Fast Oblivious AES A Dedicated Application of the MiniMac Protocol

Ivan Damgård and Rasmus Zakarias(✉)

Department of Computer Science, Aarhus University, Aarhus, Denmark
{ivan,rwl}@cs.au.dk

Abstract. We present actively secure multi-party computation of the Advanced Encryption Standard (AES). To the best of our knowledge it is the fastest of its kind to date. We start from an efficient actively secure evaluation of general binary circuits that was implemented by the authors of [DLT14]. They presented an optimized implementation of the so-called MiniMac protocol [DZ13] that runs in the pre-processing model, and applied this to a binary AES circuit. In this paper we describe how to dedicate the pre-processing to the structure of AES, which improves significantly the throughput and latency of previous actively secure implementations. We get a latency of about 6 ms and amortised time about 0.4 ms per AES block, which seems completely adequate for practical applications such as verification of 1-time passwords.

Keywords: Multiparty computation · Arithmetic black box · Arithmetic circuit · Binary circuit · AES

1 Introduction

Secure Multi-party computation (MPC) allows a set of players (or computers) with *private inputs* to evaluate a function on these inputs. Security means that all players learns the output of the function and essentially nothing else. More precisely the problem of MPC for n players is to compute a function $f(x_1, ..., x_n) = (y_1, ..., y_n)$ such that Player, P_i, learns only y_i after evaluating f and x_i is the private input held by P_i. This problem was first proposed by Yao in [Yao82, Yao86] and has been an active area of research since.

The description of the function f can take different forms. In this work we consider descriptions of f as a circuit over the (AND, XOR) or (MUL, ADD) basis for binary and arithmetic circuits respectively. Protocols for evaluating such function typically implement an ideal functionality sometimes called an Arithmetic black-box [DN03]. The Arithmetic black-box is depicted in Fig. 1.

R. Zakarias—The authors acknowledge support from the Danish National Research Foundation and The National Science Foundation of China (under the grant 61061130540) for the Sino-Danish Center for the Theory of Interactive Computation, within part of this work was performed; and from the CFEM research center, supported by the Danish Strategic Research Council.

D. Pointcheval et al. (Eds.): AFRICACRYPT 2016, LNCS 9646, pp. 245–264, 2016.
DOI: 10.1007/978-3-319-31517-1_13

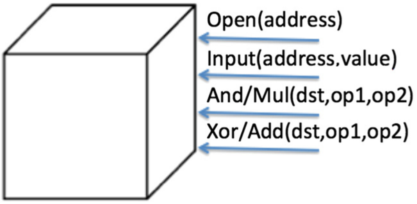


Fig. 1. The Arithmetic Black Box.

Intuitively, the players agree on a circuit over the actions *Open, Input, Xor/Add* and *And/Mul*. Players provide input values using the *Input* command, and then work their way through the circuit invoking the appropriate command for each gate. Finally the *Open* command is used to obtain the result.

MPC for the case where a majority of the players are corrupt requires public-key machinery and was therefore for a long time thought to be impractical, especially for the case of active security. To resolve this, the so-called pre-processing-model was proposed, where the heavy computations are pushed to a pre-processing phase. Using precomputed material one can evaluate the function securely much faster in the on-line phase. With recent result [NNOB12, DKL+13, DLT14, FJN14] in particular, practicality is within our reach, see [IKM+13] for an in depth discussion on the power of correlated randomness. We consider here the particular protocol nick-named *MiniMac* from [DZ13]. This is an arithmetic black-box protocol in the pre-processing-model, which was designed to handle arithmetic circuits over small fields efficiently. To do this, it computes on vectors of field elements instead of single values.

Benchmarking. Oblivious AES is a much used example of how practical MPC is becoming, see [PSSW09] and [NNOB12, DKL+12, GHS12, DLT14, HKS+10, HEKM11]. Oblivious AES distinguishes itself from the classical AES encryption by being distributed between two or more parties. All players know the plaintext and everybody learns the cipher-text. The key, however, is additively shared meaning that the key k is not known to any player.

Performance wise, previous state of the art for Oblivious AES with malicious security is the implementation using a binary circuit in [DLT14] where they report on amortized running times of less than 4 milliseconds per AES-block and 3–4 seconds of latency on ordinary consumer hardware[1]. In [KSS13] a different implementation was reported that uses the algebraic description of AES over $\mathbb{F}_{2^8}$. They achieve about 1 ms amortized time per AES block and a latency of 100 ms.

Our contributions. We show that both amortized time and latency can be improved significantly: in the fastest configuration, we obtain about 0.4 ms amortized time and a latency of about 6 ms. We present three constructions which are variations on the idea that if we exploit the special structure of AES, rather that seeing it as a general binary or arithmetic circuit, we can tailor the

[1] The concrete specifications of our experimental setup can be found in Appendix A.

No AES blocks	**Time/AES** *ms*	**Latency** *ms*	**Prep. size** *MB/player*
MiniMac [DLT14]			
680	4	9962	130, 0.2/AES
Protocol 1 incl. Key Expansion			
5	3	15	270, 54/AES
Protocol 1 without Key Expansion			
5	1.2	6	220, 44/AES
Estimated time, Protocol 2 (no on-line multiplications)			
15	0.4	6	650, 44/AES
Estimated time, Protocol 3 (minimized pre-processing)			
15	0.8	12	10.5, 0.7/AES

Fig. 2. Execution times of our AES protocol.

pre-processing such that we save on the number of rounds and also on local computation. We try out these ideas in practice using the implementation of MiniMac from [DLT14][2] as our starting point.

The basic approach in our first protocol is to pre-process a number of tables, each of which implement an AES S-box followed by the Mix-Column and Shift-Row operations applied to the bits output from the S-box in question. We first describe what the correlated randomness should look like to implement the tables, and then we present a slightly modified version of the MiniMac protocol using this material to perform AES. This solution computes 5 simultaneous AES blocks in only 10 rounds of online communication. In comparison [DLT14] required at least 6800 rounds.

For second protocol, we observe that after we introduce the tables, we no longer need to do secure multiplication in the on-line phase. This allows us to change the internal representation used in MiniMac to allow more parallelism at no extra cost. This immediately saves us a factor of about 3 in amortized time per AES instance.

For the third protocol we give a new construction that shows how to obtain much smaller pre-processing material. We save a factor of at least 60 in the size of pre-processed data at the cost of doing 1 extra round of communication and more local computation in the final protocol. Some explanation of the idea behind this optimization is in order as the idea may be of interest beyond oblivious AES: the efficiency of MiniMac is based on the idea of computing on vectors of values in a SIMD fashion, i.e. we add and multiply vectors coordinate-wise. Concretely, the implementation of MiniMac we started from uses vectors containing 85 data bytes. Now, the reason why it makes sense to compute the AES S-Box by table look-up is that the input is only 1 byte, so we need only 256 entries in the table. However, the result we get will be just one byte, and this result needs to go to the right place in the vector representing the state after the table look-up, of course without revealing what was output from the table. The simplest solution

[2] Available at http://tinyurl.com/q2dmcuw.

is to make the table entry be an entire MiniMac word containing data that only depends on the single byte that is output from that table entry. This will work, as we explain in more detail later, but of course means that tables get very large. What we do for protocol 3 is to show that with an appropriate combination of masking by random values and unconditional MACs, we can have table entries that only consist of a single data byte and some authentication information. This idea can be applied to computation of any function with small input and output, possibly followed by some linear function.

