box																
Input:	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
Output:	0	4	8	c	1	5	9	d	2	6	a	e	3	7	b	f



Plaintext: 0x4b1d

Key 1: 0x1a6d
Key 2: 0x2ac2
Key 3: 0x452f
Key 4: 0x6ff1
Key 5: 0xb520

Ciphertext:

5.2 Differential Cryptanalysis of an SPN

This section is partly based on Howard M. Heys's "A Tutorial on Linear and Differential Cryptanalysis" [8].

We begin by selecting the S-box and P-box for the cipher. We'll keep the S-box from the previous section and the default P-box from the figure above.

S-box															
Input:	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
Output:	e	4	d	1	2	f	b	8	3	a	6	c	5	9	0	7

P-box																
Input:	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
Output:	0	4	8	c	1	5	9	d	2	6	a	e	3	7	b	f

The first step, just like before, is to calculate the difference distribution table. Since we're using the same S-box, it remains unchanged from the previous section.

Input XOR	Output XOR															
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	0	0	0	2	0	2	4	0	4	2	0	0
2	0	0	0	2	0	6	2	2	0	2	0	0	0	0	2	0
3	0	0	2	0	2	0	0	0	0	4	2	0	2	0	0	4
4	0	0	0	2	0	0	6	0	0	2	0	4	2	0	0	0
5	0	4	0	0	0	2	2	0	0	0	4	0	2	0	0	2
6	0	0	0	4	0	4	0	0	0	0	0	0	2	2	2	2
7	0	0	2	2	2	0	2	0	0	2	2	0	0	0	0	4
8	0	0	0	0	0	0	2	2	0	0	0	4	0	4	2	2
9	0	2	0	0	2	0	0	4	2	0	2	2	2	0	0	0
a	0	2	2	0	0	0	0	0	6	0	0	2	0	0	4	0
b	0	0	8	0	0	2	0	2	0	0	0	0	0	2	0	2
c	0	2	0	0	2	2	2	0	0	0	0	2	0	6	0	0
d	0	4	0	0	0	0	0	4	2	0	2	0	2	0	2	0
e	0	0	2	4	2	0	0	0	6	0	0	0	0	0	2	0
f	0	2	0	0	6	0	0	0	0	4	0	2	0	0	2	0

Before we move on we need to discuss permutation. We've considered what happens to the XOR of two plaintexts when they go through key addition (it doesn't change) and when they go through substitution (we can guess it based on the probabilities in the difference distribution table). However, we have not considered what happens when they go through permutation.

Let's consider two plaintexts:

x = 1100 1110 1001 1011

x^* = 0011 1110 0101 0111

$x \oplus x^*$ = 1111 0000 1100 1100

Now let's send both of these plaintexts through the P-box:

permutate(x) = 1111 1100 0101 0011

permutate(x^*) = 0100 0111 1101 1011

permutate(x) $\oplus$ permutate(x^*) = 1011 1011 1000 1000

We can see that, unlike key addition, the XOR value before the permutation operation is different than the XOR of the value after the permutation, which could make things difficult. However, what happens if we send the XOR value before the permutation through the P-box?

permutate($x \oplus x^*$) = permutate(1111 0000 1100 1100) = 1011 1011 1000 0001

You should see that both permutate(x) $\oplus$ permutate(x^*) and permutate($x \oplus x^*$) come out to the same value.

For some intuition as to why this is true, notice that both of the plaintexts will travel through the same P-box. If their XOR is 1 at some bit position, it will remain 1 after applying the permutation. It will just move to a different bit position determined by the P-box. The same logic applies when their XOR is 0. Therefore, while the XOR doesn't remain the same, if we know the input XOR (which, remember, we can guess with significant certainty) we can predict the output XOR with 100% certainty by simply putting the XOR through the permutation. One way of writing this more mathematically is:

permutate(x) $\oplus$ permutate(x^*) = permutate($x \oplus x^*$)

