Secure Multiparty AES (full paper)

Ivan Damgård and Marcel Keller

Dept. of Computer Science, Aarhus University, Denmark
{ivan,mkeller}@cs.au.dk

Abstract We propose several variants of a secure multiparty computation protocol for AES encryption. The best variant requires $2200 + \frac{400}{255}$ expected elementary operations in expected $70 + \frac{20}{255}$ rounds to encrypt one 128-bit block with a 128-bit key. We implemented the variants using VIFF, a software framework for implementing secure multiparty computation (MPC). Tests with three players (passive security against at most one corrupted player) in a local network showed that one block can be encrypted in 2 seconds. We also argue that this result could be improved by an optimized implementation. The security requirements are the same as for the underlying MPC scheme.

1 Introduction and Motivation

In secure multiparty computation (MPC), a number of players each supply a private input and then compute an agreed function on these inputs *securely*, i.e., even if an adversary corrupts some of the players, honest players obtain correct results, and the intended outputs is the only new information released about the inputs. Several general feasibility results for MPC are known, for instance, given secure point to point channels, any function can be computed securely against an honest but curious adversary corrupting any minority of the players, and securely against a malicious adversary corrupting less than one third of the players [1, 4].

Although MPC has been a topic in cryptographic research for many years, and despite the obvious potential for applications, implementations have evolved only recently [2, 6, 7]. Some of them have even been used to solve real-world tasks, such as privacy-preserving auctions [3].

In this paper, we present several variants of an MPC protocol for computing AES encryption [10]. We assume that key and plaintext are byte-wise secret shared among the players; the same holds for the outputted ciphertext.

Apart from the general motivation of investigating how far we can take MPC in practice, there is also a more direct motivation for looking at such a "threshold-approach" to symmetric encryption. An example: suppose a set of players hold some secret shared data and wish to communicate this data to an external party. A trivial solution is for each player to send his shares securely to the receiver, who can then reconstruct the data. But this will mean that the receiver must be aware of the fact that the data is secret shared and must apply a non-standard algorithm to get the data. In addition, his work is linear in the number of players. From this point of view it would be a more attractive solution if the players could cooperate to generate a ciphertext for the receiver in standard form, which would typically be an encryption of an AES key K under the receivers public key, followed by the data encrypted under K. Note that a similar solution used in the opposite direction could be used for a party to supply encrypted inputs to a multiparty computation, even if that party is not aware of the number of players, or the concrete MPC protocol they execute. He only needs to know a public key for the system, where the players share the private key. This could be useful in any application of MPC, e.g. for secure auctions, procurement or benchmarking. In practice, this would mean that parties submitting data to the system can use completely standard

client software for sending data securely protected under a public key. Moreover, the back-end of the system can be updated with new MPC protocols or migrate a to a new set of players with no change on the client side, as long as the public key remains the same.

Another application could be the following: Analogously to encrypted hard disks, one could imagine to store data encrypted in a place with weak security compliance (e.g., a cloud), whereas the key is secret shared between different secured machines. Those machines then can run multiparty AES to read and write data together with further MPC to process it. The secret sharing of the key reduces the risk of leakage, as well as the risk of loosing the key. A more naive solution, where one reconstructs the key and encrypts/decrypts data in the normal way, would create a single point of attack from where the entire data-set can be stolen even if one only meant to read a small part.

Whereas choosing a random key K and encrypting it under a public key is easy using known techniques, there is virtually no previous work considering specifically MPC for symmetric encryption (except for an existing 2-player solution, see next section).

Our work on AES exploits the fact that AES is based on arithmetic in $\mathrm{GF}(2^8)$. Therefore, our protocol can be based on any general MPC protocol that is based on Shamir secret sharing [11] and implements secure multiplication and addition in $\mathrm{GF}(2^8)$. With respect to security threshold and type of adversary (passive/active), our protocol will be as secure as the underlying MPC protocol we use. We can, for instance, use the classic passively secure protocol from [1] tolerating a dishonest majority, or the actively secure protocol from [6] tolerating less then one third corrupted players.

The non-trivial problem we need solve is to implement the AES S-box efficiently, since this is the only non-linear part of the algorithm, and essentially requires us to securely compute a multiplicative inverse of an element in $\mathrm{GF}(2^8)$ where 0 should be mapped to 0. The naive solution to this is to raise to the power of 254. We propose several alternative solutions that improve on this by reducing the number of elementary operations, or the number of rounds, or both.

We have implemented our protocol in VIFF, a software framework for implementing secure multiparty computation [6]. Tests for three players running a passively secure protocol on a local network show that an AES block can be encrypted in 2 seconds, and tests also confirm that our methods for reducing the number of rounds lead to better performance when network delays are large enough to influence speed. Since our implementation uses a general framework based on the high-level interpreted language Python, much better performance can certainly be obtained using a dedicated C implementation. We therefore believe our results demonstrate that MPC for symmetric encryption is definitely a possibility in practice.

