Homework 2 – Signatures & Symmetric Cryptography

Cryptography and Security 2024

- You are free to use any programming language you want, although Python/SAGE is recommended.
- Put all your answers **and only your answers** in the provided `[id]-answers.txt` file where `[id]` is the student ID.[1] This means you need to provide us with all `Q`-values specified in the questions below. Personal files are to be found on Moodle under the feedback section of Parameters HW2.
- **Please do not put any comment or strange character or any new line** in the submission file and do **NOT** rename the provided files.
- Do **NOT** modify the `SCIPER`, `id` and `seed` headers in the `[id]-answers.txt` file.
- **Submissions that do not respect the expected format may lose points.**
- We also ask you to submit your **source code**. This file can of course be of any readable format and we encourage you to comment your code. Notebook files are allowed, but we prefer if you export your code as normal textual files containing Python/SAGE code. If an answer is incorrect, we may grant partial marks depending on the implementation.
- Be careful to always cite external code that was used in your implementation if the latter is not part of the public domain and include the corresponding license if needed. Submissions that do not meet this guideline may be flagged as plagiarism or cheating.
- Some plaintexts may contain random words. Do not be offended by them and search them online at your own risk. Note that they might be really strange.
- Please list the name of the **other** person you worked with (if any) in the designated area of the answers file.
- Corrections and revisions may be announced on Moodle in the "News" forum. By default, everybody is subscribed to it and does receive an email as well. If you decided to ignore Moodle emails, we recommend that you check the forum regularly.
- The homework is due on Moodle on **December 15th, 2024** at 23h59.

[1] Depending on the nature of the exercise, an example of parameters and answers will be provided on Moodle.

Exercise 1 Unconventional Symmetric Cryptography

A junior crypto apprentice attempts to come up with several new cryptography schemes. In this exercise, we will try to break them.

Question 1.1 RSA-OFB

After seeing several modes of operations for block ciphers in class. The crypto apprentice thinks: *"What if we use RSA instead of a block cipher in these modes of operations. We can assume that the secret key is composed of both* pk *and* sk *of RSA to allow both encryption and decrption. If we use RSA-2048, we can even support a larger block size compare to 128 bits in AES!"*. Then the apprentice decides to use RSA in OFB mode.

Given an RSA public key (e, N) as Q1a_pk, a Python list Q1a_y of size $n+1$ with an IV as the first element followed by n encrypted blocks, report the list of n decrypted blocks under Q1a_x. You can verify your solution by using the Q1a_xhash value and `verify_Q1a` function under `utils.sage`.

Question 1.2 Auction using AES

The crypto apprentice is tasked to create a secure auction mechanism, where the bidders are supposed to send their bids over an insecure channel without being able to observe other people's bids. During this whole process we assume that we have a trusted auctioneer. Here is the flow of the idea:

- **Encrypted Bidding**: Each participant will agree on a seperate AES-256 key with the auctioneer. This way, the auctioneer will be able to decrypt all the bids and decide on the winner. Each participant encrypts their bid with AES-256 in CBC mode. Figure 1 shows the full details.

Bidder.EncryptBid(k, bid)	Auctioneer.DecryptBid(k, c)
1 : **if** $bid < 10000 \vee bid > 99999$	1 : $m \leftarrow$ AES256CBC.Decrypt(k, c)
2 : **abort**	2 : // Extracts the bytes from index 27 to 31 (inclusive).
3 : $m \leftarrow$ `Dear Auctioneer:My bid is $bid`	3 : $bid \leftarrow m[27:32]$
4 : $c \leftarrow$ AES256CBC.Encrypt(m, k)	4 : **return** bid
5 : **return** c	

Figure 1: Bid Encryption protocol. **Example:** If a bidder encrypts the bid 12345, the resulting message m is `"Dear Auctioneer:My bid is $12345"`. For convenience, `AES256CBC_encrypt(k, m)` and `AES256CBC_decrypt(k, m)` functions are provided in `utils.sage`

You realized that you can listen on the network and alter the ciphertexts being sent to the auctioneer. You also had an insider information about how much one of your main competitors is going to bid. To guarantee your winning at a cheaper price, you would like to set the bids of your main competitor to the minimum bid which is \$10000. Given your competitor's message Q1b_m, a corresponding ciphertext encrypted with AES256CBC as Q1b_c construct

a ciphertext Q1b_cnew such that the result of Auctioneer.DecryptBid(k, Q1b_cnew) is 10000. Note that both ciphertexts are Python byte strings with the first 16 bytes corresponds to the IV for the CBC mode.

