Applied Information Security and Cryptography

Academic Year 2019/20

Group 7:
Arianna Ravera s277752
Chiara Lanza s277750

November 27, 2020

Contents

1 Lab01: Breaking classical encryption schemes

Three ciphertexts are obtained using classical encryption schemes with Python, the goal is to decrypt them.

1.1 First ciphertext

The first plaintext has been obtained cracking a **monoalphabetic cipher**, from the beginning it is excluded a brute force attack since there are 26! possible mappings.
The technique adopted exploits the analysis of the frequencies of monograms, digrams and trigrams, comparing the results of the ciphertext with those of the English language. The most frequent letters are compared with those of the English language in table 1.

Most frequent monograms in the text:	X	T	B
Most frequent monograms in English language:	e	t	a
Most frequent digrams in the text:	BY	YX	YT
Most frequent digrams in English language:	th	he	in
Most frequent trigrams in the text:	BYX	ATV	BYT
Most frequent trigrams in English language:	the	and	ing

Table 1: Results table

It is possible to see that in different analysis they contradict each other, so letters can't be easily replaced.

In a more accurate study of the results we evaluated the repetitiveness of letters among them and the only correspondence always confirmed is **e-X** (lowercase letter for the plaintext and uppercase one for the ciphertext) while **t-B** and **h-Y** are repeated several times.

The next step was based on the analysis of the digrams, trigrams and quadgrams formed by the letters that has been already 'translated' and the most frequent strings of the English language.
A practical example is **BYT** which is translated as **th-**, since the third most frequent trigram is **tha** we replaced **T** with **a**, and so on for all the other strings obtained previously.

After these operations the replaced letters are: **a-T d-F e-X f-L h-Y i-N l-O n-J o-R r-W t-B w-A y-H**.
The remained plaintext letters unknown (**b c g j k m p q s u v x z**) has been found with a trial & error approach until the text was comprehensible and correct.

It's easy to see that frequency tables are not enough for an automated function because at least one English dictionary would be needed to evaluate the

correctness of translations.
With the addition of this information you could develop a decryption algorithm whose solution will be found when all the letters present in the text will be part of one and only one translatable word with a complete meaning in the language of belonging.

1.2 Second ciphertext

The second ciphertext has been obtained cracking a **Vigenere cipher** with a unknown key of an unknown length which is continually repeated. The longer the key will be the more the number of cycles to run will increase and therefore the number of monoalphabetic ciphers to find for each different letter in the key, with a maximum of 26 possible alphabets for 26 possible letters.

There are sequence of repeating shifts and the brute-force way to break it without statistical analysis depends on the key space, which is $O(26^k)$ for a key of length k.

If we try to break this algorithm by hand, using only tools that estimate the frequencies of ciphertext letters, as in the case of monoalphabetic cipher, it would certainly be more complex because the relative frequencies would be much lower and the comparison with digrams and trigrams would become useless. The letters decoded with the same cipher cannot be consecutive between them except in the case of key length equal to 1.
The key we found is long 19 letters and is '*qwertyuiopasdfghjkl*'.

1.3 Third ciphertext

In the third ciphertext, evaluating the crypto-frequencies of the whole text, it has been noticed that they do not follow a coherent distribution regarding that of the letters of the English alphabet but, on the contrary, the crypto-frequencies of the subsequences through the calculation of the parameter $S=\sum_{i=0}^{25} Q[i]^2$ give a correct result. It's noticed that every 8 characters the frequency will be about 0.065 so the key will have this length and we can establish with certainty that the text has been encrypted using **Vigenere**. In fact in this way we found the key able to decrypt which is '*password*'.

1.4 Stream cipher

The keystream of a stream cipher should approximate the properties of a true random number stream so bits should be uniformly distributed and unpredictable when observing previous bits. Of course if we use the same key for multiple encryptions the concept of stream cipher falls and it is easy to attack it with '*Vigenere cracking algorithm*'

2 Lab02: A simplified version of AES

2.1 Avalanche effect on different versions of simplified AES

In order to study the avalanche effect, several versions of simplified AES has been tested using different number of rounds and different experiments (changing a single bit in the plaintext or changing a single bit in the key). The **Table 2** shows the results.