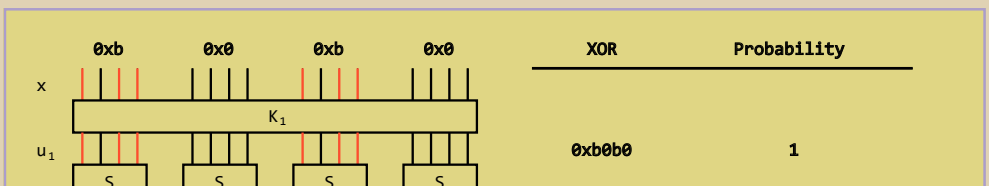
The significance of this might not be immediately obvious, but remember that the actual values of x and x^* are essentially impossible to guess, which makes computing permutate(x) $\oplus$ permutate(x^*) difficult. The value of $x \oplus x^*$, on the other hand, is what we've been keeping track of this whole time! The P-box, like the S-box, is public knowledge so computing permutate($x \oplus x^*$) is very easy.

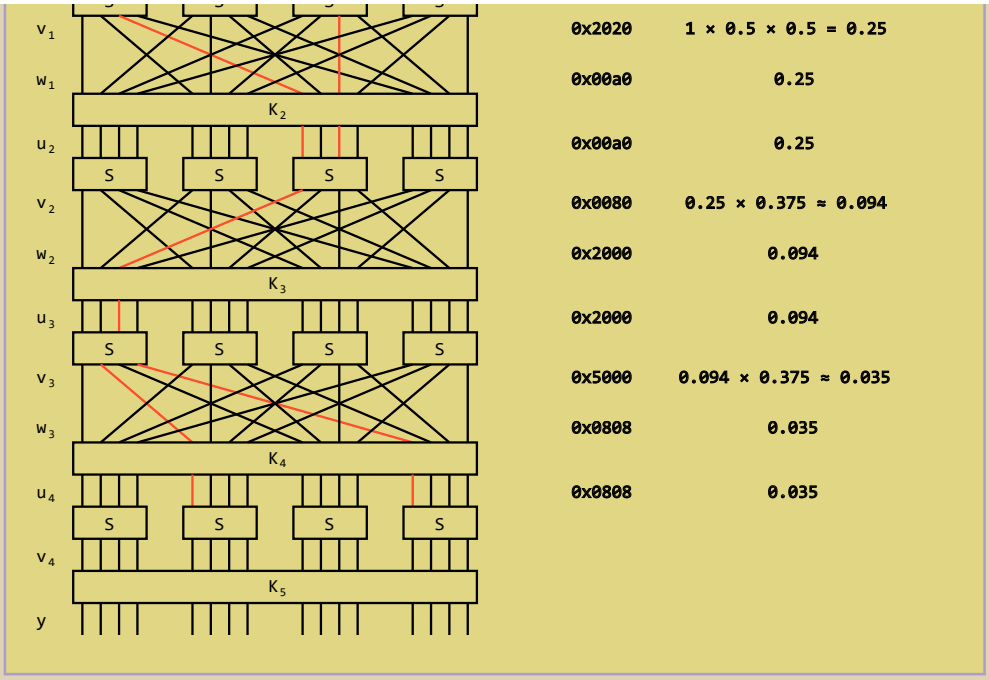
Now that we have discussed all three components let's quickly review how the XOR interacts with them:

- Key addition: the XOR of two plaintexts *does not* change over key addition. We can therefore predict the output XOR with 100% certainty.
- Substitution: the XOR of two plaintext *does* change over substitution. However, it does so probabilistically based on the input XOR and the difference distribution table of the S-box, which gives us some ability to predict it.
- Permutation: the XOR of two plaintext *does* change over permutation. However, we can predict it with 100% certainty by simply sending the input XOR through the P-box.

We will now introduce the concept of differential trails (sometimes also called differential characteristics), which expands the idea of differential characteristics we talked about in the previous section. Differential trails are simply a way to keep track of the most likely XOR as two plaintexts make their way through multiple rounds of key addition, substitution, and permutation. The best way to understand it is via an example.

