Tutorial 2

Computer Security
Orfeas Stefanos Thyfronitis Litos
School of Informatics
University of Edinburgh

In this tutorial for the Introduction to Computer Security course we cover Cryptography. You are free to discuss these questions and their solutions with fellow students also taking the course, and also to discuss in the course forum. Bear in mind that if other people simply tell you the answers directly, you may not learn as much as you would by solving the problems for yourself; also, it may be harder for you to assess your progress with the course material.

1 Hash functions

Let $\mathcal{M} = \{0,1\}^*$ and $\mathcal{T} = \{0,1\}^n$ for some integer n.

1. Explain what does it mean for a hash function $h : \mathcal{M} \to \mathcal{T}$ to be one-way.
2. Explain what does it mean for a hash function $h : \mathcal{M} \to \mathcal{T}$ to be collision resistant.
3. Suppose $h : \mathcal{M} \to \mathcal{T}$ is collision resistant. Is h also one-way? If so, explain why. If not, give an example of a collision resistant function that is not one-way.
4. Suppose $h : \mathcal{M} \to \mathcal{T}$ is one-way. Is h also collision resistant? If so, explain why. If not, give an example of a one-way function that is not collision resistant. Suppose the DLOG assumption is true.
5. Bob is on an under cover mission for a week and wants to prove to Alice that he is alive each day of that week. He has chosen a secret random number, s, which he told to no one (not even Alice). But he did tell her the value $H = h(h(h(h(h(h(h(s)))))))$, where h is a cryptographic hash function. During that week Bob will have access to a broadcast channel, so he knows any message he sends to Alice will be received by Alice. Unfortunately Bob knows that Eve was able to intercept message H. Explain how Bob can broadcast a single message everyday that will prove to Alice that he is still alive. Note that your solution should not allow anyone (and in particular Eve) to replay any previous message from Bob as a (false) proof that he still is alive.

2 Symmetric key encryption

2.1 Theory

Let's now do a quick refresher on what we've learned on symmetric key encryption schemes. Such a scheme consists of a pair of functions, one for encryption (takes a key and a plaintext, gives a ciphertext) and one for decryption (takes a key and a ciphertext, gives a plaintext):

$$Enc : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^*$$
$$Dec : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^* \ .$$

For such a scheme to work as expected, we require that if an encrypted message is then decrypted with the same key, we should get back the original message:

$$\textit{Consistency:} \forall k \in \{0,1\}^\lambda, \forall m \in \{0,1\}^* : Dec(k, Enc(k,m)) = m \ .$$

This also frequently referred to as *correctness*. Unfortunately that's not enough because it doesn't say anything regarding security[1]. In particular we'd want some notion of "the ciphertext leaks no bits of the plaintext". The formal version of this notion is *semantic security*, which is exactly the kind of security you should expect from any respectable symmetric key encryption scheme.

A little bit more unavoidable theory before the practice: The most widely used building block for symmetric key encryption schemes are *block ciphers*[2]. A block cipher looks mostly like an encryption scheme but takes as plaintext (or ciphertext) a small, fixed-length message, which means that it's not suitable for encrypting my latest 3-page-long love letter.

The simplest solution is to split up the plaintext into chunks of the correct size (padding the last one if needed) and encrypt each chunk separately, then do the reverse for decryption. That's also a bad idea. If my letter contains the phrase "I love you Alice" many times (and the repetitions happen to be aligned with the block size), an attacker that sees the ciphertext can easily deduce that I use the same phrase over and over[3]. That's not semantically secure.

To get security back, we have to somehow make each block depend on the rest[4]. There is an abundance of ways to achieve this dependence. These ways are called *modes of operation*, each with its own pros and cons[5]. The naive way described above is the Electronic Codebook (ECB) and we never use it in practice[6]. The modes of operation we'll use are Ciphertext Block Chaining (CBC), Counter (CTR), and Cipher Feedback (CFB). We will see the most crucial details as we go during the practical part in Section 4, but a full-blown comparison of all modes is beyond the aims of this tutorial[7]. Refer to the documentation[8] if you're unsure how to use any of the modes.