2 Related Work

MPC protocols can be divided into two categories. The first consist of protocols computing an arithmetic circuit over a suitable field. These are usually related to a secret-sharing scheme [11]. Other protocols can be used to compute any binary circuit. These are mostly based on Yao's garbled circuits [12]. An optimized implementation by Pinkas et al. was recently used for secure two-party AES [9]. Their protocol differs from ours, which is of the first category. Their protocol requires one party to know the key, the other to know the cleartext, and outputs the ciphertext to the latter one. Our protocol works for the multiparty case, it takes a secret shared key and cleartext as input and outputs a secret shared ciphertext. The communication complexity of our protocol is smaller, due to the utilization of the arithmetic properties of AES. Our implementation is also faster than that of [9], as detailed later. However, Yao's garbled circuits lead to constant-round protocols, contrary

to ours, in the sense that if one increases the number of AES rounds, our number of rounds increase as well.

Since the original proposal of MPC there have been several improvements to make it more efficient. On of those is pseudorandom secret sharing [5], which allows to generate a secret shared random number without any communication at all. Another improvement is an MPC protocol providing active security which allows preprocessing, i.e., performing some computations without knowing the input to reduce the online time [6]. We will use both techniques in the following.

3 Preliminaries

3.1 The Finite Field $\mathrm{GF}(2^8)$

AES treats bytes mostly as elements in $\mathrm{GF}(2^8)$ because there exists a bijective mapping from the set of bytes to the field:

$$\{0,1\}^8 \to \mathrm{GF}(2^8)$$
$$a = a_7 \dots a_0 \mapsto \sum_{i=0}^{7} a_i \cdot x^i,$$

where $\mathrm{GF}(2^8)$ is represented by $\mathrm{GF}(2)[x]/(p)$, the ring of polynomials over $\mathrm{GF}(2)$ modulo some polynomial p. If the polynomial is irreducible, every element except 0 has a multiplicative inverse, i.e., the ring is also a field. AES uses the polynomial $p(x) := x^8 + x^4 + x^3 + x + 1$. Note that in any such representation, the addition of two elements corresponds to the bitwise XOR of the represented bytes due to the similar property of $\mathrm{GF}(2)$. For the same reason, every element is its own additive inverse, so subtraction equals addition. Algebraically spoken, $\mathrm{GF}(2^8)$ has the characteristic 2.

3.2 Secure Multiparty Computation

Secure multiparty computation (MPC) denotes the distributed computation of an arbitrary function such that any non-qualified set of players learns not more than their inputs and outputs. More formally, every non-qualified set of players can simulate their view of the protocol execution using their inputs and outputs.

In our implementation, we use the MPC scheme based on Shamir secret sharing, which allows to compute any arithmetic circuit on a finite field, i.e., any function built with addition and multiplication. Addition and multiplication with public constants can be done locally; multiplication in general and opening of results requires one value to be sent from every player to every other player. We will refer to the latter two as elementary operations.

In the mentioned MPC scheme any player subset up to size t is considered as non-qualified. The maximal possible size of t depends on the number of players n, the security model, and the computational power of the adversary. For instance, t must be strictly less than $n/2$ for perfect security against a passive, adaptive adversary.

In general, every MPC scheme providing addition and multiplication in $\mathrm{GF}(2^8)$ can be used. However, our analysis is based on the assumption that addition and multiplication with public constant can be done without communication, and that secret shared random bits can be generated without communication (see next section). MPC based on Shamir secret sharing fulfills both properties. Throughout the paper, we will use square brackets to denote secret shared values: $[x]$.

3.3 Pseudorandom Secret Sharing

Our protocol requires the generation of random shared values. This can be done as follows. Every player chooses a value in $\mathrm{GF}(2^8)$ uniformly at random. These values are then added using MPC. If at least one player follows the protocol, the result is distributed uniformly at random, and the view of the other $n-1$ players is independent of the result. Hence, the protocol is as secure as the MPC scheme used.

Compared to the just mentioned protocol, pseudorandom secret sharing (PRSS) is more efficient because it can be done without communication, at the cost of losing perfect security. It works as follows: Every minimal qualified set of players knows a secret key for a pseudorandom function used to compute a random value on some public input. The random shared value then is the sum of all those values. By choosing appropriate polynomials for Shamir secret sharing, every player can compute his share of the sum knowing only the summands he knows, i.e., those which are generated using the secret of the player sets he belongs to. A disadvantage of PRSS is that computation complexity increases exponentially with the number of players, i.e. with $O\left(\binom{n}{t+1}\right)$ for threshold t.

Moreover, we require the generation of random shared bits. This is straight-forward in the field $\mathrm{GF}(2^8)$. Since it has characteristic 2, $1+1=0$ and therefore the sum of values in $\{0,1\}$ is in the same set. So we can use the same technique with a pseudorandom function outputting bits instead of values in $\mathrm{GF}(2^8)$. This allows us also to generate a random shared value which is also shared bit-wise with no communication because multiplication with constants (multiples of x in this case) can be done locally.

