

# Secure Multi-party AES

10th February 2009

## 1 Implementation

AES operates on matrices of bytes. Most of the operations of AES work on  $\text{GF}(2^8)$ , considering every byte as an element of some representation of the field, i.e.  $\text{GF}(2)[X]/(p)$ , the field of polynomials over  $\text{GF}(2)$  modulo some irreducible polynomial  $p$ . The byte  $a = a_7 \dots a_0$  is treated as  $a(x) = \sum_{i=0}^7 a_i x^i$ . We use the same field and representation as a basis for Shamir secret sharings of the bytes.

The internal state of AES is a matrix of  $4 * 4$  bytes, corresponding to a block size of 128. Encryption and decryption are round-based, with each round consisting of some operations on the internal state.

### 1.1 ByteSub

In ByteSub, an S-box is applied to every byte of the input. Because the S-box is defined arithmetically, we can compute it with multi-party computation.

#### 1.1.1 Inversion

The field element represented by the byte is inverted in  $\text{GF}(2^8)$ , except 0, which is mapped to 0. There are two possibilities of doing this with multi-party computation:

- Exponentiate the field element to  $\text{ord}(\text{GF}(2^8)^*) - 1 = 254$  using some square-and-multiply variant. This costs 11 multiplications in 11 rounds per byte, using the addition chain 1, 2, 3, 6, 12, 15, 30, 60, 63, 126, 252, 254. Standard square-and-multiply costs 13 multiplications in 8 rounds.
- Mask the field element  $a \in \text{GF}(2^8)$  by multiplying with some random number  $r \in_R \text{GF}(2^8)$  and open the result. Then invert it and multiply it with the random number to get a sharing of the inverted element:

$$\begin{aligned} [a] \cdot [r] &= [ar] \\ (ar)^{-1} \cdot [r] &= [a^{-1}r^{-1}r] = [a^{-1}]. \end{aligned}$$

To handle the case of  $a = 0$ , we add 1 before doing the above if  $a = 0$ . Let

$$b = \begin{cases} 0, & a \neq 0 \\ 1, & a = 0. \end{cases}$$

$b$  can be computed by decomposing  $a$  into bits  $a_i$ ,  $a = a_7 \dots a_0$ . This costs one open operation if uniformly random bits can be generated locally using e.g. pseudo-random secret sharing. Then the inversion is computed as follows:

$$\begin{aligned} [b] &= 1 - \prod_{i=0}^7 (1 - [a_i]) \\ ([a] + [b]) \cdot [r] &= [(a + b)r] \\ ((a + b)r)^{-1} \cdot [r] &= [(a + b)^{-1}] \\ [(a + b)^{-1}] - [b] &= \begin{cases} [(a + 0)^{-1} - 0] = [a^{-1}], & a \neq 0, b = 0 \\ [(0 + 1)^{-1} - 1] = [0], & a = 0, b = 1. \end{cases} \end{aligned}$$

If  $(a + b)r$  is now 0, we know that  $r = 0$ . In that case, we just choose another random  $r$ .

The computation of  $[b]$  costs 7 multiplications in 3 rounds and the computation of  $(a + b)r$  costs 1 opening, again assuming that the random shared number  $[r]$  can be computed locally. Since  $r$  might be zero, we need expected  $\frac{1}{1 - \frac{1}{256}} = 1 + \frac{1}{255}$  opening operations until  $(a + b)r$  is non-zero. The rest can be computed locally. Together with the opening needed for bit decomposition, we get expected  $9 + \frac{1}{255}$  elementary operations in  $5 + \frac{1}{255}$  expected rounds.

### 1.1.2 Affine linear transformation

Here, the byte is seen as a vector in  $\text{GF}(2)^8$  which is multiplied with a fixed invertible matrix and then added to a constant vector. To do so, we decompose the input into bits (cost: one opening if random secret bits can be generated locally) and then we compute the rest locally because the matrix and the vector are publicly known.

### 1.1.3 Communication cost

The computation of one S-box costs 12 elementary operations in 12 rounds using exponentiation and expected  $10 + \frac{1}{255}$  elementary operations in  $6 + \frac{1}{255}$  rounds using masking.

## 1.2 ShiftRow

In ShiftRow the rows of the state matrix are shifted in some way. The bytes are not changed, so this can be done locally.

### 1.3 MixColumn

In MixColumn the columns of the state matrix are treated as vectors over  $\text{GF}(2^8)$  which are multiplied with a fixed  $4 \times 4$  matrix over  $\text{GF}(2^8)$ . Since this only implies multiplications of shared values with known constants and additions, it can be done locally.

### 1.4 Round key addition

In every round, a round key with the same size as the internal state matrix is added to it. This can be done locally.

### 1.5 Key expansion

Key expansion is the computation of the round keys from the input key. It consists of ByteSub operations, byte-wise shift operations, and additions. The latter two can be done locally.

## 2 Analysis and Benchmarks

Since the S-box is the only part which needs communication, one AES round consists mainly of 16 S-boxes which are computed in parallel. With inversion by exponentiation, one round consists of 176 elementary operations in 12 rounds. Depending on the key size, one block is encrypted in 8, 12, or 14 AES rounds, which sums up 2112 elementary operations in 144 rounds in the case of key size 192 bits.

The number of S-boxes needed for key expansion is 31 (for key size 192 bits), which can be computed in parallel with the AES rounds. Therefore, the encryption of one block requires 2453 elementary operations in 144 rounds.

The implementation of the encryption was tested on a local network with recent machines. Using three machines and passive security against one opponent, the encryption of one block with key size 192 bits took about 5 seconds on average including key expansion. About 2 thereof were needed for local computations. This was achieved using inversion by exponentiation which turned out to be faster than inversion by masking in the given setting. This is contradictory to our analysis. Furthermore, the use of a different addition chain with 9 instead of 11 rounds (1, 2, 4, 8, 9, 16, 25, 50, 54, 100, 200, 254) in the inversion increases the time about 1 second.