2 The MMAP and EMAP Protocols

The Minimalist Mutual Authentication Protocol (MMAP) and Efficient Mutual Authentication Protocol (EMAP) were first proposed in [1, 2]. The steps of the two protocols, starting with MMAP, are described as follows. In the MMAP protocol, the tag and the reader both store four secret keys, K_1, K_2, K_3, and K_4. The tag also stores an identifier IDP, which is used by the reader to identify the tag, as well as a secret quantity ID. All keys and identifiers are assumed to be bit strings of length k. **All computations are performed modulo 2^k. The notations $\oplus$, $\wedge$, and $\vee$ denote bit-wise XOR, AND, and OR, respectively.** The steps in MMAP are:

1. The reader sends a *Hello* message to the tag, which powers the tag.
2. The tag responds with IDP.
3. Based on the value of IDP, the reader looks up the values of K_1, K_2, K_3, and K_4. The reader then generates two random bit strings n_1 and n_2, and sends a message to the tag consisting of three bit strings, $A = IDP \oplus K_1 \oplus n_1$, $B = (IDP \wedge K_2) \vee n_1$, and $C = IDP + K_3 + n_2$.
4. Upon receiving A, B, and C, the tag computes $n_1 = A \oplus IDP \oplus K_1$ and $n_2 = C - IDP - K_3$. The tag then checks if $B \stackrel{?}{=} (IDP \wedge K_2) \vee n_1$. If so, the tag authenticates the reader and proceeds to the next step. Otherwise the tag terminates the protocol.
5. The tag sends a message to the reader consisting of the bit strings $D = (IDP \vee K_4) \wedge n_2$ and $E = (ID + IDP) \oplus n_1$.
6. The reader computes $ID = E \oplus n_1 - IDP$.
7. The tag and reader each update the values of IDP, K_1, K_2, K_3, and K_4 as follows:

$$\begin{aligned} IDP^{(n+1)} &= (IDP^{(n)} + (n_1 \oplus n_2)) \oplus ID \\ K_1^{(n+1)} &= K_1^{(n)} \oplus n_2 \oplus (K_3^{(n)} + ID) \\ K_2^{(n+1)} &= K_2^{(n)} \oplus n_2 \oplus (K_4^{(n)} + ID) \\ K_3^{(n+1)} &= (K_3^{(n)} \oplus n_1) + (K_1^{(n)} \oplus ID) \\ K_4^{(n+1)} &= (K_4^{(n)} \oplus n_1) + (K_2^{(n)} \oplus ID) \end{aligned}$$

   Note that ID is unchanged.

In step 3, the role of the messages A, B, and C is as follows. Messages A and C are used to deliver the random numbers n_1 and n_2 to the tag without the adversary determining them. Messages B and D are used to authenticate the reader and tag, respectively. Message E is used to transmit the tag's secret information, ID, to the reader.

The EMAP protocol follows a similar idea. As in MMAP, the tag and reader maintain four shared keys, K_1, K_2, K_3, and K_4, as well as identifier IDP and secret information ID. The steps in EMAP are as follows.

1. The reader sends a *Hello* messgae to the tag, which powers the tag.
2. The tag responds with IDP.
3. Based on the value of IDP, the reader looks up the values of K_1, K_2, K_3, and K_4. The reader generates two random bit strings n_1 and n_2 and sends a message to the tag consisting of three bit strings, $A = IDP \oplus K_1 \oplus n_1$, $B = (IDP \vee K_2) \oplus n_1$, and $C = IDP \oplus K_3 \oplus n_2$.
4. The tag computes $n_1 = A \oplus IDP \oplus K_1$ and $n_2 = C \oplus IDP \oplus K_3$, and checks if $B \stackrel{?}{=} (IDP \vee K_2) \oplus n_1$. If the authentication check is passed, then the tag sends a message to the reader containing the bit strings $D = (IDP \wedge K_4) \oplus n_2$ and $E = (IDP \wedge n_1 \vee n_2) \oplus ID \oplus K_1 \oplus K_2 \oplus K_3 \oplus K_4$.
5. The reader computes ID using the received message E.
6. The tag and reader each update the values of IDP, K_1, K_2, K_3, and K_4 as follows:

$$\begin{aligned} IDP^{(n+1)} &= IDP^{(n)} \oplus n_2^{(n)} \oplus K_1^{(n)} \\ K_1^{(n+1)} &= K_1^{(n)} \oplus n_2^{(n)} \oplus ((ID)_{1:48} || F_p(K_4^{(n)}) || F_p(K_3^{(n)})) \\ K_2^{(n+1)} &= K_2^{(n)} \oplus n_2^{(n)} \oplus (F_p(K_1^{(n)}) || F_p(K_4^{(n)}) || (ID)_{49:96}) \\ K_3^{(n+1)} &= K_3^{(n)} \oplus n_1^{(n)} \oplus ((ID)_{1:48} || F_p(K_4^{(n)}) || F_p(K_2^{(n)})) \\ K_4^{(n+1)} &= K_4^{(n)} \oplus n_1^{(n)} \oplus (F_p(K_3^{(n)}) || F_p(K_1^{(n)}) || (ID)_{49:96}) \end{aligned}$$

The notation F_p is defined as follows. If x is a bit string, where the length of x is a multiple of 4, then $F_p(x)$ is computed by first dividing x into 4-bit blocks. The four bits in each block are then XORed. For example, if $x = 1011\ 0110\ 1000$, then $F_p(x) = 101$. The notation $(ID)_{1:48}$ refers to the 48 most significant bits of ID, while $(ID)_{49:96}$ denotes the 49 least significant bits of ID. As in MMAP, the ID is unchanged.

3 Attacks on MMAP and EMAP

The attacks on each protocol that you will implement are discussed in the references [3, 4]. The goal of both attacks is to determine the secret quantity ID. An example of the attack on MMAP is as follows. When the adversary eavesdrops on a protocol instance, the adversary has access to the messages $B = (IDP \wedge K_2) \vee n_1$ and IDP. By the properties of bit-wise OR and AND, if $(IDP)_i = 0$, then $(B)_i = (IDP \wedge K_2)_i \vee (n_1)_i = (n_1)_i$. Hence any bits of n_1 corresponding to 0 bits of IDP will become known to the adversary.

For example, suppose that the adversary observes that $B = 011000$ and $IDP = 101100$. Based on the above, the adversary has that $n_1 = *1**00$. Using the observed message E, the adversary then determines that ID is given by $ID = (E \oplus n_1) - IDP = *1**00 + 010100 = ****00$. This reveals the two least significant bits of ID; the steps to recovering the remaining bits of E based on further protocol runs are described in [3].

The attack on EMAP is described as follows. Since the message D is given by $D = (IDP \wedge K_4) \oplus n_2$, whenever $(IDP)_i = 0$, $(n_2)_i = D_i$. Similarly, whenever $(IDP)_i = 1$, B_i is given by the complement of $(n_1)_i$. The remaining bits of n_1 and n_2 are obtained by tampering with the messages sent between the reader and tag, as described in Section 3.2 of [4].

4 Python Coding for this Project

You will need to create two **Python** classes, MMAPoracle and EMAPoracle, which simulate the MMAP and EMAP protocols, respectively. MMAPoracle must contain a constructor function, as well as the function

$$[outStruct, oracle] = protocolRun(oracle)$$

which takes as input an MMAPoracle and outputs a structure containing the strings A, B, C, D, and E, as well as the updated oracle. The values in outStruct will be used to mount the attack. Similarly, EMAPoracle must contain a constructor and the function

$$[outStruct, oracle] = protocolRun1(oracle).$$

EMAPoracle also implements a function impersonate_reader, defined by

$$[D, E, oracle] = impersonate\_reader(oracle, A, B, C),$$

which takes as input an oracle and three messages A, B, C given to the tag, and gives as output the tag's response D and E. EMAPoracle simulates an attack in which the adversary sends a set of messages to the adversary in order to observe the response, as in Stage 2 in Section 3.2 of [4].

Lastly, the MMAPoracle and EMAPoracle must include a function verifyID, which returns 1 if the given ID is the true ID of the tag and 0 otherwise. This function can be used to check whether your simulation of the attack is returning the correct ID.

Then your task is to implement the **Python** functions MMAP_attack and EMAP_attack, which take as input an MMAPoracle (for MMAP_attack) or EMAPoracle (for EMAP_attack) and output the ID of the tag. Each function can make queries to the corresponding protocolRun function, as well as the impersonate_reader function in the case of EMAP_attack, as needed in order to implement the attacks. The correctness of the ID returned by MMAP_attack and EMAP_attack can be checked using the corresponding verifyID function.