2.2 Theoretical exercises

Let $(\mathcal{E}_{32}, \mathcal{D}_{32})$ be a secure (deterministic) block cipher with 32-bits key size and 32-bits message size. We want to use this cipher to build a new (deterministic) block cipher $(\mathcal{E}_{64}, \mathcal{D}_{64})$ that will encrypt 64-bits messages under 64-bits keys. We consider the following encryption algorithm. To encrypt a message M under a key K, we split M into two parts M_1 and M_2, and we also split K into two parts K_1 and K_2. The ciphertext C is then computed as $\mathcal{E}_{32}(K_1, M_1)||\mathcal{E}_{32}(K_2, M_2)$. In other words we concatenate the encryption of M_1 under K_1 using $\mathcal{E}_{32}$, with the encryption of M_2 under K_2 using $\mathcal{E}_{32}$.

[1] Observe that the useless scheme $Enc(k,x) = Dec(k,x) = x$ satisfies correctness.

[2] The reason: they are generally faster than the alternative, *stream ciphers*.

[3] If that doesn't seem scary enough, take a look at the images in `https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#ECB`.

[4] Therefore achieving *diffusion*.

[5] Who would have guessed that cryptography is complicated...

[6] `https://www.reddit.com/r/ProgrammerHumor/comments/6m6bvv/all_block_cipher_modes_are_beautiful/`

[7] In practice, we often need *authenticated encryption*. We then use the GCM mode of operation.

[8] `https://pycryptodome.readthedocs.io/en/latest/src/cipher/classic.html`

1. What is the corresponding decryption algorithm? To justify your answer prove that the consistency property is satisfied.

2. Consider the following game.

   - In the first phase, the attacker chooses a few 64-bit plaintext messages $M_1, \ldots, M_n$ and gets back from an encryption oracle the corresponding ciphertexts $C_1, \ldots, C_n$ under some key K that he does not know. The attacker gets to know that C_i is the ciphertext corresponding to M_i for all $i \in \{1, \ldots, n\}$.
   - In the second phase the attacker builds two 64-bit messages M_A and M_B and gets back C which is the encryption under K either of M_A or M_B. But now, the attacker doesn't know if the plaintext underlying C is M_A or M_B and has to guess it.

   Informally, a symmetric cipher is said to be vulnerable to a chosen plaintext attack if the attacker can guess (with high probability) which of M_A or M_B is the plaintext corresponding to C. Show that the new cipher $(\mathcal{E}_{64}, \mathcal{D}_{64})$ is subject to a chosen plaintext attack even though $(\mathcal{E}_{32}, \mathcal{D}_{32})$ is not.

3 Cryptographic Proofs

1. Prove that in a classroom of 23 students the probability that any two students have the same birthday is over 50%, a.k.a the Birthday Paradox. Suppose birthdays are distributed uniformly over the 365 days of the year.

   Note (6)

2. Prove that, given a collision-resistant one-way compression function, the Merkle-Dåmgard construction builds a collision resistant hash function.

3. Prove that the RSA encryption scheme is consistent: Given a public - secret keypair $(n, e), d$, it is

   $$Dec_{RSA}(d, Enc_{RSA}((n, e), m)) = m \pmod{n} .$$

   Hint: Use Euler's theorem.

4 Practical symmetric key exercises

> Don't roll your own crypto, bro.
>
> Joseph Cox, VICE.com

Here we will gain some hands-on experience with symmetric encryption leveraging the `pycryptodome`[9] [10] library provided for Python 3. The library is available on DICE machines, so don't just read but try out the code yourself!