3.4 Bit Decomposition

At some points, AES operates on bytes bitwise instead of treating bytes as elements of $\mathrm{GF}(2^8)$. Therefore, our protocol also requires the computation of shares $[a_i]_{,i=0,\dots,7}$ from a secret shared value $[a]$, where $[a_i]$ are the secret shared bits of the secret shared byte $[a]$. In fields with characteristic 2 this is straight-forward. First we generate eight secret shared random bits $[r_i]$, and mask $[a]$ with the bit-wise shared random value $[r]$:

$$[r] := \sum_{i=0}^{7} [r_i] \cdot x^i$$
$$[c] := [a] + [r].$$

c is then opened and decomposed into bits c_i:

$$\sum_{i=0}^{7} c_i \cdot x^i = c.$$

In the end, we compute

$$[a_i] := c_i + [r_i].$$

This gives the desired result because

$$\sum_{i=0}^{7} a_i \cdot x^i = \sum_{i=0}^{7} (c_i + r_i) \cdot x^i = c + r = a + r + r = a.$$

The last equality holds because $\mathrm{GF}(2^8)$ has characteristic 2. If the bits $[r_i]$ are generated using pseudorandom secret sharing, the communication cost is one opening operation.

4 The AES Protocol

AES encryption and decryption are round-based, with each round consisting of some operations on the internal state. This is a matrix of 4×4 bytes, corresponding to a block size of 128 bits. Initial state is the input, final state the output. A typical encryption round looks as follows:

- SubBytes
- ShiftRows
- MixColumns
- AddRoundKey

The only exceptions are an additional AddRoundKey at the beginning of encryption, and that MixColumns is skipped in the last round.

We now describe how to compute all operations using MPC. Both cleartext and key are assumed to be byte-wise secret shared over $\mathrm{GF}\left(2^8\right)$. The internal state and so the output will be as well.

4.1 SubBytes

In SubBytes, an S-box is applied to every byte of the input. Because the S-box is defined arithmetically, we can compute it relatively efficiently with multiparty computation. This is the only part of the protocol requiring communication, everything else can be done locally. The S-box consists of two steps: an inversion on $\mathrm{GF}\left(2^8\right)$ and an affine linear transformation on $\mathrm{GF}(2)^8$.

$$\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} \mapsto \begin{pmatrix} \text{S-box}(a) & \text{S-box}(b) & \text{S-box}(c) & \text{S-box}(d) \\ \text{S-box}(e) & \text{S-box}(f) & \text{S-box}(g) & \text{S-box}(h) \\ \text{S-box}(i) & \text{S-box}(j) & \text{S-box}(k) & \text{S-box}(l) \\ \text{S-box}(m) & \text{S-box}(n) & \text{S-box}(o) & \text{S-box}(p) \end{pmatrix}$$

Inversion The field element represented by the byte is inverted in $\mathrm{GF}\left(2^8\right)$, except 0, which is mapped to 0. There are several possibilities of doing this with multiparty computation.

Square-and-multiply We raise the field element to the power of $\mathrm{ord}\left(\mathrm{GF}\left(2^8\right)^*\right) - 1 = 254$ using some square-and-multiply variant. This costs 11 multiplications in 9 rounds per byte, using the addition chain (1, 2, 4, 8, 9, 18, 19, 36, 55, 72, 127, 254). This is optimal regarding the number of multiplications. Since the number of rounds is the lowest possible for the number of multiplications, we will refer to this variant as *square-and-multiply with shortest addition chain and least number of rounds*. Another multiplication chain, (1, 2, 3, 4, 7, 8, 15, 16, 31, 32, 63, 64, 127, 254) requires 13 multiplications in 8 rounds, which is optimal regarding the number of rounds. We will refer to this as *square-and-multiply with least rounds*. Figure 1 shows the two additions chains. Standard square-and-multiply costs 13 multiplications in 13 rounds.

Masked Exponentiation This method uses the fact that

$$(x+y)^2 = x^2 + 2xy + y^2 = x^2 + y^2$$

for fields with characteristic 2. By a simple induction, it follows that

$$(x+y)^{2^i} = \left((x+y)^{2^{i-1}}\right)^2 = \left(x^{2^{i-1}} + x^{2^{i-1}}\right)^2 = x^{2^i} + y^{2^i}$$

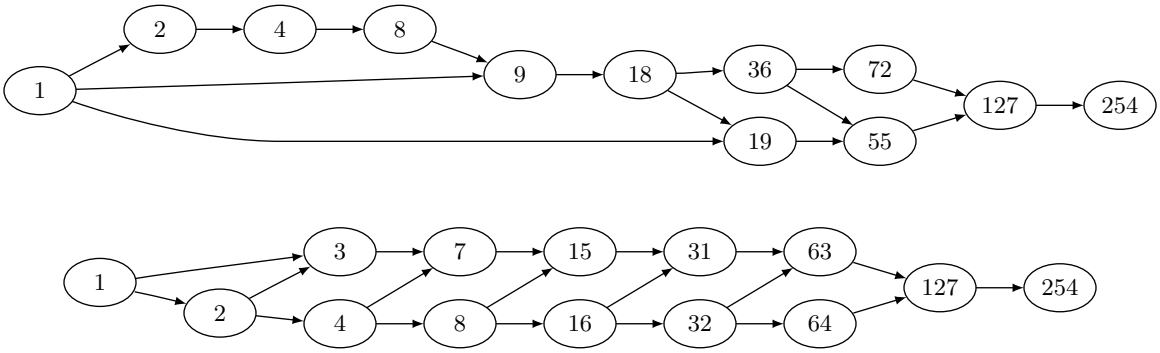

Figure 1. Two addition chains to compute the inverse in $\mathrm{GF}(2^8)$ with square-and-multiply. The first one is optimal with respect to the number of multiplications. It was chosen among many possibilities because the average lifetime (number of rounds between creation and last usage) of the elements was the lowest. The seconds is one of two addition chains with the lowest possible number of rounds, but more multiplications. The number of multiplications corresponds to the number of nodes minus 1, the number of rounds to the lenght of the graph.