We implemented the Protocol 1 and based on this we calculated the size of pre-processed data for the other results and conservatively estimated their running times, as detailed in the following sections. A summary of this can be seen in Fig. 2.

The demands we have to the pre-processed data are quite specialized and it may not be clear how we can do the required pre-processing reasonably efficiently. In particular, the pre-processing assumed by the original MiniMac protocol does not produce data of the form we need. In principle, the problem can be solved by writing down an arithmetic circuit that takes some random bits as input and outputs the data players need; and then evaluate that circuit using the original MiniMac protocol. This would be an extremely large circuit, and therefore, in the final section, we give a recipe for how pre-processed material may be constructed more efficiently from a generic MiniMac instance.

A main take-home message from our paper is that the only structure we need from AES to speed up the computation is that its non-linear parts consist only of S-Boxes with small inputs, this is what allows us to use table look-up with tables of feasible size. In future work, it will therefore be interesting to investigate if secure computation of other ciphers or hash functions can be made practical using a similar approach.

The MiniMac protocol

This protocol is in a nutshell a SIMD Arithmetic black-box. That is, the protocol operates on a so called representation consisting of a vector of field elements. The actions of the Arithmetic black-box operates in parallel on all elements of the vector simultaneously. The details and proof of security can be found in [DZ13] while the extension for operating efficiently on binary fields was discovered in [DLT14]. For our purposes here we will think of MiniMac simply as an implementation of a SIMD Arithmetic Black-box and hence describe MiniMac's representation of data as containing an l-vector over a finite field[3].

$$[\![\mathbf{a}]\!] = [\![(a_1, ..., a_l)]\!]$$

The operations Input, Add, Mul and Open for the Arithmetic black-box operate on such vectors. E.g. adding two elements in the box with MiniMac yields the computation:

$$[\![\mathbf{a}]\!] + [\![\mathbf{b}]\!] = [\![(a_1 + b_1, ..., a_l + b_l)]\!] = [\![\mathbf{a} + \mathbf{b}]\!]$$

[3] Actually, the players in MiniMac have additive shares of the vectors and a special type of MACs are used to prevent cheating, but these details are not important here.

In a similar way Input requires the secret values to be loaded into the box to be l-vectors and Open gives the parties an l-vector back.

Advanced Encryption Standard

AES is described in [DR00]. Here we consider only 128-bit 10 round encryption with AES. We give a different description of AES in terms of matrix operations rather then the algorithmic approach in [DR00]. We do this for two reasons. Firstly, our implementation framework is geared towards matrix operations thus should the reader be interested in looking at our C code this section is extremely useful for understanding our code design choices. Secondly, this interpretation makes it completely natural and straight-forward to describe AES concisely as a series of matrix products interleaved with table lookups. In particular this make the description of our results in Sects. 2, 3, 4 and 5 easier to describe. The algorithm can be considered to have two distinct phases: The *Key Expansion* and the 10 *Rounds*. The Key Expansion operates on a 16 byte state of key material and the 10 rounds operate on a 16 byte state of plain/cipher-text.

The key expansion in more details operates on a 16 byte state of key material initially containing the encryption key. This state is updated in each round to contain the corresponding round-key.

In [DR00] the key expansion algorithm is explained in an algorithmic way over bytes. Our framework of implementation is geared for matrix operations thus we here give an alternative characterization of the key schedule in terms of matrices. Here we abstract the S-Box operations to merely a table lookup and explain later how such lookups can be done securely with MiniMac. Let $K_0 = (k_0, ..., k_{15})$ be the initial 16 bytes of encryption key. The key is divided in to 4 so called words w_0 through w_3 in the natural increasing order of indices.

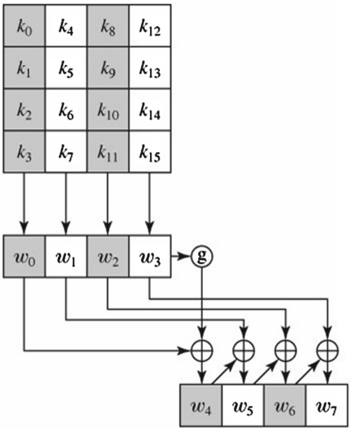


Fig. 3. The AES key schedule

Word $w_3 = (k_{12}, ..., k_{15})$ is passed through the *key schedule core*, denoted g in Fig. 3, which rotates the word one position left $rot(w_3) = (k_{13}, k_{14}, k_{15}, k_{12})$ and then forms the result $T_3 = (sb[k_{13}], sb[k_{14}], sb[k_{15}], sb[k_{12}])$. Here $sb[k]$ refers to the S-Box at index $k, k \in \mathbb{F}_{2^8}$. We have the four word state $T = (T_0, T_1, T_2, T_3) = (k_0, ..., k_{11}, sb[k_{13}], sb[k_{14}], sb[k_{15}], sb[k_{12}])$. From T the remaining of the key schedule is (almost) a linear transformation, KS, over ${\mathbb{F}_{2^8}}^{16}$ depicted below:

$$KS \times T \oplus w_3 = \left[\begin{array}{cccc|cccc|cccc|cccc}
\mathbf{1}&0&0&0&0&0&0&0&0&0&0&0&\mathbf{1}&0&0&0\\
0&\mathbf{1}&0&0&0&0&0&0&0&0&0&0&0&\mathbf{1}&0&0\\
0&0&\mathbf{1}&0&0&0&0&0&0&0&0&0&0&0&\mathbf{1}&0\\
0&0&0&\mathbf{1}&0&0&0&0&0&0&0&0&0&0&0&\mathbf{1}\\
\hline
\mathbf{1}&0&0&0&\mathbf{1}&0&0&0&0&0&0&0&\mathbf{1}&0&0&0\\
0&\mathbf{1}&0&0&0&\mathbf{1}&0&0&0&0&0&0&0&\mathbf{1}&0&0\\
0&0&\mathbf{1}&0&0&0&\mathbf{1}&0&0&0&0&0&0&0&\mathbf{1}&0\\
0&0&0&\mathbf{1}&0&0&0&\mathbf{1}&0&0&0&0&0&0&0&\mathbf{1}\\
\hline
\mathbf{1}&0&0&0&\mathbf{1}&0&0&0&\mathbf{1}&0&0&0&\mathbf{1}&0&0&0\\
0&\mathbf{1}&0&0&0&\mathbf{1}&0&0&0&\mathbf{1}&0&0&0&\mathbf{1}&0&0\\
0&0&\mathbf{1}&0&0&0&\mathbf{1}&0&0&0&\mathbf{1}&0&0&0&\mathbf{1}&0\\
0&0&0&\mathbf{1}&0&0&0&\mathbf{1}&0&0&0&\mathbf{1}&0&0&0&\mathbf{1}\\
\hline
\mathbf{1}&0&0&0&\mathbf{1}&0&0&0&\mathbf{1}&0&0&0&\mathbf{1}&0&0&0\\
0&\mathbf{1}&0&0&0&\mathbf{1}&0&0&0&\mathbf{1}&0&0&0&\mathbf{1}&0&0\\
0&0&\mathbf{1}&0&0&0&\mathbf{1}&0&0&0&\mathbf{1}&0&0&0&\mathbf{1}&0\\
0&0&0&\mathbf{1}&0&0&0&\mathbf{1}&0&0&0&\mathbf{1}&0&0&0&\mathbf{1}
\end{array}\right] \times \begin{bmatrix} t_0\\ t_1\\ t_2\\ t_3\\ t_4\\ t_5\\ t_6\\ t_7\\ t_8\\ t_9\\ t_{10}\\ t_{11}\\ t_{12}\\ t_{13}\\ t_{14}\\ t_{15} \end{bmatrix} + \begin{bmatrix} 0\\ 0\\ 0\\ 0\\ 0\\ 0\\ 0\\ 0\\ 0\\ 0\\ 0\\ 0\\ k_{12}\\ k_{13}\\ k_{14}\\ k_{15} \end{bmatrix} \quad (1)$$

When expanding the key from round i to round $i + 1$ we denote $w_0, ..., w_3$ the four words of the current key and denote the new key $w_4, ..., w_7$.