We begin by picking an input XOR to the cipher: 0xb0b0. Why we picked this input XOR in particular will be explained later on. Below is a figure showing the differential trail explained in the following paragraph. Note that the red lines in the figure do not represent ones in a plaintext but *ones in the XOR of two plaintexts as they travel through the cipher*.





We start with two plaintexts which XOR to 0xb0b0. As they make their way through the key addition, their XOR doesn't change.

Next they go through the S-boxes. The S-box input XOR in their second and fourth nibble is 0 (the plaintexts are equal there) so their XOR doesn't change in those nibbles. The first and third nibbles, however, each has an S-box input XOR of 0xb. Remember, we cannot predict the S-box output XOR given the input XOR with 100% certainty. However, we can use the difference distribution table to find the most probable S-box output XOR for an input XOR of 0xb. In this case, we have a very likely output XOR of 0x2 which occurs 8/16 times (50%). We pick this output XOR for both S-boxes and keep going.

Some vocabulary: an S-box with a non-zero input XOR is called an *active S-box* for that trail. In the first round of the cipher for our differential trail, the first and third S-boxes were active.

As explained earlier, the XOR of the plaintexts as they go through the permutation can be calculated by simply putting the P-box input XOR (which is now 0x2020) through the permutation, which leaves us with a value of 0x00a0 at position w_1.

We now simply continue this process until we get to position u_4 in the cipher. Why we stop there will be clear later.

Now let's talk about the probability of this full trail occurring. Remember that the only source of uncertainty comes from the S-boxes and, because of the difference distribution table, we know exactly how uncertain we are. For every active S-box, we picked the most likely output XOR. To get the probability of the trail we simply multiply the probabilities of these output XORs occurring.

This calculation only works under the assumption that these events are independent, which is not necessarily true. However, this method performs well in practice so we ignore any slight miscalculation. Just be aware that the probability we find is not the *true* probability of the trail but an approximation of it.

As you can see in the figure above, the final probability of the whole trail occurring is 0.035 = 3.5%. This is an enormous bias. Remember what this is saying: given two plaintexts which XOR to 0xb0b0, there's a 3.5% chance that their XOR at position u_4 is 0x0808. We will now exploit this bias to break 8 bits of the fifth key.

First note that the bits which we will break are the bits coming out of the last round's active S-boxes (for our trail this means bits 5–8 and 13–16).

We begin by encrypting two random plaintexts which XOR to 0xb0b0 (remember that differential cryptanalysis is a chosen plaintext attack; we can choose whichever plaintexts we want). Now what we do is attempt to *decrypt* the message to the end of the differential trail (position u_4) by *guessing all possible key bits for bits 5–8 and 13–16*. This step is called *partial decryption*. The reason we don't need to consider key bits 1–4 and 9–12 is that they will not affect the bits 5–8 and 13–16 at u_4, which will be the only ones we care about. The key bits we are breaking are called the *partial key* since they are only a part of the whole key, in our case 8 bits of it.

Now comes the core of the method. We create a table that keeps track of a count (initialized to 0) for each of the partial keys. *Each time the two partially decrypted ciphertexts XOR to the trail's final XOR (in our case 0x0808), we increment the value for the partial key which was used in the partial decryption.*

We then repeat this process for a large number of random plaintext pairs which XOR to 0xb0b0.

The partial key with the largest value in the table is most likely the correct partial key, meaning we broke 8 bits of the fifth round key!

Here's an example of this table after running differential cryptanalysis on our trail, ordered by count. The fifth key was set as 0xb520. The first and third nibbles are always set to 0 since we're not breaking them:

Partial Key Value	Count
0x0500	41
0x050f	20
0x0300	19
0x0800	16
0x050d	15
⋮	⋮
0x0104	1
0x010a	1
0x0f04	1
0x0f0a	1
0x0e07	1

You can see that the partial key 0x0500 has the highest count, which is correct since the fifth key's second nibble is 0x5 and its fourth nibble is 0x0.