Question 1.3 Auction using AES with commitments

After realizing the problem with the previous attempt. The crypto apprentice comes up with a new scheme, including commitments to bids with the following flow:

1. **Commitment Phase**: Each party commits to their bids.
2. **Encrypted Bidding Phase**: Each party encrypts their bids using the same approach before.
3. **Bid Opening Phase**: Each party opens their committed bids.

Again, during this whole process we assume that we have a trusted auctioneer. The idea is that if the commitments are binding, even if we alter the ciphertext in the encrypted bidding phase, the auctioner will catch that the resulting bid after decryption will be inconsistent with the opened commitment. Hence, the scheme should be more secure.

The commitment scheme being used is defined in Figure 2. In summary, given an elliptic curve $E(\mathbb{F}_p)$ with a subgroup of prime order q with generators G and H. With the public parameters (p, q, G, H) we can commit to a message m by sampling a random opening value r from $\mathbf{Z}_q$ and compute $bid \cdot G + r \cdot H$ as our commitment. Note that in order to make the public parameters smaller and save some bandwith, the auctioneer publishes k instead of H (since the scalar k has smaller representation than a point on the elliptic curve) as part of the public parameters (since given G and p, H can be inferred from k).

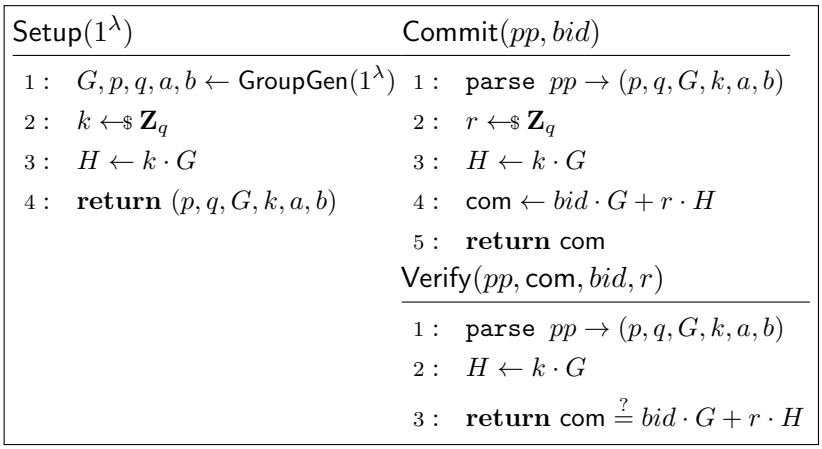

Figure 2: Commitment Protocol. GroupGen(1^λ) generates an elliptic curve over $\mathbb{F}_p$ (with p of size λ bits) with subgroup of prime order q defined by the equation $y^2 = x^3 + ax + b$ with generator G.

Given public parameters (p, q, G, k, a, b) as Q1c_pp, a commitment Q1c_com and its opening value Q1c_r for Q1c_bid. Construct a new opening value for Q1c_bidnew such that when the commitment is computed, it is equal to Q1c_com. Report this new opening value under

Q1c_rnew. Curve points such as G and Q1c_com are represented as a python tuple of (x, y) coordinates.

Exercise 2 Q2: An insecure signature scheme based on quadratic residues

In this exercise, we construct an RSA-like signature scheme that is *non-trivially insecure*. We define the following signature scheme:

KeyGen:

1: Sample primes p, q such that $p \approx q$.
2: Compute $N = p \cdot q$
3: Sample $k \in \mathbb{Z}_N$
4: $\mathsf{pk} = (k, N)$
5: $\mathsf{sk} = (k, p, q)$
6: **return** $(\mathsf{sk}, \mathsf{pk})$

Sign:

1: **Input:** sk, m
2: Find $a, b \in \mathbb{Z}_N$ such that $a^2 + kb^2 \equiv m \pmod N$
3: $\sigma = (a, b)$
4: **return** σ

Verify:

1: **Input:** $\mathsf{pk}, m, \sigma = (a, b)$
2: **return** $a^2 + kb^2 \stackrel{?}{\equiv} m \pmod N$

Figure 3: Our RSA-like signature scheme

Question 2.1 Q2a: Implementing the Signature Scheme

Implement the signature scheme. Specifically, given inputs `Q2a_p`, `Q2a_q`, `Q2a_N`, `Q2a_k`, `Q2a_m`, compute and return `Q2a_a`, `Q2a_b` non zeros that form a valid signature for `Q2a_m`.

Question 2.2 Q2b: A Reduction Mechanism

We now attempt to break the scheme's security by forging signatures. From now on, we fix `Q2_N`.

