Assignment - Advanced Topics in Cybersecurity (650.050)

Baptiste CALLARD

23 may 2022

Table des matières

I Introduction

In this assignment, we will focus on hash functions. First, we will try to find collisions for two different toy ciphers. Finally, we will study the length extension attack. We will also implement a variant of this attack used in message corruption in a context where two people want to exchange files in a secure way.

II Question 1

This question considers the 64-bit TC01 block cipher (the explaination of the structure is explaned in appendix 1) of HW-2 with 4 rounds and the following matrix used in the MixColumn operation :

$$M = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

We keep the key set to 0 (i.e. all 64 bits are zero). Instead we add 4 keys of 64 bits which are randomly drawn. The values are drawn once at random but then remain the same for all encryptions. This means that there is no key scheduling. The 4 values are then used in each of the rounds respectively.

In the next two sections we will use this function as the basis for our compression function. A first observation is that the matrix M is not invertible. Indeed, this matrix is not of full rank. This means that the linear application associated with the matrix M is not injective. Here, this property could be useful for the compression function because once we have the hash, the attacker should not have a simple way to invert the MixColumn.

II.1 Part 1

Presentation of the reduction function

We define the compression function F : $\{0,1\}^{64} \longrightarrow \{0,1\}^{32}$ which is constructed from G(m) = E(m$\oplus$IV) by extracting the first 32 bits of the result from G.

In this section, we consider 64-bit messages. There will be no question of padding. Thus, we will have a 64-bit word. We can divide this word into 16 nibbles of 4 bits in the AES. I have chosen the following representation :

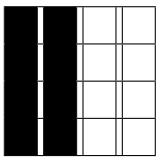
$$m = m_0||m_1||m_2||m_3||m_4||m_5||m_6||m_7||m_8||m_9||m_{10}||m_{11}||m_{12}||m_{13}||m_{14}||m_{15}$$

with $|m_0| = ... = |m_{15}| = 4$

To represent the message in matrix form I have chosen the following convention :

$$\begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix}$$

So, when we keep the first 32 bits we will talk about the following bits which are represented in black :



We will try to show that this function is not a good reduction function to produce a hash because it is possible to find a collision.

Again, when we reduce the space from 64 to 32 bits, we get collisions even if the function is perfect. A function is said to be collision resistant if we approach the birthday paradox.

We will first define the following concepts :

Def : **Compression function**

It is a function that operates on data of fixed sizes $|m| = p \in \{64, 128, ...\}$. This function reduces/compresses the size of the input data.

$$f : \left| \begin{array}{ccc} \{0,1\}^p & \longrightarrow & \{0,1\}^n \\ m & \longmapsto & m' \end{array} \right. \text{ with p>n}$$

Thus, the function we are considering is indeed a compression function.

Def : **Collision**

Let f : $\{0,1\}^p \longrightarrow \{0,1\}^n$ *with* $p > n$
A collision for f is a pair x_1,$x_2 \in \{0,1\}^p$ such that

$$f(x_1) = f(x_1) \; and \; x_1 \neq x_2$$

I will use a differential attack to find this collision. To do this we will first study the weaknesses of the Mixcolumn matrix M.

$$M = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

We can see that on the 4th column we have the identity. So the differences on the nibbles that are on the 4th line of our input message m will not be affected by the Mixcolumn. This is a big weakness. Indeed, if we have a difference on this 4th line then only the Shift Row will have an impact. However, we can follow this difference because it will always remain on this same 4th line (will just rotated). Therefore, the 4th line will always remain independent of the Mixcolumn.