You should now be left with a few questions:

1. Why does this work?
2. How do we break the other 8 bits of the fifth-round key?
3. How do we break the other round keys?
4. How many plaintext pairs is enough to break the partial key?
5. How do we find good differential trails; or, why did we pick 0xb0b0 as our input XOR?

We will answer these in order.

1. **Why does this work?** The basic intuition behind the method is this: if the partial key used for partial *decryption* was the same key used for the initial *encryption* (remember, the encryption is done with the correct keys), then we would expect the differential trail to "hold." In other words, if the partial key guess is correct we would still expect the XOR to be 0x0808 at position u_4 with a very high probability (about 3.5% in our case). The count for that partial key would then be incremented.

   If the partial key used for partial decryption *does not* match the key used for encryption, then that expectation is broken. The value at v_4 would be fairly random, meaning that when we put that value through the inverse S-box to decrypt it to u_4, we would no longer expect the value to be 0x0808 (it could still happen of course, but that would be mostly down to random chance). The count for that partial key would then *not* be incremented.

   As we repeat this for many plaintext pairs we expect that the true partial key will have the highest count because of the 3.5% bias.
2. **How do we break the other 8 bits of the fifth-round key?** Breaking the other 8 bits of the fifth key is very simple: we just need to find a differential trail with different S-boxes active in the last round. If we find a highly probable differential trail with an active first S-box, then we'll be able to break bits 1–4. Note that it's ok if that trail also has other S-boxes active for parts of the key that we've already broken. It will just break them again.
3. **How do we break the other round keys?** Breaking the other round keys is also fairly simple. We continue breaking them from the fifth key up.

   For the fourth, third, and second round key we first need to find a differential trail of appropriate length. For the fourth key, the trail stops at position u_3, for the third, at u_2, etc. You should notice that it'll be easier to find ones with a bigger bias since the trails will be traveling through fever S-boxes. We must also remember to take permutation into account. It doesn't change the process but it does affect which key bits are broken by an active S-box. For example, in our cipher, if the first S-box is active, we would actually be breaking bits 1, 5, 9, and 13.

   Another important thing to note is that to break the fourth key we need to be able to partially decrypt the ciphertext all the way to position u_3. To do that, however, we need the fifth key. In other words, we're relying on our previous key guesses to break the other round keys. If our guess at the fifth key is incorrect, so will all our other round key guesses.

   Breaking the first key is trivial with simple XOR rules. Send one plaintext through the cipher, decrypt it to position u_1 with round keys that are already broken, then simply XOR the original plaintext with this partially decrypted value to get the first round key.

4. **How many plaintext pairs is enough to break the partial key?** Differential cryptanalysis is a probabilistic attack. There's no guarantee that any of the partial keys you break will be correct. It's just highly probable that they will be. Consequently, there's generally a tradeoff between speed and accuracy. The more plaintext pairs that are put through the cipher, the higher the chance of the partial key being broken correctly.

   The reason why the aforementioned tradeoff is only *generally* true is that, as mentioned earlier, our trail probabilities are approximations. There's no way to get the trails' *true* probabilities. Therefore, no matter how many plaintext pairs are used, it's still possible to break the partial key incorrectly.

   That being said, the general rule is that the number of plaintext pairs used should be inversely proportional to the probability of the trail.
5. **How do we find good differential trails?** This is just a matter of brute force. You simply try all possible input XORs until you find a highly probable trail. On a small cipher like this, even a personal laptop finds them practically instantly. Also note that you only need to find them once, so long as the cipher design remains unchanged.

   There is a catch though. Remember that the number of bits that are broken in a partial key depends on the number of active S-boxes at the end of the trail. Remember also that we need to loop through all possible key bits for the bits we're breaking. If there are only two active S-boxes at the end of our trail, that means we need to loop through 2^8 possible partial keys for each plaintext pair. If all four S-boxes are active, we need to loop through 2^{16} possible partial keys for each plaintext pair, which is 256 times slower.

   In other words, if we want to minimize the time it takes to break the keys, we not only need to find a trail that is highly probable (so that fewer plaintext pairs will be needed), but one which has as few active S-boxes in the last round of the trail as possible.