	2 rounds	3 rounds	4 rounds	Lazy 4 rounds	VL 4 rounds
Fixed key rand plaintext	4.204	8.604	7.889	2.164	2.215
Fixed plaintext random key	8.447	8.652	7.772	8.076	2.350

Table 2: Avalanche effect

On a simplified AES with two rounds we saw that changing one bit of the plaintext the avalanche effect is not so evident, on the other hand, changing one bit of the key is more efficient. Since there are few permutations the diffusion property is not so evident and the key has an important role.
When we applied the AES with three and four rounds the avalanche effect is finally visible.

The problems are much more feasible with the lazy versions where there is no permutation at all. The shiftRows and MixColumns functions are never used, only the sBoxs and the addKey ones work.

Of course there is more avalanche effect when we change a bit in the key, since the sBoxs never change and addKey is a XOR operation with the expanded keys. So if we change the plaintext leaving the key fixed, the loss of permutations is fatal for the confusion and diffusion properties desired.

The last experiments with the very lazy algorithm are the worst because we are using always the same two keys of the expanded ones, so the addKey function is done with the same input on different states.

2.2 Improperly implemented block cipher

We had a plaintext and the corresponding ciphertext, the goal was to decrypt another ciphertext that use the same key exploiting them and the sBoxs (which are never a secret, neither in the full AES).

When we have a plaintext and its ciphertext a KPA attack it is easy to compute. In order to find the key it is possible exploit them in this way:
the first two steps are done on the plaintext and they are the same as the encryption, so substitution through sBoxs and shifRow. We have to remember that the addKey function is just the XOR operation between the first two keys of the expanded one and the state, and the XOR operation has the commutative property. Therefore if this operation is computed between the state computed

before and the known ciphertext, since it's the last step of the encryption, the result will be the combination of the two expanded key that we were looking for.

Then, having the key, it is possible to divide the binary ciphertext into blocks of 16 bits and process them in accordance with what has been do in their encryption, so reverse the order of the first 8 bits with the second 8, and then use the decrypt_foo method that will return the result. Once this is done, we turn the result into characters to have this text:

The studio was filled with the rich odour of roses, and when the light summer wind stirred amidst the trees of the garden, there came through the open door the heavy scent of the lilac, or the more delicate perfume of the pink-flowering thorn

The main weakness of this encryption methods is that AddKey is performed only at the end and there is no round computed. In this way the algorithm results much easy to crack as we proved.

In the case in which the plaintext is unknown, the ciphertext in the file would be decrypted with cryptanalysis tools. Being the operand mode of the block cipher the ECB, it is possible to forge it, as in the case of stream cipher, encrypting multiple messages with the same key.

3 Lab03: Public-key encryption

3.1 RSA encryption

Public key encryption is based on the generation of two different keys for each part: a public key which is shared and a private one kept in secret.

For public-key encryption we will use the RSA algorithm. Both encryption and decryption are based on modular exponentiation. Our key pair will be 1024 bits long, so first of all it requires two prime number large enough for the keys length desired, in our case at least 512 bits.

The chosen random values for the keys generation should be uniformly distributed, but as we know common libraries are not sufficiently unpredictable for cryptographic applications, so a function is built for this purpose. Those numbers require to be prime so we reproduce the Miller-Rabin test, which is a probabilistic test that runs in polynomial time and exploits the Fermat's theorem. Since it is probabilistic and not deterministic, it needs enough iterations. At the end the found numbers are:

p=114464804031370118593974323147511604360630957353745532050990
39541370419369139702100685174527914454230203946527213035093562905
61478921539731731997416815357 3

q=108778484144425606576955363207299060152068022956493800214814
22871894077686171187025325350554137035690296044695620985737586249 4
77791544907783535521928356273

In order to generate the public key we look for a number which is relatively prime with the totient of the previous prime numbers, so a recursive Euclidean algorithm is implemented to test if the chosen number is acceptable. This function will also return the d value which will be our private key.
The RSA key pair values *(N,e)* and *(d,e)* generated are:

N=124513078704211786974956933795104867207095934442496736329 70
90355373401338106415583437047029841696008482180529360270083358070
32476961957328286958128049561025593457159746704296663275345788984
56564996490188328260309416375907054822319784137727538806205663641
36021917412647864707838428384332748319957 6859321913429

e=4991

d=41013724982914080902991224035093648905723445446496620822689 17
13931851693016819719328893070939641001391445560061767985782542098 5
61661907128238347350257419934631230203411670297763142119307187697 9
37948178106131787193999223612799278713984173671478138906954186856 3
76782809414368983022749311829056732597101067047 67

3.2 Complexity of RSA and Diffie-Hellman

Diffie-Hellman key exchange enable two users to securely exchange a key that can then be used for subsequent symmetric encryption of messages because its security is based on the difficulty of computing discrete logarithms. Public keys are computed from common parameter for both parties and the key that will be used for the specific communication will be the same but nobody else can understand.