for $i \geq 2$. We exploit this property to split up the computation in a preprocessing and an online phase, which saves some rounds because the preprocessing operations for all S-boxes of the protocol can be executed in parallel.

In the preprocessing phase, we generate a random shared $[r] \in \mathrm{GF}(2^8)$ and square $[r]$ 7 times. This costs 7 multiplications in 7 rounds if pseudorandom secret sharing is used:

$$\begin{aligned} \left[r^2\right] &= [r] \cdot [r] \\ \left[r^4\right] &= \left[r^2\right] \cdot \left[r^2\right] \\ &\dots \\ \left[r^{128}\right] &= \left[r^{64}\right] \cdot \left[r^{64}\right]. \end{aligned}$$

To invert $[x]$, we mask $[x]$ by $[r]$, open it, and exponentiate locally:

$$\begin{gathered} \mathrm{open}\left([x] + [r]\right) = (x + r) \\ (x+r)^2, (x+r)^4, (x+r)^8, \dots, (x+r)^{128}. \end{gathered}$$

Finally, we compute the shared binary powers of $[x]$ and multiply them to get $\left[x^{254}\right]$:

$$\begin{gathered} (x+r)^{2^i} + \left[r^{2^i}\right] = \left[x^{2^i} + r^{2^i} + r^{2^i}\right] = \left[x^{2^i}\right] \ \forall\, i = 1, \dots, 7 \\ \prod_{i=1}^{7} \left[x^{2^i}\right] = \left[x^{\sum_{i=1}^{7} 2^i}\right] = \left[x^{254}\right]. \end{gathered}$$

The online operations cost 7 elementary operations (1 opening and 6 multiplications) in 4 rounds.

Masking Here we exploit that the inversion is a homomorphism with respect to multiplication. The field element $[a] \in \mathrm{GF}(2^8)$ can be masked by multiplying with some shared random number $[r] \in_R \mathrm{GF}(2^8)$. We open the masked value, invert it and multiply the result with the random

number to get a sharing of the inverted element:

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

This method would leak whether a is 0 because $ar = 0$ for all $r \in \mathrm{GF}\left(2^8\right)$ if $a = 0$. Therefore, if $a = 0$, we add 1 before doing the inversion, and subtract 1 after it. This guarantees that 0 is mapped to 0 without leaking it. Let

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

b can be computed by decomposing a into bits a_i, $a = a_7 \ldots a_0$ and then ANDing them together. Altogether, the inversion is computed as follows:

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

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

The computation of $[b]$ costs 1 opening operation for bit decomposition, and 7 multiplications in 3 rounds afterwards. The computation of $(a+b)r$ costs 1 multiplication and 1 opening, again assuming that the random shared number $[r]$ can be generated locally using PRSS. Since r might be zero, we require expected $\frac{2}{1-1/256} = 2 + \frac{2}{255}$ elementary operations until $(a+b)r$ is non-zero. The rest can be computed locally, so we get expected $10 + \frac{2}{255}$ elementary operations in $6 + \frac{2}{255}$ expected rounds overall.

Affine Linear Transformation Here, the byte $a = \sum_{i=0}^{7} a_i \cdot x^i$ is considered as a bit vector $(a_0, \ldots, a_7) \in \mathrm{GF}(2)^8$, 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: 1 opening if PRSS is used), and then we compute the rest locally because the matrix and the vector are fixed.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

Communication Cost The computation of one S-box costs at least 12 elementary operations in 10 rounds or 14 elementary operations in 9 rounds using square-and-multiply and expected $11 + \frac{2}{255}$ elementary operations in $7 + \frac{2}{255}$ rounds using masking. For masked exponentiation, preprocessing requires 7 elementary operations in 7 rounds, and online computation requires 8 elementary operations in 5 rounds, both per S-box.