Let's do a quick review of the steps:

- We started by making the difference distribution table for the given S-box
- We then found a highly probable differential trail ending at position u_4 in the cipher
- We found the key bits attached to the trail's last active S-boxes. These are the key bits we'll be breaking
- Then, we send in a plaintext pair which XORs to the initial XOR of the trail we chose
- We encrypt those two plaintexts (differential cryptanalysis is a chosen plaintext attack)
- We partially decrypt both ciphertexts to position u_4 by looping through all possible partial key values for the bits we're breaking
- Each time the two partially decrypted ciphertexts XOR to the trail's final XOR, we increment a value in a table for the partial key which was used in the partial decryption
- We repeat this process for many plaintext pairs
- Using a different differential trail, we break the other key bits of the fifth key
- Using shorter differential trails and the broken fifth key, we break the other round keys

As with the toy cipher, these steps have been implemented in a Python script that you are *highly* encouraged to check out. You can change the S-box, P-box, key combinations, number of chosen plaintexts, see some of the implementation details which were not discussed, and play around with the code in order to gain a deeper understanding of this section.

There are two scripts: spn-diff-crypt-simple.py and spn-diff-crypt.py. Both scripts implement the steps described in this tutorial. The difference is in how the partial keys are picked out of the table. spn-diff-crypt-simple.py just picks the partial key with the highest count in the table. spn-diff-crypt.py takes the first few (specified by MIN_OPTIONS variable) partial key candidates to account for the fact that differential cryptanalysis is probabilistic. It then combines those partial key candidates into many round key candidates. All this makes the code harder to follow, even though its core stays exactly the same.

You're encouraged to start with spn-diff-crypt-simple.py to get a basic understanding of the code and then move on to spn-diff-crypt.py if you want a slightly more realistic application.

This concludes the differential cryptanalysis tutorial. If you want more discussion of differential cryptanalysis check out Heys's "A Tutorial on Linear and Differential Cryptanalysis" [8], a more advanced and fast-paced tutorial. It could serve as a great resource for confirming your understanding as well as getting some new information, particularly in sections 4.5 and 5.

Another shorter resource is the differential cryptanalysis section in Stinson's textbook *Cryptography: Theory and Practice* [10], which is a good review of the main points.

If you're still hungry for more, move on to the next section where we discuss how modern ciphers protect themselves against differential cryptanalysis.

6 Modern Cryptography

> *I have said enough to convince you that ciphers of this nature are readily soluble, and to give you some insight into the rationale of their development. But be assured that the specimen before us appertains to the very simplest species of cryptograph.*
>
> —Poe

The Data Encryption Standard (DES) had used a key size of only 56 bits and, by the late 1990s, became vulnerable to brute-force attacks. Consequently, in September of 1997, the National Institute of Standards and Technology (NIST) sent a request to the cryptographic community to design "an unclassified, publicly disclosed encryption algorithm available royalty-free worldwide that is capable of protecting sensitive government information well into the next century" [5]. This was to be called the Advanced Encryption Standard (AES). The submitted candidate ciphers would then be thoroughly investigated by cryptographers for any possible flaws in various categories, such as security and performance, until a candidate was chosen to become the AES.

One such candidate was the cipher *Rijndael*, a portmanteau of the last names of its two developers, Belgian cryptographers Joan Daemen and Vincent Rijmen. Through a stringent process that lasted until October 2000 and was praised by many people in the cryptographic community, Rijndael was selected as the AES.

Since differential cryptanalysis was made public in the early 1990s by Eli Biham and Adi Shamir, Rijndael (from now on called AES) had to be secure against this attack. In this section we'll discuss how this was accomplished.

First let's discuss the structure of AES. We'll only be touching on the basics but all details can be found in the specification [7].