In **Table 3** we reported the average time taken by the script to generate a valid RSA key pair and to generate a valid Diffie-Hellman key pair and shared key.

RSA key pair	**DH key pair**	**DH shared key**
0.021117 s	0.025460 s	0.035990 s

Table 3: Generating time

Instead, the average times required by RSA and AES encryption and decryption algorithms are in **Table 4**.

RSA		**AES**	
encryption	**decryption**	**encryption**	**decryption**
0.009561 s	0.475474 s	0.000041 s	0.000001 s

Table 4: Time

AES using CTR mode result the faster because the bit-wise operations are inherently simpler than combining exponentiation and modular arithmetic as in the case of RSA.
But if we take into account even the time for generating the keys, since in this AES we use Diffie-Hellman, in **Table 3** we saw that it takes longer than double for generate its key and the shared one, without taking into account that the public key requires some time to be shared.

In the case in which in an encrypted conversation one of the parties always responds with either Yes or No:

- using RSA this response will be always equal because we are using the same N, e and d values
- using AES the answers will change continuously thanks to the randomness of their generation

In this case it is not obvious that the communication will be always confidential. In the case of AES the confidentiality is guarantee, while considering RSA, if we know that the possible messages are Yes or No, the confidentiality of the communication is no more ensure. It will be easy to understand which

one of the two answers correspond to the Yes and which one to the No and so arrive to decrypt them.

The only possible way to guarantee the confidentiality also for RSA is making it probabilistic.

If we want decrypt a message sent to the instructor by another group we can exploit the instructor as padding oracle because the first byte will be always set to zero for our function. So if we set a second ciphertext as the first one multiplied by a value x modulo N, we can try all the 256 possible values of x since a valid message is found.

4 Lab04: Hash functions and digital signatures

4.1 Hash functions

We will analyze two different types of hash functions, the well known SHA-256 and the Universal hash function (which has the peculiarity that its output is uniformly distributed over the possible 2^n values), in order to verify that the complexity of finding random collisions in their output is proportional to $2^{n/2}$.

In the **Table 5** are shown the average values of C1 and C2, which are respectively the repetitions of the number of times that the two while loops that look for a collision are executed in each function. The values expected are in the order of 16 for a hash digest of 1 byte (so for 8 bit: $2^{8/2}$), of 256 for 2 bytes, of 4.096 for 3 bytes and in the order of 65.536 for 4 bytes' hash digest. It is possible to see that values of SHA-256 and Universal are quite similar although in a couple of cases, such as in the C1 for 2 bytes, the result suffers a peak in the case of the Universal function.

	SHA-256		Universal	
	C1	C2	C1	C2
1 byte	10.4	9.9	8.7	6.4
2 bytes	377.0	132.5	946.0	113.0
3 bytes	5106.2	3568.1	6853.7	1224.8
4 bytes	36464.9	19624.6	75916.4	35462.2

Table 5: Comparison between SHA-256 and Universal

A cryptographic hash function always gives outputs uniformly distributed over the possible values, in order to guarantee the best randomness of the output; this doesn't means that if a hash function's output is uniformly distributed it is a cryptographic hash function. It is possible to think to some non-secure hash functions like the Bit-by-bit XOR with rotation: this operation guarantees a randomly distributed output, but can be easily reversible. It has to be infeasible to generate a message that yields a given hash value and also to find two different messages with the same hash value.

The Universal hash function is built so that its output is uniformly distributed over the possible 2^n values but it is easy to find a collision also for a large number of bytes. This means that it is not a cryptographic hash function by definition.
We find a collision for the message *b"SHA-256 is a cryptographic hash function"* given by:
`m2=609315216172957437010334111094774013624185439035613951731642372713743307915315085955355277823880`

The situation changes for SHA-256: since it is a cryptographic hash function it would result unfeasible to find a collision for 32-bytes digest because the number of steps to execute would be in the order of $O(2^{256/2})$.