To see how the computation above resembles the operations depicted in Fig. 3 we consider each word of the result in turn. The first word in the new key is w_4. We see from the figure that it should be $w_0 \oplus T_3 = (k_0 \oplus t_{12}, k_1 \oplus t_{13}, k_2 \oplus t_{14}, k_3 \oplus t_{15})$. Doing the inner product with the first four rows of KS and T yields this. Similar observations can be made progressing to rows 4 though 7 and rows 8 through 11 for words w_5 and w_6 respectively. For w_7 the operation is slightly different as it according to Fig. 3 should be $w_3 \oplus w_4 \oplus w_5 \oplus w_6$ where only w_4, w_5 and w_6 are present in our T vector. To complete the operation depicted in Fig. 3 we additionally XOR w_3 onto $KS \times T$ obtaining the same operation as in Fig. 3. The reason for laying out the computation as above will become clear later. Finally we add the round constant to obtain the new key.

The 10 *rounds* are the main encryption loop of AES. Algorithm 1 describes the algorithm. *Add Round Key* covers the operation of XORing the current round key with the current AES state obtaining a new AES state. *The KeyExpansion* step updates the current round key into the one needed for the following round. If the key expansion is computed beforehand as suggested above the KeyExpansion step can be ignored.

Algorithm 1. AES Encryption

```
Data: $S = (s_0, ..., s_{15})$ - /* the AES state */
Data: $K = (k_0, ..., k_{15})$ - /* the AES key */
/* Prepare the 11 round keys */
/* Xor the 0th round key to the state */
AddRoundRey($K$,$S$,0);
for $round \in \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ do
    KeyExpansion($K$,$round$);
    SubByte($S$);
    ShiftRows($S$);
    /* Considered as polynomials over $\mathbb{F}_{2^8}$ the columns of the state
       are multiplied with the fixed polynomial $3x^3 + x^2 + x + 2$
       mod $x^4 + 1$. */
    MixColumns($S$);
    /* Xor the [$round$]th key to the state */
    AddRoundKey($K$,$S$,$round$);
KeyExpansion($K$,10);
SubByte($S$);
ShiftRows($S$);
AddRoundKey($K$,$S$,10);
```

After the final step the 16 bytes AES state in S contains the cipher-text. Our results rely on a mathematical interpretation of the steps in AES which we will give in the following. We consider each step in Algorithm 1.

The *Sub-Byte* step is the operation of replacing each byte in the AES state with the S-Box lookup for that byte. More precisely, if the AES state is $\mathbf{S} = (s_0, ..., s_{15})$ after applying Sub-bytes the AES state is $\mathbf{S}' = (sb[s_0], ..., sb[s_{15}])$. Another interpretation of the S-Box can be found in [DK10]. Here the S-Box is considered a degree 254 polynomial over $\mathbb{F}_{2^8}$. In our case, we will use a lookup table however a low-depth binary circuit for the S-Box can be found in [BP11].

In the *Shift-rows* step we consider the 16 bytes AES state as a 4 × 4 matrix as in Fig. 4a. Then the shiftrows cycles the second row one element, the third row two elements and the fourth row three elements as depicted in Fig. 4b. This operation corresponds to the 16 × 16 linear transformation performed by the

$$S = \begin{bmatrix} s_0 & s_4 & s_8 & s_{12} \\ s_1 & s_5 & s_9 & s_{13} \\ s_2 & s_6 & s_{10} & s_{14} \\ s_3 & s_7 & s_{11} & s_{15} \end{bmatrix}$$

(a) 16 bytes of AES state laid out in a 4×4-matrix.

$$S = \begin{bmatrix} s_0 & s_4 & s_8 & s_{12} \\ s_5 & s_9 & s_{13} & s_1 \\ s_{10} & s_{14} & s_2 & s_6 \\ s_{15} & s_3 & s_7 & s_{11} \end{bmatrix}$$

(b) 16 bytes of AES state with *Shift-rows* applied.

Fig. 4. Illustration of *Shift-rows*.

matrix on the 16-vector holding the AES state, $S = (s_0, ..., s_{15})$

$$\left[\begin{array}{cccc|cccc|cccc|cccc}
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1\\
\hline
0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0\\
0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0\\
\hline
0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0\\
0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0\\
\hline
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0\\
0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0
\end{array}\right]
\times
\begin{bmatrix} s_0\\ s_1\\ s_2\\ s_3\\ s_4\\ s_5\\ s_6\\ s_7\\ s_8\\ s_9\\ s_{10}\\ s_{11}\\ s_{12}\\ s_{13}\\ s_{14}\\ s_{15} \end{bmatrix}
=
\begin{bmatrix} s_0\\ s_5\\ s_{10}\\ s_{15}\\ s_4\\ s_9\\ s_{14}\\ s_3\\ s_8\\ s_{13}\\ s_2\\ s_7\\ s_{12}\\ s_1\\ s_6\\ s_{11} \end{bmatrix}
\qquad (2)$$

The *Mix-columns* step can also be described as a linear transformation, see Eq. 3. We are going to apply these matrices in the pre-processing phase and in the on-line phases of our protocol using a trick which will be explained later. In all of our application of *Shift-rows* and *Mix-columns* we compute on many AES blocks in parallel. For this we introduce one additional bit of notation. Let SR and MC be the *Shift-rows* and *Mix-columns* matrices as above respectively. To apply e.g. SR to a vector holding n AES states we write $SR_n \times S$ where SR_n is the $16n \times 16n$-matrix having n SR on the diagonal and zero everywhere else. We denote $SRMC$ the matrix that applies *Shift-rows* followed by *Mix-columns* to S. That is $SRMC \times S = MC \times SR \times S$. Likewise we also denote $SRMC_n$ the $16n \times 16n$ matrix having n SRMC on its diagonal.