Let m be the message for which we want to find a collision. We consider the message :

$$m = m_0||m_1||m_2||m_3||m_4||m_5||m_6||m_7||m_8||m_9||m_{10}||m_{11}||m_{12}||m_{13}||m_{14}||m_{15}$$

Then, one can define $\hat{m}$ as the following

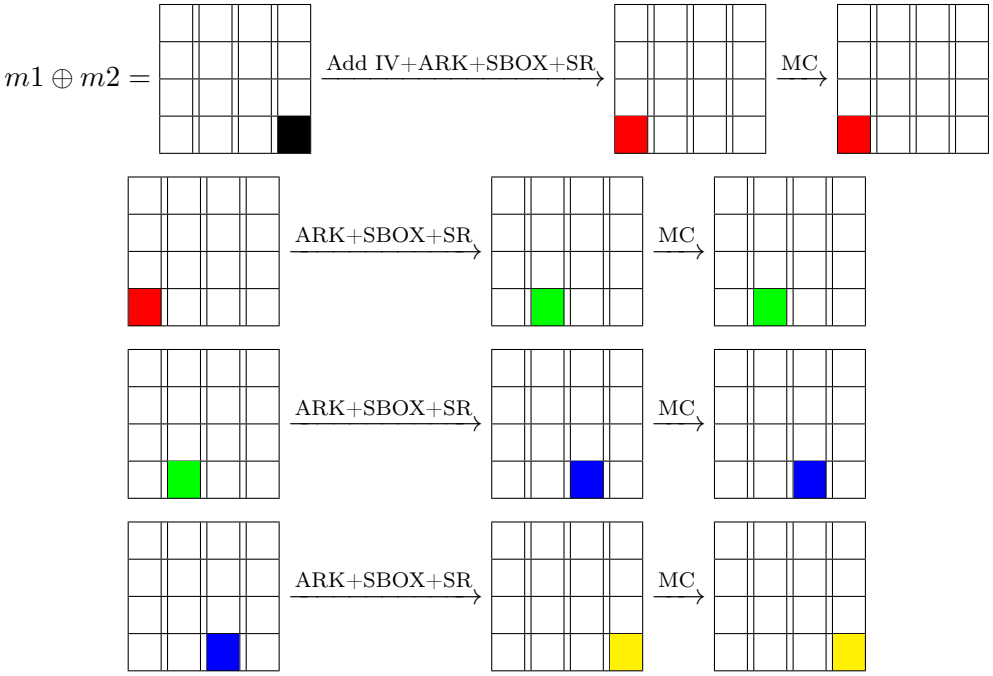
$$\hat{m} = m_0||m_1||m_2||m_3||m_4||m_5||m_6||m_7||m_8||m_9||m_{10}||m_{11}||m_{12}||m_{13}||m_{14}||m_{15} \oplus \Delta \; with \; \Delta \neq 0$$

and we will show that this message $\hat{m}$ allows us to find a collision.

$$\hat{m} \oplus m = 0||0||0||0||0||0||0||0||0||0||0||0||0||0||0||0||\Delta = \begin{array}{|c|c|c|c|}\hline 0&0&0&0\\\hline 0&0&0&0\\\hline 0&0&0&0\\\hline 0&0&0&\Delta\\\hline\end{array}$$

with $\Delta \neq 0$

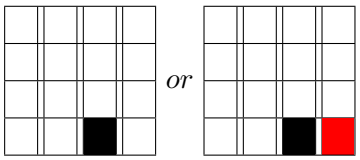
So I chose to study the path below. The nibbles that are coloured have a non-zero difference. The colours change when the differences change. Nothing is imposed on the differences except that they are strictly non-zero at the beginning (black nibble). The difference only changes after the sub-nibble because all other operations are linear and therefore invariant to the difference.



The difference changes after the SubNibble. Furthermore, it is possible to control the difference position throughout TC01. So, after 4 rounds, if we extract the first 32 bits then we are sure to have a zero difference. This means that we will have the same hash for two messages that are different. So, we have found a collision for this cipher. Indeed, if the Xored between two elements is zero then they are necessarily identical. Thus we have :

$$m1 \oplus m2 = \begin{array}{|c|c|c|c|}\hline &&&\\\hline &&&\\\hline &&&\\\hline &&&\blacksquare\\\hline\end{array} \Longrightarrow m1 \neq m2 \;\; and \;\; F(m1) = F(m2)$$

This method also works if you place the differences like this :



Indeed, we want the difference to be on the 4th row but also on the last 2 columns that we do not keep. Thus, there are many possibilities to create a collision.