Luckily for us, AES is based on an SPN. It has a block size of 128 bits (16 bytes). The cipher key sizes and round numbers come in three varieties depending on the required level of security:

Key Length	Number of Rounds
128 bits	10
192 bits	12
256 bits	14

A *key expansion* operation is then performed on this single cipher key (of 128, 192, or 256 bits) to generate 11, 13, or 15 total round keys, each with a length of 128 bits. These round keys are called the *key schedule*.

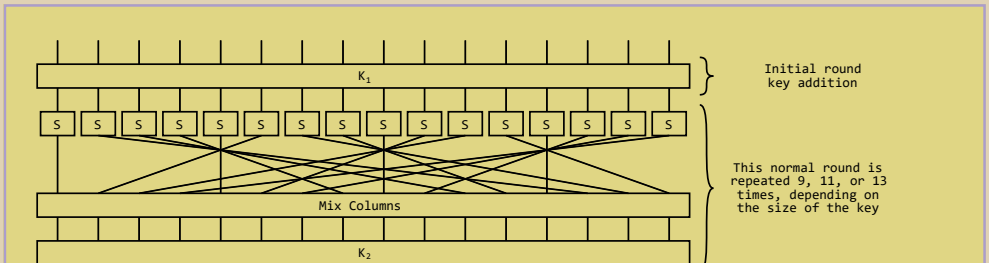
The encryption then follows a very standard SPN structure. It begins with a single key addition. Then follows 9, 11, or 13 rounds of the following four operations:

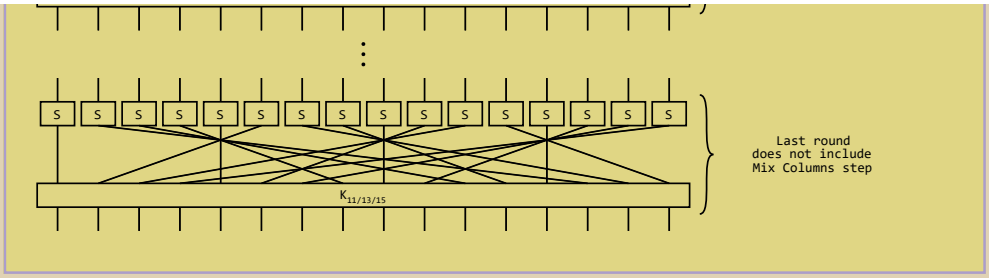
1. *SubBytes*: a substitution of each byte. AES has an 8-bit S-box (our SPN had a 4-bit S-box)
2. *ShiftRows*: a permutation of each byte. Unlike our SPN, AES's P-box works on bytes not bits
3. *MixColumns*: a second and more complicated permutation of each bit, the details of which will not be discussed here
4. *AddRoundKey*: a bitwise XOR with the next round key in the key schedule

After these rounds comes a final round (making a total of 10, 12, or 14 rounds) which is the same as the previous ones but does not perform the *MixColumns* step.

Note that the words "rows" and "columns" are used because the 16 bytes of the AES block are represented in a 4×4 array. If you want to see the exact S-box, P-box, details of the *MixColumns* step, and details of the key expansion algorithm, they can all be found within the specification in sections 5.1.1, 5.1.2, 5.1.3, and 5.2 respectively [7].

Below you can see a visual representation of the cipher. Note that because of the big block size of AES, the lines do not represent bits, as they did in the previous visualizations, but bytes.





Now that we understand AES's structure, let's understand the motivation behind the design as it pertains to differential cryptanalysis. We will only discuss this at a very high level. If you want something more detailed, you can first look at the Rijndael proposal [6], but the authoritative source in this area is the 2002 book *The Design of Rijndael* by Daemen and Rijmen [4], which dives much deeper into the mathematics.

There are two main reasons why AES is mathematically immune to differential cryptanalysis.

The first and most obvious way to protect against differential cryptanalysis is carefully choosing a differentially unbiased S-box. The smaller the values in a difference distribution table for the S-box, the smaller the trail probability will be. And the smaller the trail probability, the more chosen plaintexts (and therefore more time) will be required to get an accurate partial key guess.