4.2 Digital signatures

We implement a Schnorr signature scheme based on the previous hash.
The public key generated is:

```
pk =53090624274644489534290341757334670483795321506522535790073985695899651889443723806736334504225582131413369338585558121947290436315956266321274218141729607771374505179410680124489310180712329176446933016851188196480929708498258901647585684938786978712977155801333846972093108212659637421449015888193553335 01
```

For the message "*Signed message, verify it with the public key*", encoded as a bytes-like object, the result of the signature algorithm is:

```
r = 265342832893082732560463209681586644653592791248
s = 388271925941366607193369294538154854614678330516
```

In order to derive the instructor's private key from the two signed messages we have simply apply some mathematical passages: putting the two messages and signatures in an equation it is easy to find x and k such than the k is incremented with a number from 1 to 100 in a while loop until it works.
The result obtained is for a *k1=k+70*, so for the 70^{th} iteration, and the private key found is:

```
sk = 175101539802224303086090570571484452964925793718342847690672091104687753111297896553288143612471153181184281603547639952029811926994470540338 58
```

In an implementation of the digital signature algorithm in which the value k is obtained by increasing a counter and the encryption is done with AES whose key is derived from the private key, is a secure implementation. In fact using AES the encryption is trivial to broke if the k remains the same, but in this case we increment an initial random k every time and then we encrypt it, so in this way k remains secure.

An example of a forged signature based on the universal hash function is given by the signature:

```
r = 776518120378836367956166965531631433446605258 29
s = 396326171081672787770131088060552525632448224996
```

build over the message:
message = *Message one*
and this works also for the message:
message2 = *Message two*
encoded as:

```
m2 = b'Message two-r\x98\r\x8c\xd6Ŷ\rr\xb8L\xe5\xb52\xca4\xe1\xf7e'
```

The forge of the signature is possible working on the preimage of r and the padding. Setting the last 20 bytes of the new message to 0 we can create a padding equal to the preimage of r minus the concatenated message modulo q. The concatenated message is computed as in the verification procedure. In this way we obtain a message with the same signature of the previous one adding the padding to the message converted as an integer and the re-converted in bytes(m2 above).

5 Lab05: Certificates and Transport Layer Security

5.1 Creating and using certificates

The creation of the certificates is based on the type of key contained in the certificate (RSA or ECDSA) and the hash function used to sign it (SHA1 or SHA256). The subject of the certificate is the hostname and the issuer of the certificate will be the same since this is a self-signed certificate.

Table 6 reports the size of the created certificates and, as expected, it increase with the RSA algorithm. The reason of this is that the problem of discrete logarithm used in ECDSA is much more difficult than prime factorization of the RSA, so ECDSA encryption will require public keys of smaller size.
A smaller difference (32 bit, so the dimension of a word) is also visible in the EDCSA case using SHA256 instead of SHA1.

	EDCSA	RSA
SHA1	652byte	1.192byte
SHA256	656byte	1.192byte

Table 6: Size of the different certificates

The certificate also leads to localhost as an alternative name, so the server use it running on localhost and the result will be an alert from the browser. This happens obviously because our certificate is seen as a non-secure certificate and if you want to continue browsing you must specify that you are doing it consciously.

5.2 TLS handshake

We try to connect to the local server with the default values of the certificate and the error seen is:

```
ssl.SSLCertVerificationError: [SSL: CERTIFICATE_VERIFY_FAILED]
certificate verify failed: self signed certificate (_ssl.c:1056)
```

Instead, loading the certificate as a trusted certificate (with max TLS version negotiated by the client: TLSv1.2) the result of the TLS negotiation is:

```
TLS version: TLSv1.2
Cipher suite: ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2
key-exchange: Kx=ECDH
server authentication: Au=RSA
data encryption and authentication: Enc=AESGCM(256) Mac=AEAD(it
is only the constructor)
hash function: SHA384
```

Then we set the maximum TLS version negotiated by the client to TLSv1.3 and the result doesn't change because the version used by the server is still 1.2 and so all versions above will support it. The result is:

TLS version: `TLSv1.2`
Cipher suite: `ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH Au=RSA Enc=AESGCM(256) Mac=AEAD` Therefore we set the maximum TLS version negotiated by the client to TLSv1.1 and the result is: TLS version: `TLSv1`
Cipher suite: `ECDHE-RSA-AES256-SHA TLSv1 Kx=ECDH Au=RSA Enc=AES(256) Mac=SHA1`

The main weakness is the hash function SHA1 and also the data encryption that is without the GCM, so less secure. The we try removing ECDHE from the cryptographic algorithm negotiated by the client and therefore leaving only RSA:

TLS version: `TLSv1`
Cipher suite: `AES256-SHA SSLv3 Kx=RSA Au=RSA Enc=AES(256) Mac=SHA1`