We can also note that the random value of the randomly drawn keys and the value of IV have no impact in this attack and can therefore take any value. So, as we have seen many times, making a block cipher more complex does not always make things better (here random keys are a good example).

II.2 Part 2

We saw in the previous question that when we apply TC01 then the function that maps a difference in input to a difference in output is the identity to a constant. Indeed, the difference returns to the starting point but may have changed. We are not interested in the value but in a function that takes as input a difference.

$$m_1 \oplus m_2 = \Delta m \longrightarrow \Delta m \oplus \nabla = G(m_1, IV) \oplus G(m_2, IV)$$

.
Now we will iterate the process so that we can find the message hash of multiple size of 64bits by using the **Part 1**.

The process is as follows :

F : $\{0,1\}^{64} \longrightarrow \{0,1\}^{32}$ which constructs from G(m,IV) = E(m$\oplus$IV)$\oplus$m keeping the first 32 bits. For more than one message block, the hash can be obtained by calculating G(m2, h), where h = G(m1, IV) and m1, m2 are the two message blocks, then applying the compression function (extration of 32 first bits).

Presentation of the collision

We also use a differential attack that uses the result of the previous question. For this attack we consider that we want to find a collision for a message m of size multiple of 64.

$$m = m^1||m^2||...||m^n$$

with

$$|m^1| = |m^2| = ... = |m^n| = 64$$

Similarly, the following notation should be used :

$$m^i = m_0^i||m_1^i||m_2^i||m_3^i||m_4^i||m_5^i||m_6^i||m_7^i||m_8^i||m_9^i||m_{10}^i||m_{11}^i||m_{12}^i||m_{13}^i||m_{14}^i||m_{15}^i$$

For our attack, we will show that the word

$\hat{m} = \hat{m}^1||\hat{m}^2||...||\hat{m}^n$

defined as

$\forall i \in \{1, ..., n\}$

$\hat{m}^i = m_0^i||m_1^i||m_2^i||m_3^i||m_4^i||m_5^i||m_6^i||m_7^i||m_8^i||m_9^i||m_{10}^i||m_{11}^i||m_{12}^i||m_{13}^i||m_{14}^i||(m_{15}^i \oplus \Delta_i)$ *with* $\Delta_i \neq 0$

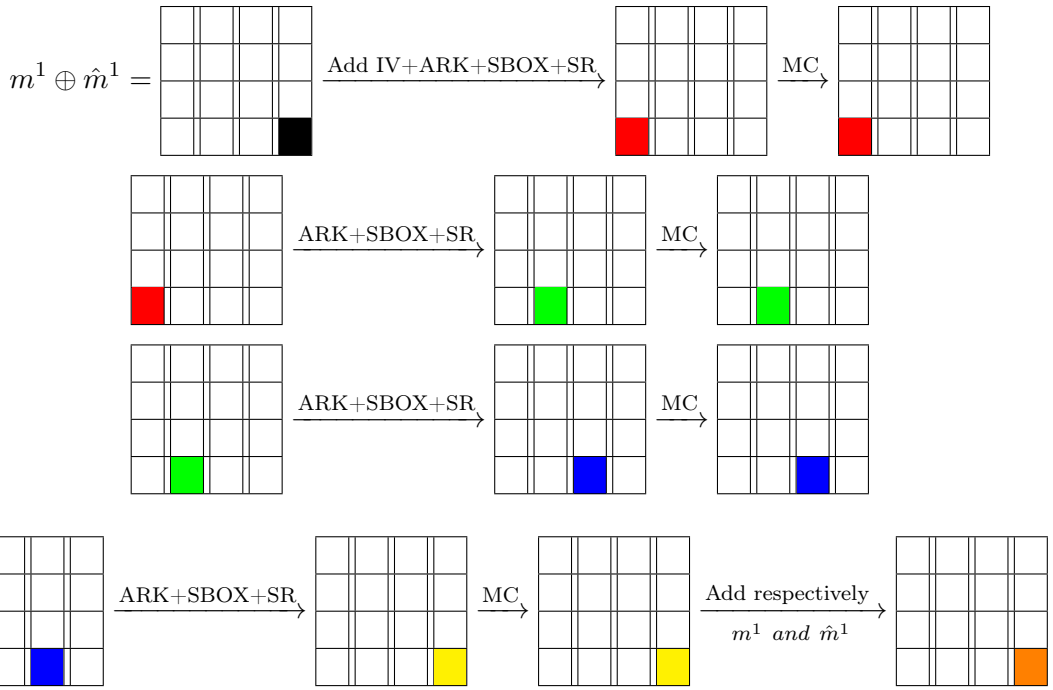
can be used to find a collision for message m.We will use the following observation from the previous question :