The AES S-box was selected very carefully. The maximum value in the difference distribution table for the AES S-box is 4. This means that the *highest* possible probability for guessing an S-box's output XOR given its input XOR is 4/256 or 1.56%. Remember that our S-box used for the SPN had a value of 8 in the S-box, meaning the associated percentage was 8/16 or 50%. For reference, the DES S-boxes' largest value is 16 [1]. The DES S-box has a 6-bit input so the associated percentage is 16/64 or 25%. You can see that even by this very simple metric, AES is much more secure.

However, the S-box design is not the chief reason for AES's security against differential cryptanalysis. In fact, the designers make this claim in their proposal:

> In the case of suspicion of a trapdoor being built into the cipher, the current S-box might be replaced by another one. … Even an S-box that is "average" in this respect is likely to provide enough resistance against differential and linear cryptanalysis. [6]

In contrast to this, when Biham and Shamir were analyzing DES during their discovery of differential cryptanalysis, they noted that "even a minimal change of one entry in one of the DES S boxes can make DES easier to break" [1].

The reason the AES authors were so confident in their design is because the second main protection of AES against differential cryptanalysis comes not from substitution, but from its two permutation steps.

The permutation of AES was designed according to the "wide trail" strategy, first devised by Daemen in his Ph.D. thesis [3]. The basic idea is straightforward: try to activate as many S-boxes as possible by making the differential trail wide. As we saw in our SPN example, every time another S-box is activated, the probability of the trail gets smaller. AES permutation was designed in such a way that its authors were able to mathematically prove that "the number of active S-boxes in a four-round differential trail … is lower bounded by 25" [4].

In other words, a four-round differential trail is mathematically guaranteed to activate at least 25 S-boxes. Even if all of those S-boxes had the maximum bias of 1.56%, the trail's probability would already be $0.0156^{25} = 6.73 \times 10^{-46}$, a bias far too small to be exploited. And this is after only four rounds of the cipher, to say nothing of the full 10, 12, or 14 rounds. This is a very important result and one which guarantees the security of AES against differential cryptanalysis. The proof of this result can be found in *The Design of Rijndael* [4].

7 Bibliography

1. Biham, Eli and Adi Shamir. "Differential Cryptanalysis of DES-like Cryptosystems." *Journal of Cryptology* 4, no. 1 (January 1991): 3–72. https://doi.org/10.1007/BF00630563.
2. Coppersmith, Don. "The Data Encryption Standard (DES) and its strength against attacks" *IBM Journal of Research and Development* 38, no. 3 (May 1994): 243–250. https://doi.org/10.1147/rd.383.0243.
3. Daemen, Joan. "Cipher and Hash Function Design Strategies Based on Linear and Differential Cryptanalysis." PhD diss. Katholieke Universiteit Leuve, 1995.

https://cs.ru.nl/~joan/papers/JDA_Thesis_1995.pdf.
4. Daemen, Joan and Vincent Rijmen. *The Design of Rijndael: AES—the Advanced Encryption Standard*. Springer-Verlag, 2002.
5. Department of Commerce. “Announcing Request for Candidate Algorithm Nominations for the Advanced Encryption Standard.” *Federal Register* 62, no. 177 (September 1997): 48051. https://csrc.nist.gov/news/1997/requesting-candidate-algorithm-nominations-for-aes.
6. Federal Information Processing Standards. “AES Proposal: Rijndael.” (1999). https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf
7. Federal Information Processing Standards. “Specification for the Advanced Encryption Standard (AES).” (November 2001). https://csrc.nist.gov/csrc/media/publications/fips/197/final/documents/fips-197.pdf.
8. Heys, Howard M. “A Tutorial on Linear and Differential Cryptanalysis.” *Cryptologia* 26, no. 3 (July 2002): 189–221. https://doi.org/10.1080/0161-110291890885.
9. King, Jon. “Differential Cryptanalysis Tutorial.” The Amazing King. http://theamazingking.com/crypto-diff.php.
10. Stinson, Douglas R. *Cryptography: Theory and Practice*. Chapman and Hall/CRC, 2006.