Also the key-exchange method becomes RSA and this implies an additional vulnerability wrt the previous case.
In this case, if the server certificate has an ECDSA key the message shown is an alert because there is no correspondence between the client's request using RSA as signature algorithm and the server using ECDSA. If we set the maximum and minimum TLS version negotiated by the client to TLSv1.3 it obviously doesn't work because the server has a lower version (v1.2) which doesn't support v1.3.

```
ssl.SSLError: [SSL: TLSV1_ALERT_PROTOCOL_VERSION] tlsv1 alert
protocol version (_ssl.c:1056)
```

Changing also the maximum TLS version negotiated by the server to TLSv1.3, always without ECDHE, the result is:

TLS version: `TLSv1.3`
Cipher suite: `TLS_AES_256_GCM_SHA384 TLSv1.3 Kx=any Au=any Enc=AESGCM(256) Mac=AEAD`

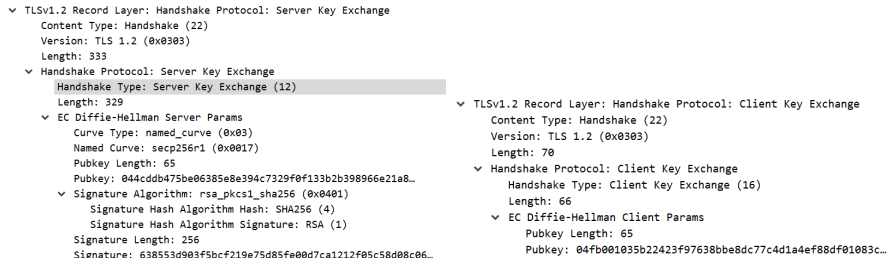
The particularity of v1.3 is that it works only with ECDHE, also without the consent of the client, because it searches always the best secrecy and takes that forward the other preferences. Also the possible vulnerabilities given by the use of SHA1 and AES without the Counter Mode are exceeded. It sets the best quality algorithms.

5.3 Inspecting TLS handshake messages

Using Wireshark we are able to inspect the TLS handshake messages also connecting to www.polito.it on port 443. Setting the maximum TLS version negotiated by the client to TLSv1.3 and the minimum to TLSv1 we can see these results:

TLS version chosen by he server: `TLSv1.2`
Cipher suite: `TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)`

The server and client key-exchange messages are reported in **Figure 1**.



(a) Server (b) Client

Figure 1: Key Exchange Messages

Here, if the client removes ECDHE from the offered cipher suites, the hand shake fails with a fatal error as shown in **Figure 2**. This is probably due to an incompatibility of cipher suites in use by the client and the server.

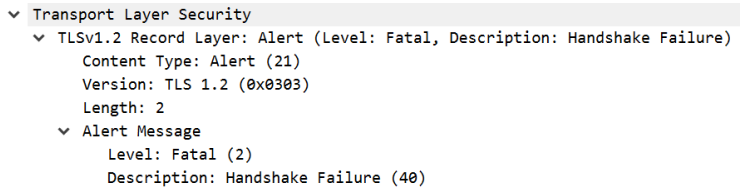


Figure 2: Alert

While if we switch the maximum TLS version negotiated by the client to TLSv1.1 it will generate an alert message with a fatal error due to the protocol version: it can't use a version unsupported.

Now connecting to www.google.com on port 443 and setting the maximum TLS version of client to TLSv1.3 and the minimum to TLSv1 the version chosen by the server and the cryptographic algorithms are:

TLS version: `TLSv1.2`
Cipher suite: `TLS_AES_256_GCM_SHA384 (0x1302)`

The server's certificate here is not visible because it is in the application data and not in the hand shake message.

The version 1.3 redesigns all the key derivation functions: the key-exchange parameters are included in the ClientHello and ServerHello extensions. In this case the key exchange messages and the parameters useed are:

```
Key Exchange: 6532c744de252bd510b78d196e7b0166168e690916028ad6...
Parameters: Group:  x25519, Key Exchange length:  32
```

Then, setting the maximum TLS version negotiated by the client to TLSv1.1 it is possible to see that there are no more restriction or changes due to the v1.3, so the hand shake goes back to the "normal" procedure.
It is a different result wrt that obtained with the www.polito.it connection, here there is no alert message.

If the client removes ECDHE from the offered cipher suites the hand shake messages seen are always the ClientHello and ServerHello, the cipher Suite is

```
TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
```

the key exchange is inserted in the client's message and the algorithm used is `RSA`.