Consider the following observations:

Lemma 1. *For any a_1, a_2, b_1, b_2 in any ring, the following identity holds:*

$$\left(a_1^2 + kb_1^2\right)\left(a_2^2 + kb_2^2\right) = (a_1a_2 \pm kb_1b_2)^2 + k(a_1b_2 \mp a_2b_1)^2$$

Lemma 2. *If $\gcd(b, N) = 1$, then:*

$$a^2 + kb^2 \equiv m \pmod N \iff a'^2 - mb'^2 \equiv -k \pmod N$$

where the transformation is defined as $a' = a/b$ and $b' = 1/b$.

Reduction

Let q_0 be a prime such that $-k \in \mathsf{QR}_{q_0}$. Construct an algorithm that returns a, b such that:

$$a^2 + kb^2 \equiv q_0 t^{-1} \pmod N$$

where $|t|$ or $|N - t| < \frac{3}{2}\sqrt{k}$.

Given a list `Q2b_q[j]` and `Q2b_k[j]`, compute and return `Q2b_a[j]`, `Q2b_b[j]`, `Q2b_t[j]` all non zeros and solutions of the equation

$$\texttt{Q2b\_a[j]}^2 + \texttt{Q2b\_k[j]} \cdot \texttt{Q2b\_b[j]}^2 = \texttt{Q2b\_q[j]} \cdot \texttt{Q2b\_t[j]}^{-1} \mod \texttt{Q2\_N}$$

for all j.

Hint: Consider the recursive sequence:

$$q_i \text{ such that } q_i q_{i-1} = x_{i-1}^2 + k \text{ in } \mathbb{Z}.$$

$$x_i = \min\left(x_{i-1} \mod q_i, q_i - (x_{i-1} \mod q_i)\right) \text{ in } \mathbb{Z}.$$

Question 2.3 Q2c: Fully Efficient Attack

Using the reduction mechanism from Q2, find an efficient algorithm to forge a signature. Given `Q2c_k`, `Q2c_m`, compute `Q2c_a`, `Q2c_b` non zeros such that they form a valid signature of `Q2c_m`.

Hints:

1. Consider finding u, v such that $(u^2 + kv^2)m = q_0 \mod N$ for some q_0 as in Q2.
2. Think of the Euclid algorithm structure.

Exercise 3 Linear Key Signing

Let $\mathbb{G}$ be a group defined over a prime p with a prime order subgroup of order q. Consdier the following signature scheme defined over $\mathbb{G}$, note that the hash function H being used is given as a black-box under `utils.sage`:

- KeyGen(1^λ): On security parameter 1^λ, generate the secret key sk and the public key pk.
  1. Randomly sample $x \leftarrow_\$ \mathbf{Z}_q$ and compute $X \leftarrow g^x \mod p$.
  2. Output $(\mathsf{sk} = x, \mathsf{pk} = X)$.
- Sign(sk, m): Sign a message m into a signature σ with the secret key sk.
  1. Randomly sample $t \leftarrow_\$ \mathbf{Z}_q$ and compute $r \leftarrow g^t \mod p$.
  2. Compute $h \leftarrow H(m \| r)$.
  3. Compute $s \leftarrow (\mathsf{sk} \cdot h + t) \mod q$.
  4. Output $\sigma = (h, s)$.
- Verify(pk, m, σ): Verify that σ is indeed a valid signature of message m.
  1. Parse $(h, s) \leftarrow \sigma$.
  2. Compute $r' \leftarrow g^s \cdot \mathsf{pk}^{-h} \mod p$.
  3. Compute $h' \leftarrow H(m \| r')$.
  4. Output $h \stackrel{?}{=} h'$.

Question 3.1

Implement the signing procedure for this scheme. You are given public parameters (p, q, g) as Q3a_pp, a message Q3a_m, a random value to be used in signing Q3a_t and a secret key Q3a_sk. Report the resulting signature under Q3a_sig.
Note: Make sure that the signature is correct by implementing verification as well.

Question 3.2

Let f be an affine function defined over Z_q. It is specified by two values $\alpha, \beta \in Z_q$ such that $f(x) = \alpha \cdot x + \beta$.

Given public parameters Q3b_pp, a valid signature (h, s) as Q3b_sig signed under an unknown key sk for message Q3b_m and a given public key Q3b_pk. Construct a valid signature Q3b_signew for the same message that is valid under a new secret key $f(\mathsf{sk})$. Specify this function under Q3b_f as a python tuple (α, β) such that $\alpha, \beta \in Z_q^*$ (i.e. $q > \alpha > 0$ and $q > \beta > 0$).