$$m^1 \oplus \hat{m}^1 = \Delta m \longrightarrow \Delta m \oplus \nabla = G(m^1, IV) \oplus G(m_2, IV)$$

On the other hand, now we also add m^1 to $G(m^1, IV)$ and $\hat{m}^1$ to $G(\hat{m}^1, IV)$ so in this new toy cipher we have :

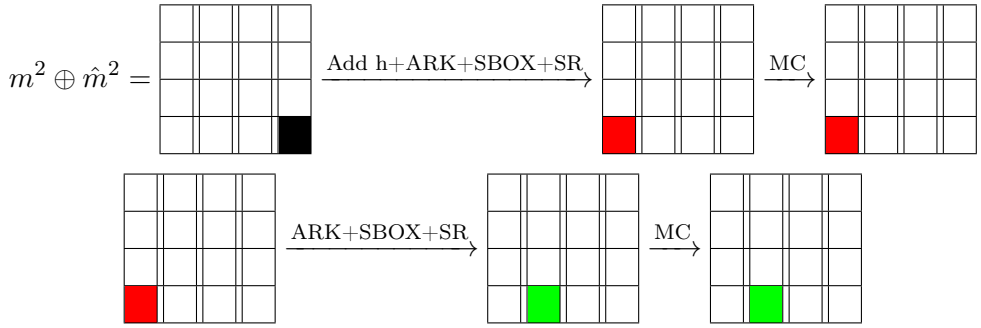
$$m^1 \oplus \hat{m}^1 = \Delta m \longrightarrow G(m^1, IV) \oplus m^1 \oplus G(\hat{m}^1, IV) \oplus \hat{m}^1 = G(m^1, IV) \oplus G(\hat{m}^1, IV) \oplus \Delta_m$$

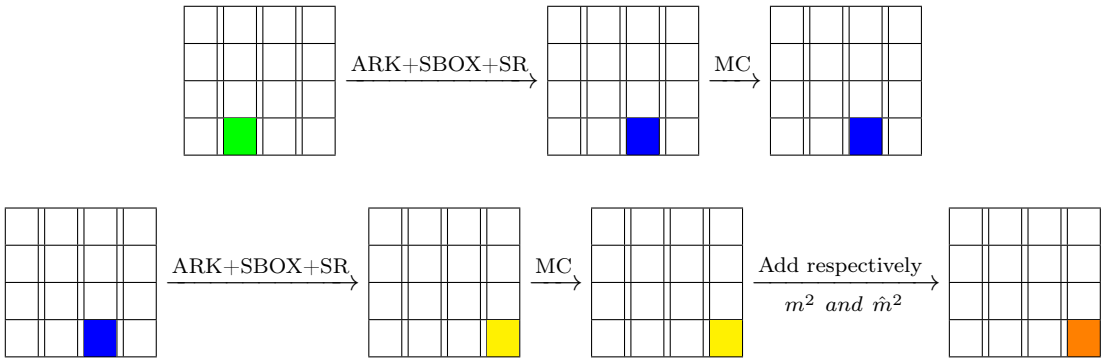
So we start by XORing m^1 and $\hat{m}^1$ with $\Delta_1 \neq 0$ on the last nibble (in black).