$$\left[\begin{array}{cccc|cccc|cccc|cccc}
2&3&1&1&0&0&0&0&0&0&0&0&0&0&0&0\\
1&2&3&1&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&2&3&0&0&0&0&0&0&0&0&0&0&0&0\\
3&1&1&2&0&0&0&0&0&0&0&0&0&0&0&0\\
\hline
0&0&0&0&2&3&1&1&0&0&0&0&0&0&0&0\\
0&0&0&0&1&2&3&1&0&0&0&0&0&0&0&0\\
0&0&0&0&1&1&2&3&0&0&0&0&0&0&0&0\\
0&0&0&0&3&1&1&2&0&0&0&0&0&0&0&0\\
\hline
0&0&0&0&0&0&0&0&2&3&1&1&0&0&0&0\\
0&0&0&0&0&0&0&0&1&2&3&1&0&0&0&0\\
0&0&0&0&0&0&0&0&1&1&2&3&0&0&0&0\\
0&0&0&0&0&0&0&0&3&1&1&2&0&0&0&0\\
\hline
0&0&0&0&0&0&0&0&0&0&0&0&2&3&1&1\\
0&0&0&0&0&0&0&0&0&0&0&0&1&2&3&1\\
0&0&0&0&0&0&0&0&0&0&0&0&1&1&2&3\\
0&0&0&0&0&0&0&0&0&0&0&0&3&1&1&2
\end{array}\right]
\times
\begin{bmatrix} s_0\\ s_1\\ s_2\\ s_3\\ s_4\\ s_5\\ s_6\\ s_7\\ s_8\\ s_9\\ s_{10}\\ s_{11}\\ s_{12}\\ s_{13}\\ s_{14}\\ s_{15} \end{bmatrix}
=
\left[\begin{array}{c}
2s_0 + 3s_1 + s_2 + s_3\\
s_0 + 2s_1 + 3s_2 + s_3\\
s_0 + s_1 + 2s_2 + 3s_3\\
3s_0 + s_1 + s_2 + 2s_3\\
\hline
2s_4 + 3s_5 + s_6 + s_7\\
s_4 + 2s_5 + 3s_6 + s_7\\
s_4 + s_5 + 2s_6 + 3s_7\\
3s_5 + s_5 + s_6 + 2s_7\\
\hline
2s_8 + 3s_9 + s_{10} + s_{11}\\
s_8 + 2s_9 + 3s_{10} + s_{11}\\
s_8 + s_9 + 2s_{10} + 3s_{11}\\
3s_8 + s_9 + s_{10} + 2s_{11}\\
\hline
2s_{12} + 3s_{13} + s_{14} + s_{15}\\
s_{12} + 2s_{13} + 3s_{14} + s_{15}\\
s_{12} + s_{13} + 2s_{14} + 3s_{15}\\
3s_{12} + s_{13} + s_{14} + 2s_{15}
\end{array}\right]
\qquad (3)$$

The organization of this paper is as follows. In Sect. 2 we present how to compute Oblivious AES as a multi-party computation with dedicated pre-processing.

We actually implemented this work as code and report on running times as low as 5 ms for 5 simultaneous AES instances in Fig. 2. Our work can be reproduced following our instructions in Appendix A. Then in Sect. 3 we discuss an optimization of the protocol from the fact that with dedicated pre-processing the online computation is all linear. In Sect. 4 we discuss another improvement reducing the size of our pre-processing material required. Finally in Sect. 5 we show how to efficiently get dedicated pre-processing from a general MiniMac instance evaluating arithmetic circuits over $\mathbb{F}_{2^8}$.

2 Fast AES Using Dedicated Pre-processing

In this Section we show how dedicated pre-processing can be used to efficiently compute Oblivious AES. We list demands for the pre-processing material needed and describe an online protocol using the material to compute Oblivious AES. Also we present an implementation and performance numbers on consumer grade hardware.

We employ the fastest version of MiniMac implementation from [DLT14] which allows us to compute on vectors containing 85 bytes. In such a vector we can pack 5 full AES states taking up 80 bytes. In this way we run a small number of AES circuits in parallel. However notice here that we are running "different" operations on individual bytes in the representation as we are not performing the same operations to all bytes in the AES state. This is not supported directly by MiniMac and hence we need help from the pre-processing.

The pre-processing will generate tables of correlated randomness corresponding to handling the entire AES round (except add round key). Loosely put, we pre-process random values with the AES round operation applied to them. Then, we correct these at runtime to yield the AES round operation on the actual input values.

We start by describing how we pre-process the *S-Box*. The S-Box can be thought of as a table with 256 entries. To apply the S-Box to a single byte in the AES state we look up the entry corresponding to that byte (e.i. the state byte has a numeric base 10 value between 0 and 255 which we use as index into the S-box). Lets consider how to do this for a single s_j in our representation with 5 AES states consisting of 80 bytes $\mathbf{S} = (s_0, ..., s_{79})$. To ensure s_j remains secret inside the box we disguise s_j with a uniformly random value R_j and open $R_j + s_j$ to all the parties. Now we will construct a pre-processed table $sb_{+R_j}[\cdot]$ that contains 256 MiniMac representations of S-Box values. However, the indexing into the table is permuted by adding the random R_j. More precisely, we want that

$$\forall s \in \mathbb{F}_{2^8}, j = 0 \ldots 79 : sb_{+R_j}[s + R_j] = [\![(0, \ldots, sb[s], 0, \ldots, 0)]\!]$$

where the value $sb[s]$ is placed in position j[4].

[4] Note that when we say an entry in the table is a MiniMac representation of some vector this actually means that players have additive shares of that vector as well as some MACs and corresponding keys, however, the details of this are not important here.

We will require the pre-processing to also output $[\![(R_0, \ldots, R_{79})]\!]$ and when the time comes to do the S-boxes, we add this to the current state and open $(s_0+R_0, \ldots, s_{79}+R_{79})$. Then we do the 80 table look-ups and add all the outputs effective applying the S-Box to all 80 bytes.

This trick can be extended so that we can make the table look-up implicitly compute also the linear transformation constituted by *Shift-rows* and *Mix-columns*. Let the matrix $SRMC$ denote the Shift-rows matrix multiplied with the Mix-columns matrix from Sect. 2. Note that if taken directly from Sect. 2 this matrix would only operate on one AES state, but it can be extended in a natural way to operate on a vector containing 5 states.

Now, using the same random values R_j, we replace the S-box tables defined above by 80 tables denoted $AESBox_j, j = 0, \ldots, 79$, such that

$$\forall s \in \mathbb{F}_{2^8}, j = 0 \ldots 79 : AESBox_j[s + R_j] = [\![SRMC \times (0, .., 0, sb[s], 0, ..., 0)]\!]$$

where again the non-zero value is placed at position j. Because the multiplication by $SRMC$ is a linear operation, it follows that if we do the 80 table look-ups using $(s_0 + R_0, \ldots, s_{79} + R_{79})$ as indices and add the results, this time we will obtain

$$\sum_{i=0}^{79} [\![SRMC \times (0, .., 0, sb[s_j], 0, ..., 0)]\!] = [\![SRMC \times (sb[s_0], ..., sb[s_{79}])]\!]$$

and these are exactly the 5 new AES states we wanted. The protocol depicted in Fig. 5 describes how one AES round is handled using this approach.

AES Round: The AES round proceeds as follows:

1. Take a fresh $AESBox = \{AESBox_j\}_{j=0,\ldots,79}$ from the available ones and the corresponding $[\![R]\!] = [\![(R_0, ..., R_{79})]\!]$.
2. Let the current state be $[\![S]\!] = [\![(s_0, ..., s_{79})]\!]$. The parties compute $[\![\delta]\!] = [\![R + S]\!]$.
3. $\delta = (\delta_0, ..., \delta_{79})$ is opened
4. As δ is known in plain by all parties, they can look up $S'_j = AESBox[\delta_j], j = 0, ..., 79$ [a].
5. The parties form the state S' after *SubBytes,ShiftRows* and *MixColumns* by computing $S' = \sum_{i=0}^{79} S'_i$.
6. Finally (a MiniMac representation of) the round key is added to the state and the next round follows.

[a] For the 10th round we have a pre-processed AESBox which is the same except we only apply the Shift-Row matrix to the values in the tables.

Fig. 5. Online phase, $\Pi_{AES-round}$

2.1 Experiments with the Implementation

We have implemented the MiniMac on-line phase and a program for creating the pre-processing material for all parties running on one machine. In Fig. 2 we list timings of our experiments.