There are many more crypto libraries out there, implemented in and available for every (useful) programming language. You can choose whichever suits you best depending on your usecase, the choice we made here is quite arbitrary. But please don't ever implement your own

[9] `https://www.pycryptodome.org`

[10] In fact we use `pycryptodomex`, because `pycryptodome` uses the same module name as `pycrypto`, an outdated crypto library that we unfortunately can't uninstall from DICE.

crypto library and proceed to use it in production code – you'll probably do a worse job than experts both in terms of speed and (more crucially) security[11].

The by far most commonly used block cipher is AES. In fact it's so common that it has been implemented in hardware and relevant instructions are widely available on commercial CPUs[12], so it is blazingly fast. We will therefore use this cipher for our tutorial. The block size of AES is 128 bits (a.k.a. 16 bytes) and it needs a key of length 128, 192, or 256 bits (i.e. 16, 24, or 32 bytes). Just to be on the safe side, I recommend using 192 bits, because it is much faster and in practice as secure as 256 bits[13] – that is, until quantum computers roam the land.

Let's see the most basic example: Encrypting a phrase with a key and then decrypting back to the original. We will use the CFB mode for this. This mode needs some good quality randomness for the IV[14], which we will get from the `Random` module of `pycryptodome` (alternatively we could have used the `secrets` module). A general piece of advice: Never, ever, ever use randomness that is not advertised as "cryptographically secure" for your cryptographic tasks. For example, the `random` Python module gives very predictable, and thus bad quality, randomness and so we stay away from it. To be honest, in production I'd rather use a library that handles the IV internally and does not expose it at all to protect myself from footguns.

aes-basic.py

```python
from Cryptodome.Cipher import AES
from Cryptodome import Random

# key and plaintext must be bytes(), not strings
key = b"I am a short key"
plaintext = b"Alice I love you"

# may be public, but must be chosen at random
iv = Random.new().read(AES.block_size)
# or, if using 'secrets':
#iv = secrets.randbits(AES.block_size*8).to_bytes(AES.block_size, 'big')

# this object remembers how many bytes have been encrypted
encrypter = AES.new(key, AES.MODE_CFB, iv)

ciphertext = encrypter.encrypt(plaintext)
print("Gibberish incoming:", ciphertext.hex())

# the same as the encrypter, but initialized anew
decrypter = AES.new(key, AES.MODE_CFB, iv)

decrypted = decrypter.decrypt(ciphertext)
if plaintext == decrypted:
    print("All good!")
```

This is the expected output:

```
$ python3 aes-basic.py
Gibberish incoming: 08054ccf5d74557b5ee98628f57f5283
All good!
```

Observe that we had to construct two `AES` objects, one for encryption and one for decryption. This is because all modes of operation (apart from ECB) encrypt the same message to different

[11] Writing a toy implementation of a cryptographic algorithm/protocol on the other hand may be an educational and fun experience. It may even be the first step for you to become a professional crypto coder!

[12] Which makes the following phrase completely valid: "Our CPU boasts of an instruction set enabling a range of commonly used operations, such as bitshifts, logical AND, primary school arithmetic and polynomial multiplication over the Galois Field 2^8."

[13] Different key sizes also result in slightly different variations of the algorithm, but the API is exactly the same.

[14] *initialization vector*, a random number needed by the CFB mode

ciphertexts depending on how many bytes have already been encrypted during this run of the mode of operation. To be decrypted properly, each ciphertext must be processed by an AES object that has already decrypted as many bytes as were encrypted before the encryption of the ciphertext (and of course with the same IV). Let's see a clarifying example:

aes-mode.py

```
from Cryptodome.Cipher import AES
from Cryptodome import Random

# Randomly generated keys are better than hardcoded ones
key = Random.new().read(AES.key_size[1]) # 192 bits
plaintext1 = b"My love for Alice is immeasurable" # length: 34
plaintext2 = b"It is quite true that Bob loves Alice" # length: 38

iv = Random.new().read(AES.block_size)

######################
## Correct usage: one object for encryption, one for decryption
## E.g. for chat between two, a pair of AES objects for each client

aes1 = AES.new(key, AES.MODE_CFB, iv)
aes2 = AES.new(key, AES.MODE_CFB, iv)

ciphertext1 = aes1.encrypt(plaintext1)
# plaintext1 encrypted at positions 0:34, aes1 at 34

ciphertext2 = aes1.encrypt(plaintext2)
# plaintext2 encrypted at positions 34:(34+38), aes1 at 34+38

decrypted1 = aes2.decrypt(ciphertext1)
# decrypting at positions 0:34, aes2 at 34

decrypted2 = aes2.decrypt(ciphertext2)
# decrypting at positions 34:(34+38) at 34+38

assert(decrypted1 == plaintext1)
assert(decrypted2 == plaintext2)

######################
## Common error 1: encrypting and decrypting with same object

aes3 = AES.new(key, AES.MODE_CFB, iv)

ciphertext3 = aes3.encrypt(plaintext1)
# decrypted3 = aes3.decrypt(ciphertext1) -- TypeError

######################
## Common error 2: encrypting and decrypting in wrong order

aes4 = AES.new(key, AES.MODE_CFB, iv)
aes5 = AES.new(key, AES.MODE_CFB, iv)

ciphertext1 = aes4.encrypt(plaintext1)
ciphertext2 = aes4.encrypt(plaintext2)

decrypted4 = aes5.decrypt(ciphertext2)
decrypted5 = aes5.decrypt(ciphertext1)

assert(decrypted4 != plaintext2)
assert(decrypted5 != plaintext1)
```

Exercise 1. Encrypt your name with AES-192-CTR, that is using counter mode, under a random key and IV and successfully decrypt the resulting ciphertext.

Exercise 2. Decrypt the ciphertext `b"8cb8419e1b2be1046bf9100be7dc72729001dc59783f241c3a2456bacf2b22b61be78d13854aa1135bb7e83eeb12060e94815085ad6357caf9f277f4345b209e6efae04fffa3620ec88117c1afca8662c761c0557d4d97bbfdb4fa812f33feca"` which has been encrypted with AES-256-CBC, under key `b"4f536b9edab33e491fc398fc173f6633d0ad7584fafb790f719f5148fe192d44"` and IV `b"c56148e34eb2f8a858b9fa621f211e41"`. Ensure the plaintext is in English.

HASH FN

1) A fn h is a OWF if for all $y \in T$, there is no efficient algo. which given y can compute x s.t. $h(x) = y$

2) A fn h is collision-resistant if there is no efficient algo. tht. can find 2 messages $m_1, m_2 \in M$ s.t. $h(m_1) = h(m_2)$

3) Let g be a hash fn which is collision resistant and maps arbitrary-length inputs to $n-1$ outputs. Consider the fn h defined as:

$$h(x) = \begin{cases} 1 \| x & \text{if } x \text{ has bit-length of } n-1 \\ 0 \| g(x) & \text{otherwise} \end{cases}$$

where $\|$ denotes concatenation.

Then h is an n-bit hash fn which is collision resistant but not one way.

4) Let p a large prime, g a generator of $\mathbb{Z}_p$, h be the fn $h(x) = g^x \bmod p$.

The fn h is one way frm DLOG assumpn:

↳ inverting exponentiation over a discrete grp. is computationally hard.

The fn is NOT collision resistant:

↳ choose x_1 and $x_2 = x_1 + (p-1)$, then

$$g^{x_2} = g^{x_1 + (p-1)} = g^{x_1} \cdot g^{p-1} = g^{x_1} \bmod p$$

5) Let d range from 1 to 7, denote day of the week.

On day d, Bob broadcasts message $h^{7-d}(s)$.

Bc. of one-wayness of h, no one else can compute $h^{7-d}(s)$ but Bob.