Despite this modification, we find the hash for an intermediate state. However, this is an internal state that we do not have access to. We want to have the same hash after hashing the whole message. So we need to continue the process.

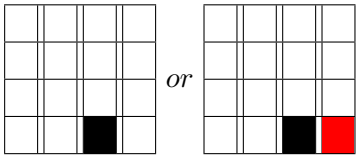
We now replace IV by h. For both messages this value is identical by construction. Thus, we find ourselves exactly in the previous configuration except for this change. The colours are the same as before, but there is no particular reason for the differences to be identical. It just shows where the differences could be different.





Thus, we can see that it is possible to iterate this process. Thus, by construction of m^i for $i \in$ $, ..., n\}$ and by immediate recurrence we have that $\mathrm{F}(m) = \mathrm{F}(\hat{m})$ and $m \neq \hat{m}$. Thus, we have produced a collision for message m with $\hat{m}$.

Again, this method also works if the intermediate messages have differences like this :



III Length extension attack against MD5

The length extension attack is an attack that calculates a valid hash from an unknown message that is modified. It is a method that is very powerful. This attack works very well against hash functions that use the Merkle Damagard structure.

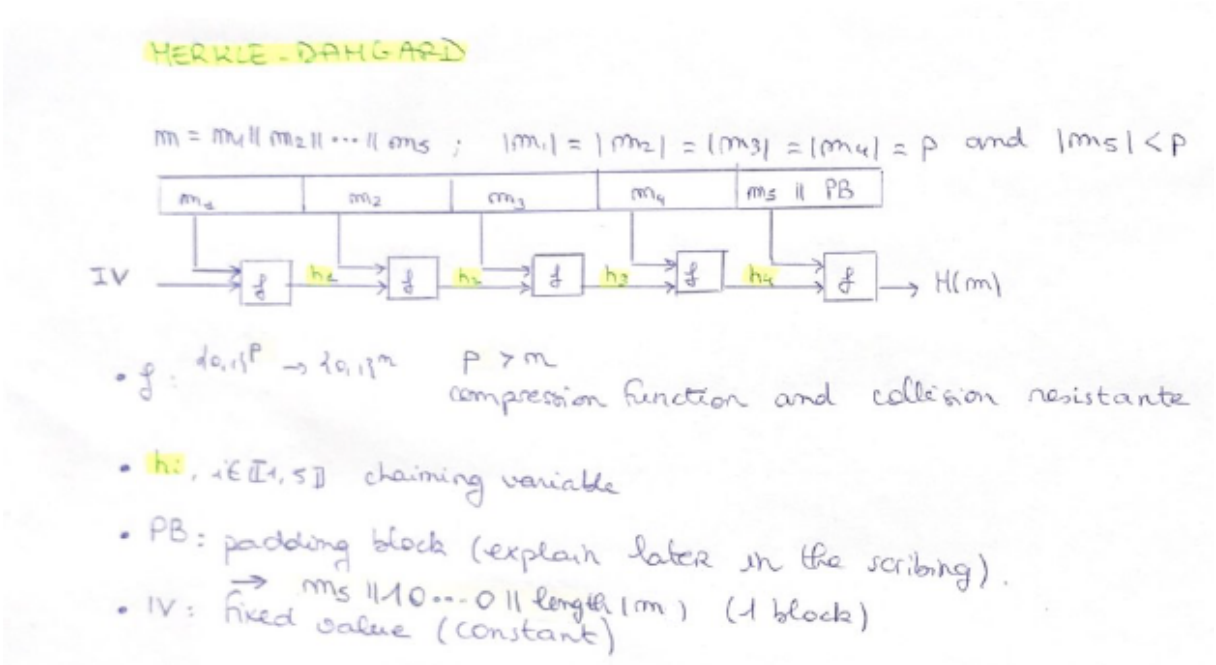


FIGURE 1 – Merkle Damgard Structure

There are many length extension attacks. Indeed, it depends on the configuration in which we place ourselves. What data is accessible to the attacker etc... Firstly, we will use the MD5 padding which is the following :

The padding consists in completing the string until its length is congruent to 56 bytes (mod 64). The remaining 8 bytes are kept for the size of the encoded length field.