4.2 ShiftRows

In ShiftRows the rows of the state matrix are shifted in the following way. The bytes are not changed, so this can be done locally because the state is shared byte-wise.

$$\begin{pmatrix} \mathbf{a} & b & c & d \\ \mathbf{e} & f & g & h \\ \mathbf{i} & j & k & l \\ \mathbf{m} & n & o & p \end{pmatrix} \begin{matrix} \text{no shift} \\ \text{shift 1 to left} \\ \text{shift 2 to left} \\ \text{shift 3 to left} \end{matrix} \begin{pmatrix} \mathbf{a} & b & c & d \\ f & g & h & \mathbf{e} \\ k & l & \mathbf{i} & j \\ p & \mathbf{m} & n & o \end{pmatrix}$$

4.3 MixColumns

In MixColumns the state matrix is multiplied with a fixed invertible 4×4 matrix over $\mathrm{GF}(2^8)$. Since this only implies multiplications of shared values with known constants and additions, it can be done locally in the MPC scheme based on Shamir secret sharing. We use a hexadecimal notation of elements in $\mathrm{GF}(2^8)$ here.

$$\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} \mapsto \begin{pmatrix} \mathtt{02} & \mathtt{03} & \mathtt{01} & \mathtt{01} \\ \mathtt{01} & \mathtt{02} & \mathtt{03} & \mathtt{01} \\ \mathtt{01} & \mathtt{01} & \mathtt{02} & \mathtt{03} \\ \mathtt{03} & \mathtt{01} & \mathtt{01} & \mathtt{02} \end{pmatrix} \cdot \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix}$$

4.4 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 with the standard MPC scheme.

$$\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} \mapsto \begin{pmatrix} a + k_{0,0} & b + k_{0,1} & c + k_{0,2} & d + k_{0,3} \\ e + k_{1,0} & f + k_{1,1} & g + k_{1,2} & h + k_{1,3} \\ i + k_{2,0} & j + k_{2,1} & k + k_{2,2} & l + k_{2,3} \\ m + k_{3,0} & n + k_{3,1} & o + k_{3,2} & p + k_{3,3} \end{pmatrix}$$

4.5 Key Expansion

Key expansion is the computation of the round keys from the input key. It consists of the S-box also used in SubBytes and additions. The following figure illustrates one key expansion round of a 128 byte key.

$$\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} \mapsto \begin{pmatrix} a' = \text{S-box}(h) + a + x^r & b' = a' + b & c' = b' + c & d' = c' + d \\ e' = \text{S-box}(l) + e & f' = e' + f & g' = f' + g & h' = g' + h \\ i' = \text{S-box}(p) + i & j' = i' + j & k' = j' + k & l' = k' + l \\ m' = \text{S-box}(d) + m & n' = m' + n & o' = m' + o & p' = o' + p \end{pmatrix}$$

This operation is started with the input key and is repeated every time a new round key is needed. r is a parameter set to 0 the first time, and then increased by 1 for every operation. Section 4.1 shows how to implement the S-box with MPC, and r is public. Therefore, also key expansion can be implemented with MPC.

Key expansion for 192 byte keys works similar, with the difference that the key is a 4×6 matrix. Thus, two key expansion operations produce enough key material for three rounds. For 256 byte keys the algorithm is slightly different, but still feasible with MPC.

5 Security

The security of our protocol relies mainly on the security of the MPC scheme used. The only information that is revealed additionally to the leakage of the MPC scheme are openings of masked values in the bit decomposition and the inversion by masking. For bit decomposition, $c := a + r$ is revealed, where r is a secret shared random value. Therefore, c is distributed uniformly at random as well and independent of a. The same argument holds for the masked values revealed in masked exponentiation. Pseudorandom secret sharing additionally relies on the security of the pseudorandom function used.

For inversion by masking, $(a + b) \cdot r$ is revealed, where $(a + b)$ is a non-zero element of $\mathrm{GF}\left(2^8\right)$, and r is a secret shared value. As every non-zero element of a field has a multiplicative inverse, $(a + b) \cdot r$ is also distributed uniformly at random.

It follows that a simulator, e.g., in the UC framework, can generate those values with the same distribution as in the real execution if there exists a simulator for the MPC scheme.

6 Analysis

Since the S-box is the only part which needs communication, it suffices to count the number of S-boxes computed. For the number of rounds, one has to consider that all S-boxes of one AES round can be computed in parallel. The same holds for the key expansion, which can be computed in parallel with the AES rounds. Putting all together, the total number of elementary operations is the number of S-boxes times the number of elementary operations per S-box, and the the total number of rounds is the number of AES rounds times the number of rounds per S-box.

We now give the analysis of AES-128 using square-and-multiply with shortest addition chain. SubBytes is called once per AES round and computes one S-box for every byte of the state matrix, i.e., 16 S-boxes. There are 10 AES rounds in AES-128, this gives 160 S-boxes or 1920 elementary operations in 100 rounds for one block without key expansion. Key expansion in AES-128 consists of 40 S-boxes or 480 elementary operations. Therefore, the encryption of one block including key expansion requires 2400 elementary operations in 100 rounds.

In the same way, one can calculate that AES-128 using inversion by masking takes $1760 + \frac{320}{255}$ or $2200 + \frac{400}{255}$ expected elementary operations in $70 + \frac{20}{255}$ expected rounds.

Masked exponentiation only gives an advantage over the other methods if all the preprocessing is done before the encryption of a block. In this way, one can calculate with only 7 / (number of AES rounds) rounds per S-box, i.e., 0.7 rounds in the case of AES-128.

Table 1 describes the different key lengths used in AES. Table 2 describes the different possibilities for inversion. We investigated more than the two optimal possibilities of inversion by exponentiation to get more information about the influence of the two parameters on the results.