Execution time of out experiments are recorded as follows:

We have three test machines, two peers who will carry out the MPC and a third monitor who will record execution time. When the Peer processes have loaded pre-processing material from Disk and otherwise ready to commence computation they report “Ready” to the monitor. When both have done so, the monitor will record a time stamp and send “Start” to the Peers. Each Peer report to the monitor “Done” when it has reached completion of the MPC circuit. When all Peers have reported “Done” the monitor records the time and execution time is taken to the difference between our two time stamps. More precisely, for *Peer 0* and *Peer 1* the following happens:

- *Peer 0* starts, connects to the Monitor and listens for *Peer 1* to connect.
- *Peer 1* starts, connects to the Monitor and connects to *Peer 0*
- Then both peers loads pre-processing material and perform input-gates obtaining the initial shared AES state and reports “Ready” to the Monitor. Then they wait for the Monitor to signal start.
- When *All* peers has arrived at an initial AES-state the Monitor signals “Start” and the MPC begins.
- Upon completing the AES circuit each peer reports “Done” to the monitor.
- The Monitor records the time before the first “Start”-signal is issued until the last Peer reports back its computation has completed. The difference between these two time stamps is the computation time we report.

When to include the KeyExpansion requires a bit of discussion. When encrypting many blocks of data the *key expansion* can be computed once and reused. This requires that the round-keys are computed beforehand and stored. Thus using 11 representations, one for each round key, we can compute the key expansion once and reuse it for encrypting any number of blocks afterwards. Therefore, a good approximation of amortized execution time per block of encryption with large bodies of plain-text can be achieved with one round of encryption omitting the key expansion entirely and multiplying up. However, when measuring the latency (with a fresh key not pre-loaded) from when starting the encryption until the first block of cipher-text is ready, the key expansion does count and as we will see, it plays a significant role. In summary we care about two types of measurements: Latency from scratch and amortized execution time per block over many blocks . See the result in Fig. 2. Here, xxx/AES is the number of pre-processed Megabytes required per computed AES block.

3 Exploiting the Absence of On-Line Multiplications

The representation of data used in MiniMac is carefully designed to support secure coordinate-wise multiplication of vectors. However, using the techniques we have seen in the previous section, we do not need such multiplication operations.

In this section we describe how we can exploit this fact to change the data representation so that we can compute on more data at smaller cost.

One important step in representing a vector in MiniMac format (and the only one we need to worry about here) is to encode it in a linear code. In order to support multiplications, one needs two properties from this code: first, the encoding must be in systematic form, that is, the encoded vector appears in the first positions of the resulting codeword. Second, the so-called Schur transform of the code must have large enough minimum distance. To obtain the Schur transform of a code C is the linear span of all vectors in $\{\boldsymbol{a} * \boldsymbol{b} | \ \boldsymbol{a}, \boldsymbol{b} \in C\}$, where $\boldsymbol{a} * \boldsymbol{b}$ is the coordinate-wise (or Schur) product of $\boldsymbol{a}$ and $\boldsymbol{b}$.

The implementation from [DLT14] obtains these properties by encoding vectors of length 85 into a Reed-Solomon code of length 255. Actually, one could use a larger value than 85 and still satisfy the two properties, but since the underlying field contains a root of unity of order 255 and 85 divides 255, these choices allow us to use the FFT algorithm to encode and decode and this speeds up the computations we need quite dramatically.

However, if we do not need to do multiplications, it turns out that the only demand we need to satisfy is that the code itself has large enough minimum distance, more precisely, it just has to be at least the security parameter divided by 8 (since each field element is 8 bits long). Furthermore we no longer need the code to be in systematic form.

With these relaxations, we can choose a Reed-Solomon code of length 255 and encode vectors of length 239. Because the codeword length is still 255, we can use FFT to encode and decode (the requirement for the data length to divide 255 was only necessary to have systematic encoding and still be able to use FFT).

This change will speed up our AES implementation in two ways: first we can pack 14 AES states into one vector instead of 5, immediately yielding a factor of 3 in computational capacity. Secondly, the encoding is faster than before because we no longer need systematic encoding. The reason for this is as follows: a Reed-Solomon codeword is computed by taking a polynomial of at most a certain degree and evaluating it in a set of fixed input points (255 in our case). For systematic encoding the polynomial must take the values specified by our input in the first points, so to encode one must first interpolate to get the right polynomial and then evaluate it in the other points to get the rest of the codeword. For non-systematic encoding one just thinks of the input as coefficients of a polynomial and then we just evaluate.

We did no do the resulting AES implementation, but since the number of rounds will be the same and local computation is simpler, we can safely assume that the total time for the protocol will not be larger than before. But we now compute 14 AES instances instead of 5, and in fact we can put 15 AES instances if we settle 120 bits of security in the authentication of data values which is more than enough in practice. So we can expect an amortized time of $0.4ms$ per AES block and the same latency of $6ms$.

The size of the pre-processing grows significantly as we now have 240 working bytes we need 240 S-Boxes and we store a 256 entry table for each. The estimated size of the pre-processing material is ≈ 650 Mb per player per AES block we will try to improve on this in the following.

4 Minimizing Size of the Pre-processing Material

The ideas we described so far require a rather large amount of pre-processed material. Each of the S-Box tables we have been using so far has 256 entries where each entry is an entire MiniMac codeword which requires 1056 bytes of storage for each player. This translates to approximately $21MB$ of pre-processed data per player per AES round using the first method we presented. We suggest in the following a different way to represent the tables that saves a factor of about 60 in the pre-processing size. The price we pay for this optimization is one extra round of communication per AES round and some extra local computation.

The Idea. We first describe our idea for organizing tables in a generic fashion because we believe it can be interesting in other contexts than secure AES. So assume that we are working with an arithmetic black-box, we have computed $[\![x]\!]$ and would like to compute $[\![f(x)]\!]$. We assume for concreteness that $x \in \mathbb{F}_{2^8}$, but this is not necessary in general. If f is rather complicated to compute via a circuit, as is the case for the AES S-Box, we can do better using a precomputed table. The first step towards this is similar to what we already did above: we will pre-process a random value $[\![R]\!]$ and also pre-process a table f_{+R} defined as

$$f_{+R}[z+R] = f(z), \text{ for } z = 0, \ldots 255.$$

Now we can compute and open $[\![x+R]\!] = [\![x]\!] + [\![R]\!]$ and look up in the table. This will hide x because we add R but is of course insecure because $f(x)$ will become public.

A slightly better idea is to pre-process a random $[\![v]\!]$ and re-define the table as

$$f_{+R}[z+R] = f(z) + v, \text{ for } z = 0, \ldots 255.$$

Now the table look-up will produce $f(x)+v$ and we can add this to $[\![v]\!]$ to get $[\![f(x)]\!]$. This is still not secure, however: we use the same mask v for all entries and so different table entries are not independent and we may reveal information on how the table was permuted and hence indirectly information on x.

So the final idea is to not store the table in the clear but instead secret share the entries additively between the players. We will only open the entry we actually look up, and now it is secure to use the same mask v for all entries. To prevent players from lying about their shares, we add standard message authentication codes (MACs) to the shares.