So for my attack I chose the following configuration :

There are three actors, an operator, a server and an attacker.

The operator wants to send a message to a server and he wants the server to be able to verify the authenticity of the message. The operator therefore also sends the hash of the message so that the server can compare and is sure to receive the authentic message. To do this, the server will calculate the hash of the message and if the hash is authentic then the server can assume that the message is also authentic.

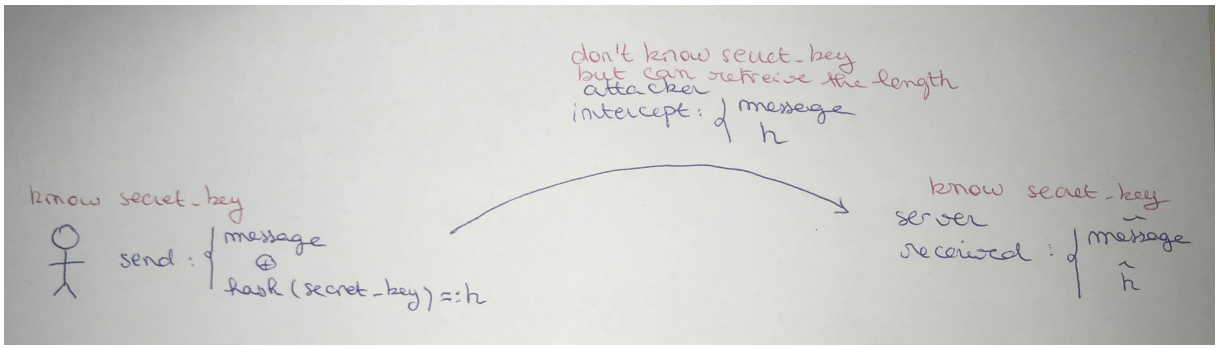


FIGURE 2 – Scheme length extension attack

To make it more secure. The operator and the server decide to share a key that will be secret and only shared by them. The idea is to place the key before the message and then calculate the hash. The operator figures that by doing this even if his hash and message are intercepted then a malicious person will not be able to send a random message with the corresponding hash. Because when the server will add the key then it can detect that it is not a message from the operator.

Now, let's assume that the attacker has access to the sent message and the hash without knowing the secret key. We can have two configurations. The attacker knows the length of the key or not. But we will see when in a simple case (negligible length then we can brute force if the server allows it [1]).

Summary of the situation

— operator : sends a message m and the H($secret\_key||message$) = h

— attacker : receives the message and h. He must also know the length of $secret\_key$ or have a way to find it. We will see that it is possible to brute force the length of the key.

1. One could imagine that the server blocks a user with too many attempts but this is outside the scope of the course and we will assume that this is not the case or that the attacker can use other tools

— server : receives a message and hash then calculates H($secret\_key||message$) and compares with the hash

Description of the attck

Once the user sends the request, a hash is calculated with a secret key. If the user sends the message : $message\_sent$ then the following hash is calculated H($secret\_key||message\_sent$). When the message $message\_sent$ is received, the server will also calculate the hash of the massage by adding the $secret\_key$ at the begging. If the hash is valid then the request will be investigated. However the attack can easily generate a valid hash without knowing $secret\_key$. The attacker can recover the message and the hash of the message between the operator and the server.

The attacker can then generate a valid hash for :

$secret\_key||message\_sent||padding||attacker\_malicious\_data$. He just had to pick up the hash algorithm where he stopped and continue the hash. To do this, he still needs to know how to do the padding correctly. Indeed, the attacker have to do the padding manually and then put his message. In my configuration the attacker does not know the key lenght and it is necessary to do the padding in MD5. To overcome that, the attacker can always sent the same forging hash h* and he can try to send messages with different padding sizes. By trying all key sizes, it is possible to create a valid message for the hash h' when it is the right one.

