DTU Technical University of Denmark

02255 Practical cryptology, E19

Homework 1

30th of September 2019

Christina Juulmann
s170735

1 Integral cryptanalysis on AES

In this exercise we are to implemenet the integral attack on a modified AES block cipher of 4 rounds.

1.1 AES-128 implemenation in software

AES block ciphers operates on bytes. An AES-128 block consist of 16 bytes. We represent this as a 16 byte sized vector in memory and operate on it as a 4x4 matrix throughout the encryption. Moreover, unlike conventional matrix calculations operations are done in a 'column-row' like fashion (in oppose to 'row-column').

1.1.1 Modified AES implementation choices

The implementaion makes heavy use of pointers thus directly manipulating the address of interest in oppose to copies. Despite saving memory and operations by using pointers, this implementation is not optimized for performance. Rather, focus has been on understanding the exercises as well as code readability.

The follwoing is an overview of the steps taken in an AES encryption and parameters for this implementation:

Input: plaintext (source), state (target), roundKey, S-box, number of rounds.
Output: ciphertext.

- **AddRoundKey**: save the XOR-sum of roundKey and AES state vector to state; i.e. state = state XOR roundKey.
- **SubBytes**: Mask first four- and last four bits of the current byte from the state vector and use it as the corresponding entry {a,b} to S-box. Substitute the found value to the corresponding byte. Do this for all 16 bytes of the AES state vector.
- **ShiftRows**: In a double loop go through all entries in state vector and operate on it as a 4x4 matrix (in a column/row like fashion). Row i is moved i times to the left.
- **MixColumns** takes the special M-matrix as input and multiply the corresponding byte of state vector with the number specified in M. If `0x01` then the byte value is the same. If `0x02` then we multiply by two in $\texttt{GF}(2^8)$, which in practice is equivalent to one left shift. However, if the most significant bit of the byte is one, this operation will cause an overflow and the value is lost. Thus, in such case we simplify using Rijndaels polynominal (`0x1B`). In summary we left-shift by 1 followed by XORing with 0x1B. Last, if the number specified by `M` is `0x03` we can simplify the calculation to the sum of the current byte and multiplication by two of the current byte (i.e. $\texttt{0x02} \cdot \texttt{state[i]} + \texttt{state[i]}$).

All sub-operations are implemented as independet functions which is later gathered and called in a certain order in the AES function (called `AES_ENC`).

1.2 Integral attack

The following is an overview of the integral attack implementation done.

1. Generate multiple sets of 256 plaintexts where all bytes but one takes on the same value. The one byte that differes takes all values from `0x00` to `0xFF`, thereby creating 256 distinct plaintexts.
2. Encrypt the sets using AES modified to 4 rounds.

3. Now we want to find all the 16 bytes of the last roundKey from which we can derive the masterkey. We find the last roundKey by integrating (XOR-summation) over one byte at a time using the sets of ciphertexts previously generated. We do this by computing backwards in the last special round *(AddRoundKey, (Inverse) ShiftRows, (inverse) SubBytes)* with each value of a roundKey guess (`0x00-0xFF`) for all ciphertext byte values. The values computed for a given roundKey guess with all ciphertext byte values are (XOR-)summed together; if this sum is 0, the roundKey guess is a candidate. A candidate array is kept globally and updated after every roundKey guess has been computed with all the values of the cipertext set. All candidates are initially assumed true and as we intergrate over multiple sets we rule out candidates which sum did not result in 0. This is done for all 16 bytes of the last roundKey, thereby revealing this entire roundKey.