Key length	AES rounds	S-boxes		
		Key expansion	Block without k. e.	Block with k. e.
128	10	40	160	200
192	12	32	192	224
256	14	52	224	276

Table 1. Overview over the different key lengths in AES.

	Elementary operations	Rounds
Masking	$11 + \frac{2}{255}$	$7 + \frac{2}{255}$
Standard square-and-multiply	14	14
S-a-m with shortest addition chain	12	12
S-a-m with shortest addition chain and least rounds	12	10
S-a-m with least rounds	14	9
Masked exponentiation	15	$5 + \frac{7}{\#\text{AES rounds}}$

Table 2. One S-box in different inversion protocols.

7 Implementation

Our implementation is based on the Virtual Ideal Functionality Framework (VIFF), a Python-based framework for secure multiparty computation. VIFF was developed to implement efficient MPC for asynchronous networks, i.e., every local computation is executed as soon as the needed input values are present. It provides protocols with passive security as well as protocols secure against active adversaries. Shamir secret sharing is used for protocols with at least three parties, and two-party MPC can be done based on the Paillier cryptosystem.

7.1 Benchmarks

The implementation of the encryption was tested on a local gigabit-network (ping 0.1 ms) with modern hardware: Dual-Core AMD Opteron Processor with 2.4 GHz per core, 2 GB RAM, Red Hat 5.2, Linux Kernel 2.6.18, Python 2.6.1. Using three machines and passive security against one opponent, the encryption of one block with AES-128 took about 2 seconds on average including key expansion when encrypting 10 blocks in parallel (see Figure 2). 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. The reason is that masking needs more local computation (more pseudorandom secret sharing used for bit decomposition), which has a higher impact if the network latency is low and the bandwidth is high. This can be verified by changing those parameters. Figure 2 also shows that the number of multiplications has more influence than the number of rounds. All figures are based on the average time of 100 executions for every data point.

Our method is faster than two-party AES by Pinkas et al. [9] which takes 7 seconds with passive security. Note that their implementation is optimized, whereas ours uses Python, a high-level interpreted language. We observed that local computation is the main bottleneck in our implementation, so this gives the possibility for better results using an implementation in a low-level language with less overhead, such as C. The transmitted data in our protocol ranges from

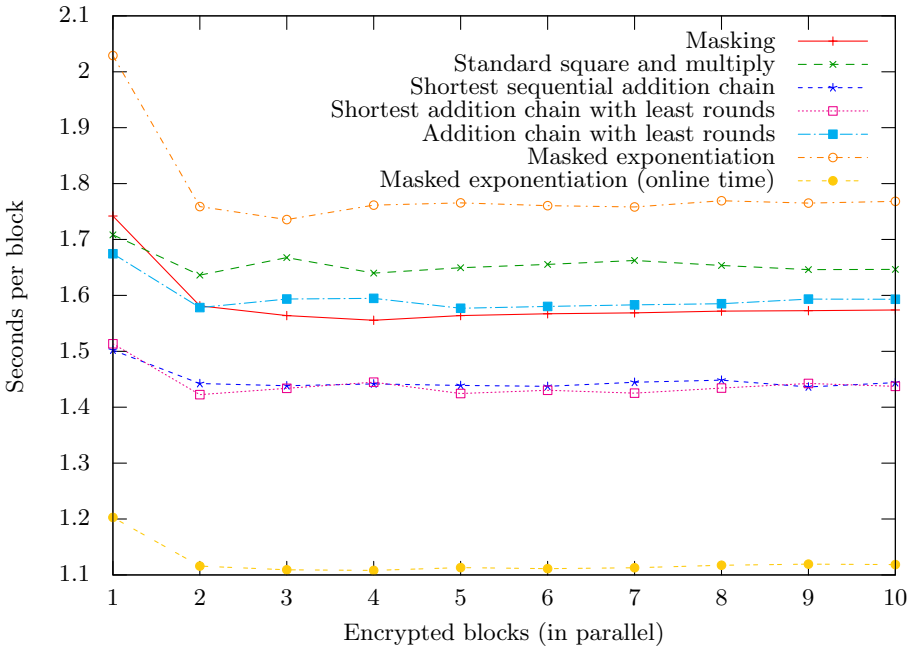

Figure 2. Encryption of up to 10 blocks in parallel.

104 kilobytes to 138 kilobytes for one block including input sharing and output reconstruction; the solution by Pinkas et al. requires 503 kilobytes.

Moreover, in a setting with four players and active security against one malicious adversary our protocol takes about 7 seconds. This is considerably less than the solution by Pinkas et al. which requires 1148 seconds to encrypt one block with security against a malicious adversary. We used the PRSS-based variant of an actively secure MPC scheme by Damgård et al. [6]. The scheme allows to generate so-called multiplication triples in a preprocessing phase, i.e., before knowing any input. By using that, the online time can be reduced to less than 4 seconds per block for masked exponentiation, which uses preprocessing also for AES inversion, as described in Section 4.1. Figure 3 also shows that there is little difference between variants performing the same number of elementary operations. We suspect that the increase with more blocks in parallel is caused by the increase in local computation time due to preprocessing.