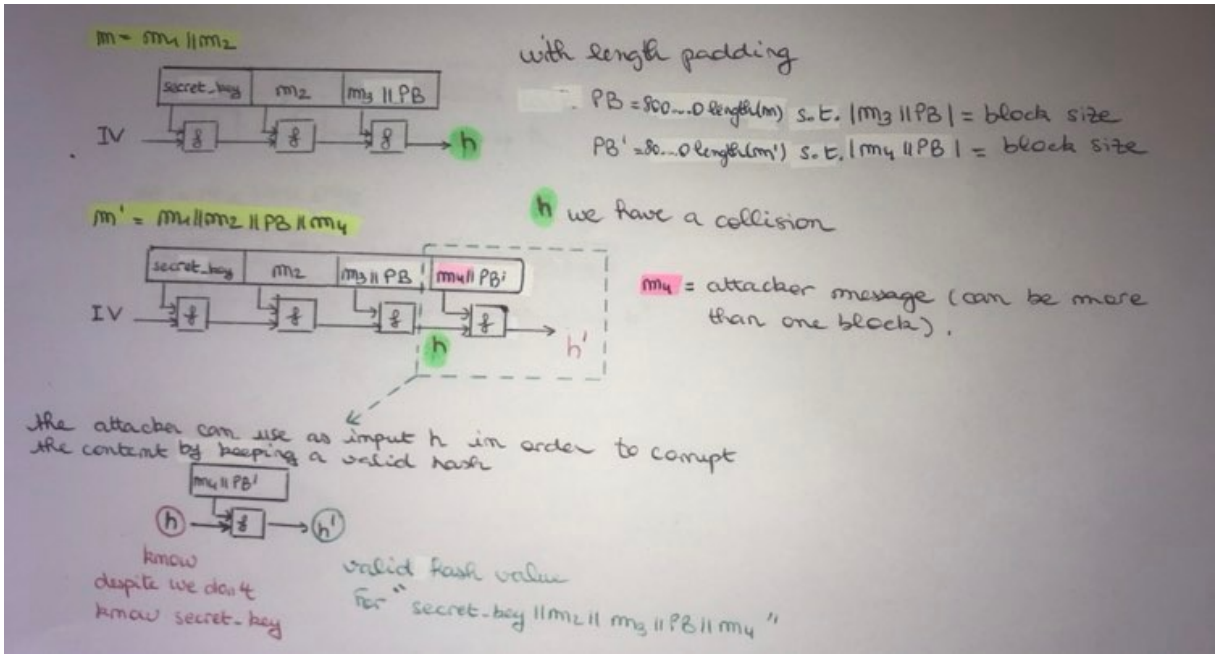


FIGURE 3 – Length extension attack in pratice

It is then possible to enter the received hash into the state of the hash function, and pick up where the original request ended. Thus, the attacker can give the hash function h as state instead of IV. The second part of the message could then be fraudulent or contain false information. The attacker can send the new message with a new hash value which will be accepted by the server. The attacker will have taken care to add the padding to the new message by hand. Here the attacker will send the message : $m_1||m_2||padding||m_4$ and h' valid for $secret\_key||m_1||m_2||padding||m_4$.

So when the server receives h' and $m_1||m_2||padding||m_4$ it will not know that the message is forged.

III.1 Result

So I managed to implement this technique using the openssl library. I have implemented a version where the attacker does not know the length of the key. So, I try to send the message with the corresponding padding at all possible sizes until the server accepts my message. Naturally, I start my length search from 0 and increase because more people have a small password than a long one.

```
############ START #############

the message sent is : Hello, I would like to order two pizzas and three drinks
the secret key is : HardS_cret@!-2
the attacker's message is : Could you please send them to this new address - 21 rue jean paul - ? I made a mistake in my last message!
the hash sent with the message is  :
2a4be48886aca72f50c2357e364523e9
-----------------------------------

We are trying to send a fraudulent message by testing several key sizes with the forging hash

The following hash is valid :
1495aeff38c07d9d2d944027c1dbd59

The length of the key is 14

-----------------------------------
We can also check that the initial message with the hash is accepted by the server
The following hash is valid :
2a4be48886aca72f50c2357e364523e9

############ END #############
```

FIGURE 4 – Results length extension attack

The attack works for messages and a key regardless of their length. They can be on several blocks. The length of the recovered key is displayed when the server accepts the request.