But anyone (and in particular Alice) can verify that

$$\boxed{h^{7-(d-1)}(s) = h(h^{7-d}(s))}$$

→ message received on day $d-1$ is the hash of the message received on day d

↓

Proving tht. Bob is alive.

2. SYMMETRIC ENCRYPTION

1) Split C into two parts: C_1, C_2 and compute $D_{32}(K_1, C_1) \| D_{32}(K_2, C_2)$

Proof of consistency:

$$D_{64}(K_1 \| K_2,\ E_{64}(K_1\|K_2,\ M_1 \| M_2))$$

$$= D_{64}(K_1 \| K_2,\ E_{32}(K_1, M_1) \| E_{32}(K_2, M_2))$$

$$= D_{64}(K_1, E_{32}(K_1, M_1)) \| D_{64}(K_2, E_{32}(K_2, M_2))$$

$$= M_1 \| M_2$$

2) Let $M_1 = 0^{32} \| 0^{32}$ and $M_2 = 1^{32} \| 1^{32}$

Let $C_1 = E_{32}(K_1, 0^{32}) \| E_{32}(K_2, 0^{32})$

$C_2 = E_{32}(K_1, 1^{32}) \| E_{32}(K_2, 1^{32})$

Let $M_A = 0^{32} \| 1^{32}$ and $M_B = 1^{32} \| 0^{32}$

Given C_1 and C_2, the attacker can trivially compute

$$E_{64}(0^{32} \| 1^{32}) = E_{32}(K_1, 0^{32}) \| E_{32}(K_2, 1^{32}) \quad \text{and}$$

$$E_{64}(1^{32} \| 0^{32}) = E_{32}(K_1, 1^{32}) \| E_{32}(K_2, 0^{32})$$

and thus win the game w/ probability 1.

$\therefore$ This new scheme is not secure under CPA.

3) KPA → given a plaintxt M and its ciphertxt C under key K not known to attacker, the attacker can recover key K in a time significantly < than by a brute force

Show tht. (E_{64}, D_{64}) is vulnerable to a KPA.

- Trying all possible keys in the key space → 2^{64}
- First, use brute-force to recover K_1. We know $C_1 = E_{32}(K_1, M_1)$ and we know M_1, C_1, so try all possibilities for K_1 and see which one is consistent
- Use brute-force to recover K_2

$\therefore 2^{32} + 2^{32} = 2^{33}$ trial decryptions in total.

CRYPTOGRAPHIC PROOFS

1) Let A be the event where there exists at least one pair of students w/ same bday.

$$P(A) = 1 - P(A')$$

We calculate $P(A')$ as ff:

- 1 student: no collision → probability of collision $= 1$

  2 students: probability of collision $= \frac{364}{365}$

  3 students: probability of collision $= \frac{364}{365} \times \frac{363}{365}$

  since the bday of 3rd student is independent of the first two and now 2 days of the yr. are alrdy taken

- We deduce that:

$$P(A') = \frac{364}{365} \times \frac{363}{365} \times \cdots \times \frac{365-22}{365} \approx 0.4927$$

Thus $P(A) = 1 - P(A') \approx \boxed{0.5073}$

3) Since $d = e^{-1} (\text{mod } \phi(n))$, $ed \equiv 1 (\text{mod } \phi(n)) \Rightarrow \underline{\exists a \in \mathbb{N} \cdot ed = 1 + a\,\phi(n)} \cdots (1)$

Thus

$$\begin{aligned} Dec_{RSA}(d, Enc_{RSA}((n,e), m)) &= (m^e)^d \text{ mod } n \\ &= m^{ed} \text{ mod } n \\ &\overset{(1)}{=} m^{a\phi(n)+1} \text{ mod } n \\ &= m^{a\phi(n)} \cdot m \text{ mod } n \\ &\overset{\text{Euler}}{=} m \text{ mod } n \end{aligned}$$