More concretely, this means that for each table entry $w = f(z)+v$, we choose in the pre-processing random r_1, r_2 such that $r_1 + r_2 = w$, and in addition we choose random vectors $\boldsymbol{a}_1, \boldsymbol{b}_1, \boldsymbol{a}_2, \boldsymbol{b}_2$, these will serve as MAC keys. Then we compute MACs, $\boldsymbol{m}_1 = \boldsymbol{a}_2 * (r_1, \ldots, r_1) + \boldsymbol{b}_2$ and $\boldsymbol{m}_2 = \boldsymbol{a}_1 * (r_2, \ldots, r_2) + \boldsymbol{b}_1$ and give $r_1, \boldsymbol{m}_1, \boldsymbol{a}_1, \boldsymbol{b}_1$ to the first player and $r_2, \boldsymbol{m}_2, \boldsymbol{a}_2, \boldsymbol{b}_2$ to the other. Here, $*$ denotes the coordinate-wise (or Schur) product of vectors.

We will use $(\!|w|\!)$ to denote all this data in the following. The reader should think of this as a randomized representation of w that can be reliably opened: the players would exchange shares and MACs and then use their keys to check the MACs. It is well known and easy to prove that having, say, the first player accept an incorrect value requires that you guess $\boldsymbol{a}_1$. So if we choose the length of $\boldsymbol{a}_i$ and $\boldsymbol{b}_i$ to be 8 bytes, for instance, we get 64 bits of (unconditional) security which should be more than enough in most cases.

So the final table is of form

$$f_{+R}[z + R] = (\!|f(z) + v|\!), \text{ for } z = 0, \ldots 255.$$

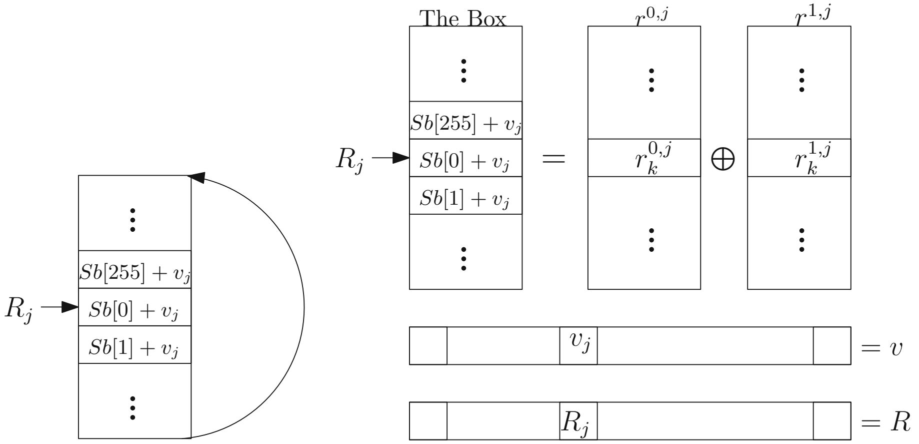
We will need one representation $(\!|\cdot|\!)$ for each table entry, but the values $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$ can be the same for all entries without affecting security. So in this case, a table entry requires essentially $1 + 8 + 8 = 17$ bytes for each player. This is a factor more than 60 less than the 1056 bytes we needed before.

A final observation is that if what we really want to compute is not $[\![f(x)]\!]$ but $[\![L(f(x))]\!]$ where L is a linear function, then we can precompute $[\![L(v)]\!]$. When players have computed $f(x) + v$ they can locally compute $L(f(x) + v) = L(f(x)) + L(v)$ and add this into $[\![L(v)]\!]$ to get $[\![L(f(x))]\!]$.

Using the Idea for AES. In the following we describe our observation above using plain codewords with 240 working bytes in each representation. We start by designing the content of the pre-processed S-Boxes differently as follows. We have a random $[\![R]\!] = [\![R_0, ..., R_{239}]\!]$ and 240 tables in mind namely one table for each byte in our 15 AES states. Let $[\![S]\!] = [\![S_0, ..., S_{239}]\!]$ denote the MiniMac representation holding our 15 AES states. For S_j the jth state byte we consider the table with S-Box values rotated by R_j and masked with a single v_j from a random $v \in \mathbb{F}_{2^8}^{240}$, see Fig. 6a. Now our idea is to additively share this table between the players with MACs. Thus each player m gets a table $r^{m,j}$ such that the entries $r_k^{m,j}$ in $r^{m,j}$ add up to $Sb[(R_j + k) \mod 256] + v_j = \sum_{m=0}^{n-1} r_k^{m,j}$ for fixed k, j summing over m adding each of the shares held by the players. This is illustrated in Fig. 6b for two players, e.g. $m \in \{0, 1\}$. Each table $r^{m,j}$ is MACed towards the other player(s). Thus in addition player m has a table $M(r^{l,j})$ for $l \in \{i | 0 \leq i < n \wedge i \neq m\}$. For two players the situation is depicted in Fig. 7. Thus we define an *AESBox* in this new set up as a triple of three things: A random representation $[\![R]\!]$, a random representation $\mathbf{v}$ with the *SRMC*-linear transformation applied to it $[\![SRMC \times v]\!]$ and a set of 240 tables with 256 entries constructed as described above.

$$AESBox = \{[\![R]\!], [\![SRMC_{15} \times \mathbf{v}]\!], \{(\!|r_k^j|\!)\}_{j \in [239], k \in [255]}\}$$

Here we used $(\!|r_k^j|\!)$ to denote the set of values shared with MACs as described in Fig. 7 for all the players constituting table j.



(a) Table with 256 S-Box entries rotated by a random value R_j and masked by v_j. Each entry is one byte.

(b) The Box intended for state position j now additively shared in $r^{0,j}$ and $r^{1,j}$. Player i gets table $r^{i,j} = (r_0^{i,j}, ..., r_{255}^{i,j})$. Each players also gets his shared state for $\llbracket \mathbf{v} \rrbracket$ and $\llbracket R \rrbracket$.

Fig. 6. The new layout

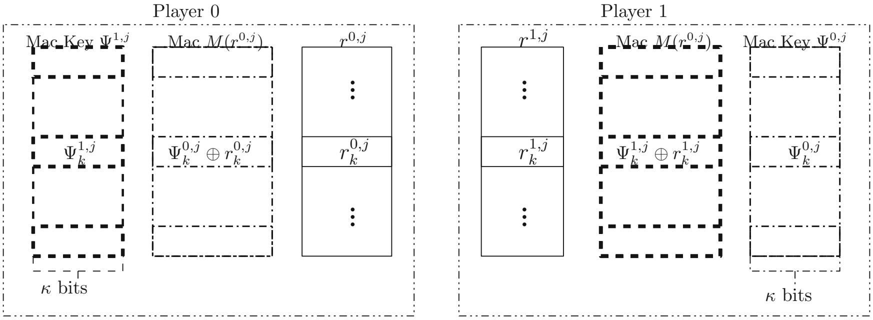


Fig. 7. Above we depict the $\llparenthesis x \rrparenthesis = \{\{x_0 \in \mathbb{F}_{2^8},\ M(x_0) \in \mathbb{F}_{2^8}{}^{\kappa/8}, \Psi_1 \in \mathbb{F}_{2^8}{}^{\kappa/8}\}, \{x_1 \in \mathbb{F}_{2^8}, M(x_1) \in \mathbb{F}_{2^8}{}^{\kappa/8}, \Psi_0 \in \mathbb{F}_{2^8}{}^{\kappa/8}\}\}$ representation. Player 0 hold from left to right a MAC key table, a Mac table and a table of values. The Key table allows Player 0 to check the table of values held by Player 1. The MAC table allows player 0 to convince Player 1 his table of values is authentic.