In the case shown here, thanks to this attack, the attacker was able to have the order delivered to his home by adding his modification. If the person receiving the message is not suspicious, it can work.

Looking at this attack, we notice that it is very complicated to counter it in this configuration. Indeed, an attacker can reconstruct a message by padding it by hand no matter what form the padding takes. Thus, one can wonder if by keeping the Merkle Damgard structure and more particularly MD5 there is a way to counter this attack.

Method to avoid this attack

One method that is very well known because it is relatively common on the internet is to use a method called HMAC (Hash-based Message Authentication Code). This is an authentication method that is specifically designed to avoid length extension attacks.

This method consists of using the key twice. Thus, in the previous configuration where we have an operator and a server sharing a key, the information is exchanged as follows :

$h := HMAC_K(m) = MD5((K' \oplus opad)||MD5((K' \oplus ipad)||m))$

$$K' := \begin{cases} H(K) & K \text{is larger than block size} \\ K & otherwise \end{cases}$$

with

- K is the $secret\_key$
- m is the message sent by the operator
- ipad = 0x363636...3636 with |ipad| = 512
- opad = 0x5c5c5c...5c5c with |opad| = 512

The ipad and opad values are not critical for the security of the algorithm. They have been chosen to have a large Hamming distance. The HMAC security reduction requires them to be at least one bit different.

This method is not a new block cipher but an additional method to exchange data. This method provides better immunity against length extension attacks.

Indeed, the external use of the hash function prevents the attacker from having access to the intermediate result of the hash internally.

The Keccak hash function, which was selected by NIST as the winner of the SHA-3 competition, does not need this nested approach and can be used to generate a MAC by simply adding the key to the message. It is not sensitive to the length extension attack.

IV Conclusion

Thus, we have seen here the power of differential attack. At the begging, this modified version of TC01 seems to be complicated enough that a naive guess cannot find a collision. However, with an analysis of the MixColumn matrix M one can simply exploit the weakness of the 4th column. This attack also proved to be usable on the second version which takes messages from several 64-bit blocks.

In a second step, we implemented a key extension attack. If you look at the attack, there is nothing complicated about it (when it works). On the other hand, it is very powerful and can be used in a multitude of configurations. I find this attack very nice because of its simplicity, its efficiency and because we can see directly its application. The methods used to avoid this kind of attack is either not to use the Merkle Damgard structure or to use more sophisticated methods like HMAC to park the same hash function but prevent this kind of attack.

V Appendix

V.1 Appendix 1 - TC01 (from my second assignement)

In this question we consider a Toy cipher named TC01. In each round we apply the following steps :

1. Add round key (ARK) : This function XORs 32 least significant bits of the round key to the least significant bits of the cipher state

2. Subcell (SC) : This function maps each 4-bit word of the cipher state according to the following Sbox
3. Shift Row (SR) : This function works exactly like in AES and rotates the nibbles in the cipher state by 0, 1, 2 and 3 places to the left.
4. MixColumn (MC) : This function multiplies each column of the cipher state with the following matrix (defined over $\mathbb{F}_{2^4}$)

The S-box is the following :

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

and MixColumn matrix is as follows :

$$M = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

The inverse matrix of MixColumns matrix is (I compute it with the package galois in Python) :

$$M^{-1} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

Finaly, the key scheduling algorithm starts with the 64-bit master key K = k_0 and for each round r $\geq$ 1 the round-key is determined as following. Let $k_r = (k_{r1} \oplus$ 0xf33f) »16 The the roundkey RKr = k_r&0x00000000ffffffff. The round index starts with 0, i.e. index of the first round is r = 0..