With an increased network delay, one expects the number of rounds taken to get more influental. Figure 4 shows the time taken to encrypt one block at different network delays. If the delay is high enough, encryption time behaves as expected: the higher the number of rounds required for an S-box (see Table 2), the higher the time taken. Note that this times were achieved by disabling Nagle's algorithm [8], i.e., every value is sent immediately over the network when it's available, without combining several messages into one TCP packet to better utilize the bandwidth.

We also investigated the behavior under bandwidth limitation. The amount of data sent over the network is determined by the number of elementary operations. Therefore, the time to encrypt one block should be determined by the number of elementary operations per S-box. Figure 5 shows that this is the case with a low enough bandwidth. Moreover, the number of rounds does not make

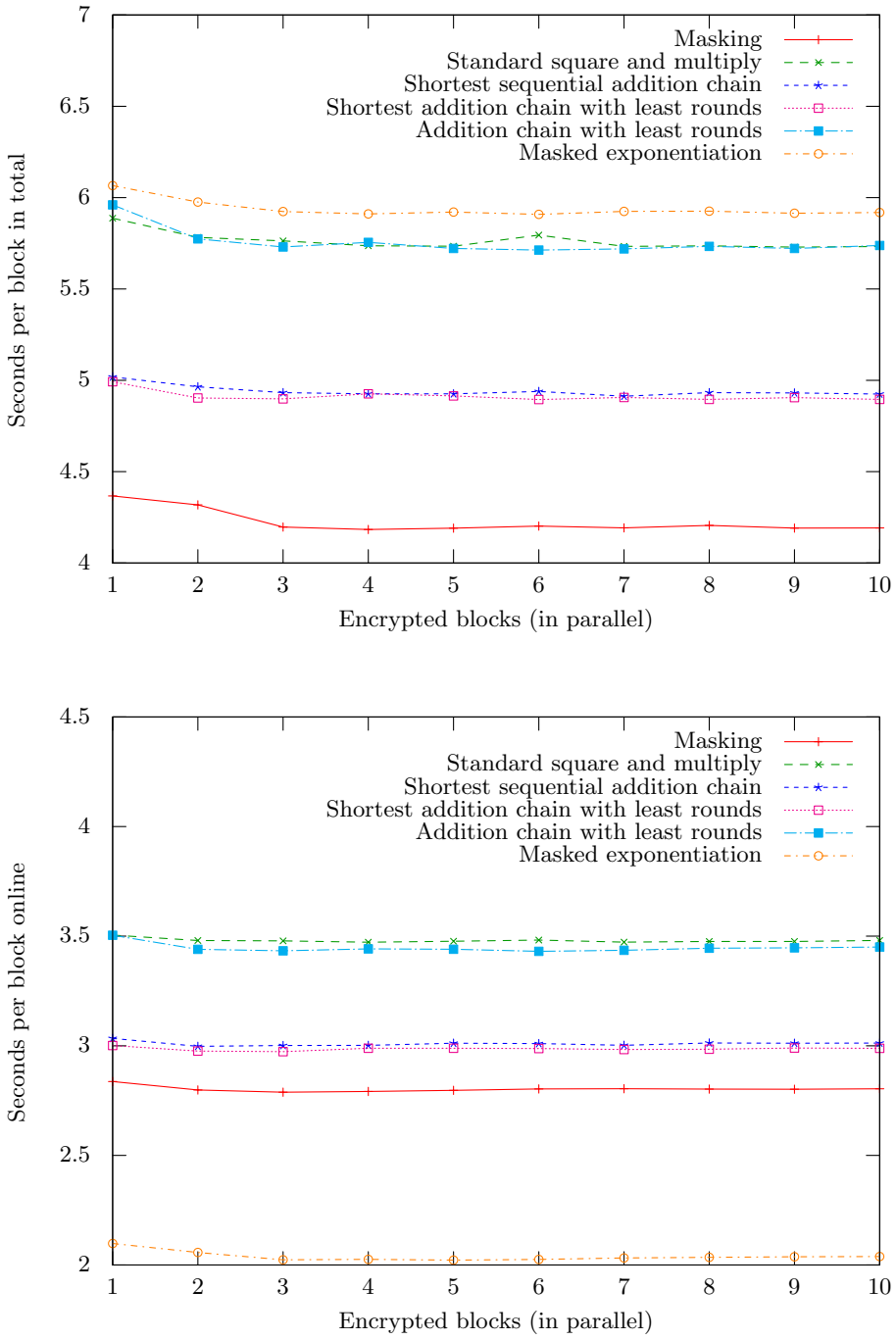

Figure 3. Encryption of up to 10 blocks in parallel with active security, total time and online time.