The on-line phase is summarized in Fig. 5. Similar to our previous solution we start by taking an $AESBox = (\llbracket R \rrbracket, \llbracket SRMC \times v \rrbracket, \{\llparenthesis r_k^j \rrparenthesis\}_{j \in [239], k \in [255]}\})$ and "blind" the the current states in S by adding our random R to it obtaining $\llbracket \Delta \rrbracket = \llbracket S \oplus R \rrbracket$. Then we open $\llbracket \Delta \rrbracket$ to everyone. Now since our $r^{m,j}$ tables are shares of the S-Box masked by v_j and rotated by R_j we can lookup $\llparenthesis sb[S_j] \oplus v_j \rrparenthesis$ by letting each player m take $r_{\Delta_j}^{m,j}$ to be his share of $\llparenthesis T_j \rrparenthesis = \llparenthesis sb[S_j] \oplus v_j \rrparenthesis$. As $\mathbf{v}$ is randomly chosen it blinds the actual value looked up in the S-Box thus we can safely open $\llparenthesis T_j \rrparenthesis$. To open $\llparenthesis T_j \rrparenthesis$ the parties exchange the Values and MACs describe above in

$\Phi_{AESRound}$

1 Take an available $AESBox = (\llbracket R \rrbracket, \llbracket SRMC \times v \rrbracket, \{\llparenthesis r^j_k \rrparenthesis\}_{j \in [239], k \in [255]})$
2 All parties compute $\llbracket \Delta \rrbracket = \llbracket S \rrbracket \oplus \llbracket R \rrbracket = \llbracket S \oplus R \rrbracket$
3 $\llbracket \Delta \rrbracket$ is opened to each player.
4 Each player uses Δ_j to lookup $r^{m,j}_{\Delta_j}$, obtaining $\llparenthesis T_j \rrparenthesis = \llparenthesis sb[S_j] \oplus v_j \rrparenthesis$[a] as a $\llparenthesis \cdot \rrparenthesis$-representation between them.
5 T_j is opened to everyone for $j \in [239]$.
6 The players compute $T = \sum_{j=0}^{239} T_j = sb[S] \oplus v$
7 The players computes $SRMC \times T = SRMC \times sb[S] \oplus SRMC \times v$
8 The players take $S' = SRMC \times T \oplus \llbracket SRMC \times v \rrbracket = \llbracket SRMC \times sb[S] \rrbracket$ as the AES state after *SubBytes*, *Shift-rows* and *Mix-columns*.
9 Finally the AES round key $\llbracket K_{round} \rrbracket$ is added obtaining the next state $S_{round+1} = S' \oplus K_{round}$.

[a] $r^{m,j}_{\Delta_j}$ is the particular share each player can lookup. Here we use $\llparenthesis T_j \rrparenthesis$ when referring to these shares as a combined shared value T_j mutually authenticated with MACs.

Fig. 8. $\Phi_{AESRound}$

Fig. 7 and the receiving parties checks that the MACs are correct. If no one aborts everybody know $T_j = sb[S_j] \oplus v_j$. Knowing all $T_j, \forall j \in [239]$ the players compute $T = \bigoplus_{j \in [239]} T_j$. Now the parties take the linear transformation $SRMC_{15}$ and apply it to T obtaining $SRMC_{15} \times T = SRMC_{15} \times S \oplus SRMC_{15} \times v$. Then the new state S' after *SubBytes*, *Shift-rows* and *Mix-columns* is computed as $SRMC_{15} \times T \oplus \llbracket SRMC_{15} \times v \rrbracket = \llbracket SRMC_{15} \times sb[S] \rrbracket$. Finally the round key is added to the S' and the following AES round follows.

For the 10th round $SRMC_{15}$ is replaced by the linear transformation SR_{15} which is the 240 × 240 matrix having 15 SR matrices on its diagonal. Note this influences $\llbracket SR_{15} \times v \rrbracket$ requiring a bit of book keeping taking a special *AESBox* for the last round.

This protocol requires two rounds of communication instead of one for the first two protocols we presented. Also, it requires players to compute the linear mapping $SRMC$ locally. This, however, can be done in a simple way by a table look-up for each byte position in the input. Therefore we conservatively estimate that this protocol will require twice the time needed for protocol 1 (Fig. 8).

5 Pre-processing from the Original MiniMac Protocol

Our solutions above put some quite specialized requirements on the pre-processing material. In this section we show how one may generate such data by first running the pre-processing phase of the original MiniMac protocol and then using this to run the original MiniMac online phase. We set this up such that the function we compute will output the pre-processing material we need for

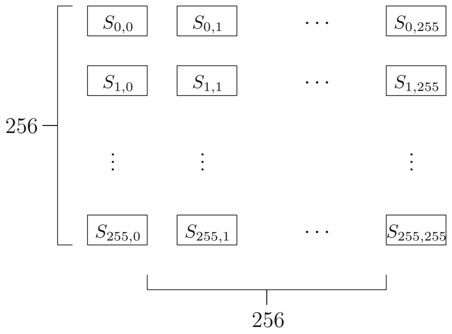


Fig. 9. 256 × 256 table for entry j in $[\![R]\!]$ with entry $S_{i,s} = SRMC_5 \times \Psi$ for Ψ having $sb[i+s]$ in position j.

our construction. We will describe how to generate the *AESBox*es as required by our protocol in Sect. 2. Generating pre-processing material for the protocol in Sects. 3 and 4 is a matter of applying appropriate linear transformations to the result presented here. Our goal is to generate *AESBox* tables from a random representation $[\![(R_1, ..., R_{85})]\!]$: Getting such a random value is directly supported by MiniMac. Now for each position j in $[\![(R_1, ..., R_{85})]\!]$ we take 256^2 as depicted in Fig. 9. Each entry $S_{i,s}$ is a public MiniMac representation with value $SRMC_5 \times (0, \ldots, sb[i+s], \ldots, 0), i, s \in \mathbb{F}_{2^8}$.

Recall that our protocol in Sect. 2 requires an *AESBox* to have the form:

$$s \in \mathbb{F}_{2^8}, AESBox_j[R_j + s] = [\![SRMC_5 \times (0, ..., sb[s], ..., 0)]\!]$$

This is exactly the values stored in row R_j of our 256^2 table above. The challenge is to lookup this row. To this end we start by computing the vector $[\![(0, \ldots, R_j, \ldots, 0)]\!] = (0, \ldots, 1, \ldots, 0) \times [\![R]\!]$. Recall that the original MiniMac protocol in [DZ13] allows its pre-processing to generate values of the form:

$$[\![R]\!], [\![L \times R]\!]$$

for linear transformations L. The particular transformation we are after here is the one replicating R_j onto every position obtaining $[\![(R_j, \ldots, R_j)]\!]$. Then we compute $\Phi_i = (\mathbf{1} - [\![(R_j, \ldots, R_j) - \mathbf{i}]\!]^{255})$ for $i \in \mathbb{F}_{2^8}$ where $\mathbf{i}$ is the 85 vector with i in all entries. The resulting table for all $s \in \mathbb{F}_{2^8}$ is $AESBox_j[R_j + s] = \sum_{i=0}^{255} \Phi_i \times S_{i,s}$.

To see why this is actually what we wanted consider Φ_i. Because the subgroup of units in $\mathbb{F}_{2^8}$ has order 255 $[\![(R_j, ..., R_j) - \mathbf{i}]\!]^{255}$ is all ones when $R_j \neq i$ and zero only when $R_j = i$. As we want ones when they are equal we compute $(\mathbf{1} - ([\![(R_j, ..., R_j) - \mathbf{i}]\!]^{255})) = \Phi_i$ which is all ones only when $R_j = i$ and all zero otherwise. In this way Φ_i selects the row of $S_{i,s}$ where $i = R_j$ forming our $AESBox_j$ for each possible value of s. Now the steps above are repeated for all entries in $[\![R]\!]$ forming the full $AESBox = \{AESBox_j\}_{j=0,\ldots,84}$.

We note that the $S_{i,j}$ tables do not all have to exist in memory at the same time; it is enough to generate the columns as needed on the fly.

6 Conclusion

We have seen that for dishonest majority protocols in the preprocessing model, the efficiency and in particular the latency of oblivious AES can be dramatically improved by tailoring the preprocessed data to the structure of AES. And that in particular that the only structure that matters is the fact that AES makes use of Sboxes with small input, so that we can use table look-up to circumvent the use of circuits to compute the non-linear parts.

Our study shows that we need only about 0.4 ms amortised time and 6 ms latency to do AES, which seems completely adequate for real life applications such as verifying 1-time passwords.

In future work, it would be interesting to see if other block ciphers or hash functions can be done securely and practically with a similar approach.

A Reproducing our results with the implementation

Getting the code

The implementation of our work can be found on GitHub at http://tinyurl.com/qbx99jv

Requirements

- AutoMake 1.15
- Bash 3.2 or later
- Reasonable GCC compiler supporting C99 (or Windows SDK Visual Studio 2013 or later).

Building on Windows IA64

Install Visual Studio 2013 and open the solution file in `miniapps/dedicatedaes/winx64/daestest.sln`. Press `F7` in the x64-release build configuration to build the code. We have experienced problems with many small allocations on Windows making the `malloc` and `free` implementation on this system degenerate in performance.

Building on Linux and OSX

To build the code type `./build.sh release` or `./build.sh debug` depending on which configuration you want. To reproduce the performance numbers reported in the paper please build in the `release` configuration.

Generating pre-processing material for testing

Running the program with command line arguments `-prep` will generate the default set of preprocessing material needed to compute one block of ciphertext. `./miniapps/dedicatedaes/linux/src/cheetah -prep` or on windows setting the command-line arguments and pressing `F5`. Alternatively the windows .exe file can be located in `miniapps/dedicatedaes/winx64/daestest/Debug/daestest.exe`.

Running the protocol

Running the program with `-mpc -prepfile <filename>` will make the process given aes preprocessing material file for player zero listen and wait for the other players to connect.

E.g. for two players

`cheetah -mpc -prepfile ./aes_prep_4_player_0.rep` will start the listening peer listening on all interfaces port 2020. While `cheetah -mpc -prepfile ./aes_prep_4_player_1.rep -ip xxx.yyy.zzz.www -port 2020` will connect to a peer at ip-address xxx.yyy.zzz.www on port 2020.

Our experimental setup

The lab computers used for our experiments are interconnected by a 1 Gigabit LAN with the specifications below.

```
CPU: i7-3770K CPU @ 3.50GHz with 8 cores
Mem: 8Gb of Ram
Net: Gigabit LAN
OS:  3.13.0-59-generic #98-Ubuntu SMP Fri Jul 24
     21:05:26 UTC 2015 x86_64 x86_64 x86_64 GNU/Linux
```

We emphasize that our implementation carries out the computational steps of the protocol single threaded.

The implementation does use additional threads for receiving and sending network messages. In this way the computational thread is as independent of network delays as the protocol allows. More precisely, the concrete interleaving of sending and receiving messages does not effect computational progress unless data from other parties are strictly required for the protocol to continue.

References

[BP11] Boyar, J., Peralta, R.: A depth-16 circuit for the AES S-box. Cryptology ePrint Archive, Report 2011/332 (2011). http://eprint.iacr.org/

[DK10] Damgård, I., Keller, M.: Secure multiparty AES. In: Financial Cryptography and Data Security, 14th International Conference, FC , Tenerife, Canary Islands, 25–28 January 2010, Revised Selected Papers, pp. 367–374 (2010)

[DKL+12] Damgård, I., Keller, M., Larraia, E., Miles, C., Smart, N.P.: Implementing AES via an actively/covertly secure dishonest-majority MPC protocol. In: Visconti, I., De Prisco, R. (eds.) SCN 2012. LNCS, vol. 7485, pp. 241–263. Springer, Heidelberg (2012)

[DKL+13] Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical covertly secure MPC for dishonest majority – or: breaking the SPDZ limits. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) ESORICS 2013. LNCS, vol. 8134, pp. 1–18. Springer, Heidelberg (2013)

[DLT14] Damgård, I., Lauritsen, R., Toft, T.: An empirical study and some improvements of the minimac protocol for secure computation. IACR Cryptology ePrint Archive 2014:289 (2014)

[DN03] Damgård, I.B., Nielsen, J.B.: Universally composable efficient multiparty computation from threshold homomorphic encryption. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 247–264. Springer, Heidelberg (2003)

[DR00] Daemen, J., Rijmen, V.: Rijndael for AES. In: AES Candidate Conference, pp. 343–348 (2000)

[DZ13] Damgård, I., Zakarias, S.: Constant-overhead secure computation of boolean circuits using preprocessing. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 621–641. Springer, Heidelberg (2013)

[FJN14] Frederiksen, T.K., Jakobsen, T.P., Nielsen, J.B.: Faster maliciously secure two-party computation using the GPU. In: Abdalla, M., De Prisco, R. (eds.) SCN 2014. LNCS, vol. 8642, pp. 358–379. Springer, Heidelberg (2014)

[GHS12] Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the AES circuit. IACR Cryptology ePrint Archive 2012:99 (2012)

[HEKM11] Huang, Y., Evans, D., Katz, J., Malka, L.: Faster secure two-party computation using garbled circuits. In: 20th USENIX Security Symposium, San Francisco, CA, USA, 8–12 August 2011

[HKS+10] Henecka, W., Kögl, S., Sadeghi, A.-R., Schneider, T., Wehrenberg, I.: TASTY: tool for automating secure two-party computations. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS, Chicago, Illinois, USA, 4–8 October 2010, pp. 451–462 (2010)

[IKM+13] Ishai, Y., Kushilevitz, E., Meldgaard, S., Orlandi, C., Paskin-Cherniavsky, A.: On the power of correlated randomness in secure computation. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 600–620. Springer, Heidelberg (2013)

[KSS13] Keller, M., Scholl, P., Smart, N.P.: An architecture for practical actively secure MPC with dishonest majority. In: ACM SIGSAC Conference on Computer and Communications Security, CCS 2013, Berlin, Germany, 4–8 November 2013, pp. 549–560 (2013)

[NNOB12] Nielsen, J.B., Nordholt, P.S., Orlandi, C., Burra, S.S.: A new approach to practical active-secure two-party computation. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 681–700. Springer, Heidelberg (2012)

[PSSW09] Pinkas, B., Schneider, T., Smart, N.P., Williams, S.C.: Secure two-party computation is practical. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 250–267. Springer, Heidelberg (2009)

[Yao82] Andrew Chi-Chih Yao: Protocols for secure computations (extended abstract). In: 27th Annual Symposium on Foundations of Computer Science, vol. 1982, pp. 160–164 (1986)

[Yao86] Yao, A.C.-C.: How to generate and exchange secrets. In: Foundations of Computer Science, vol. 1986, pp. 162–167. IEEE (1986)