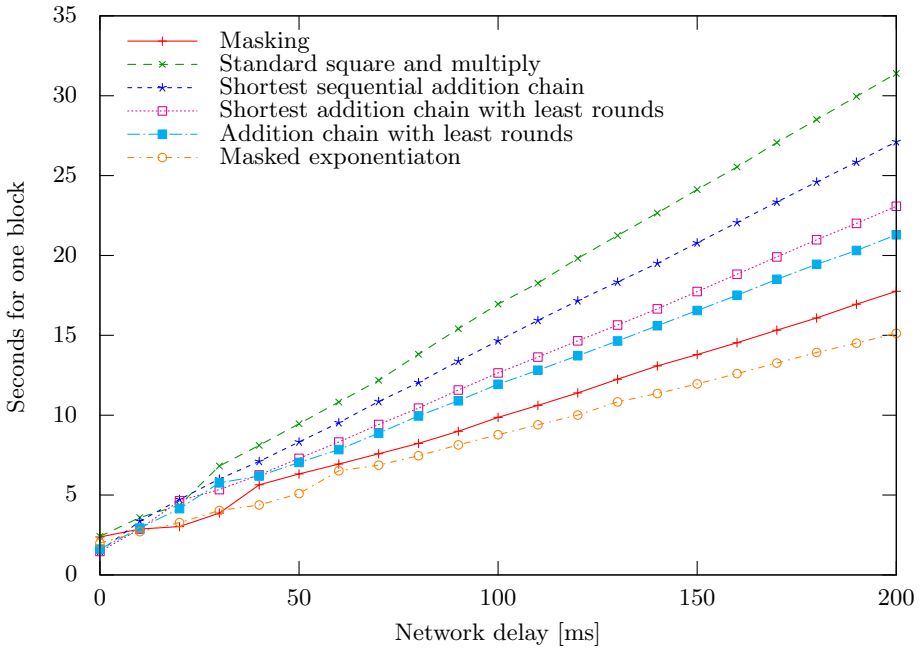

Figure 4. Encryption of one block with increased delay.

a difference, as the two square-and-multiply variants with the shortest addition chain show. Both require the same number of multiplications, but one of them takes two rounds more per S-box.

8 Conclusion

We have presented a secure multiparty computation protocol for AES together with benchmarking results of an implementation: roughly 2 seconds per block. Since the implementation is based on a high-level interpreted language, processing time was more limiting than communication. This caused the benchmarks to contradict the analysis, which in turn could be verified under increased network latency or decreased bandwidth. Our results can not be applied directly to other algorithms (including ciphers) because we made use of the arithmetic properties of AES, namely of the fact that the S-box is not just a "random" substitution.

References

1. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, pages 1–10. ACM, 1988.
2. Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.
3. Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *Financial Cryptography*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2009.

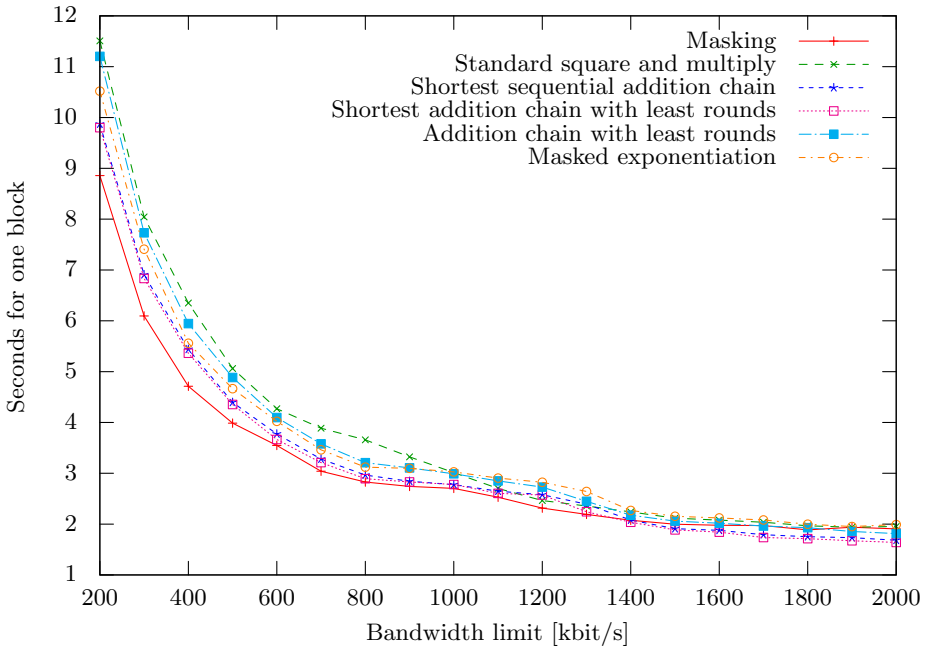

Figure 5. Encryption of one block with limited bandwidth.

4. David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *STOC*, pages 11–19. ACM, 1988.
5. Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *TCC*, volume 3378 of *Lecture Notes in Computer Science*, pages 342–362. Springer, 2005.
6. Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179. Springer, 2009.
7. Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In *USENIX Security Symposium*, pages 287–302. USENIX, 2004.
8. J. Nagle. Congestion control in IP/TCP internetworks. RFC 896, January 1984.
9. B. Pinkas, T. Schneider, N.P. Smart, and S. Williams. Secure two-party computation is practical. Cryptology ePrint Archive, Report 2009/314, 2009. http://eprint.iacr.org/.
10. Federal Information Processing Standards Publications. Advanced Encryption Standard. Technical Report FIPS PUB 197, National Institute of Standards and Technology, November 2001.
11. Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.
12. Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167. IEEE, 1986.