An Empirical Study and some Improvements of the MiniMac Protocol for Secure Computation

Ivan Damgård*, Rasmus Lauritsen*, and Tomas Toft**

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

Abstract. Recent developments in Multi-party Computation (MPC) has resulted in very efficient protocols for dishonest majority in the preprocessing model. In particular, two very promising protocols for Boolean circuits have been proposed by Nielsen et al. (nicknamed TinyOT) and by Damgård and Zakarias (nicknamed MiniMac). While TinyOT has already been implemented, we present in this paper the first implementation of MiniMac, using the same platform as the existing TinyOT implementation. We also suggest several improvements of MiniMac, both on the protocol design and implementation level. In particular, we suggest a modification of MiniMac that achieves increased parallelism at no extra communication cost. This gives an asymptotic improvement of the original protocol as well as an 8-fold speed-up of our implementation. We compare the resulting protocol to TinyOT for the case of secure computation in parallel of a large number of AES encryptions and find that it performs better than results reported so far on TinyOT, on the same hardware.

1 Introduction

In Multi-party Computation (MPC), N players wish to compute a function securely on privately held inputs, where security means that the result must be correct, and be the only new information that is released about the inputs. This must hold even if T players are corrupted by an adversary. In this paper, we consider the case of active corruption (where the adversary takes full control over corrupted players) of up to $N - 1$ players, so-called dishonest majority. In several recent papers [2,6,5,7,9] it has been shown that we can obtain very practical MPC protocols for dishonest majority using preprocessing.

The basic idea exploited is that, while dishonest majority precludes information-theoretic (IT) constructions, the expensive and inefficient, computationally secure (public key) cryptography can be pushed to a preprocessing phase. This

* 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.

** Supported by the European Research Council Stating Grant 279447

phase is independent of not only the inputs, but also the function to be securely computed. The "raw-material" generated allows the use of IT-secure protocols in the online phase, i.e., protocols that are much more computationally efficient. Two promising protocols have been proposed for the case of secure computation of Boolean circuits, namely the TinyOT protocol by Nielsen et al. [9] and the MiniMac protocol by Damgård and Zakarias [7]. [7] presents a theoretical comparison between the two: For security parameter κ, it was shown that where TinyOT requires each player to do $\Theta(\kappa)$ elementary bit operations per gate in the circuit, MiniMac requires only $O(\log \kappa)$ operations (or even $O(1)$ for some instantiations, when the number of players is large). The same overhead was found for communication and the amount of preprocessing-data each player stores.
It is, however, very unclear whether these theoretical advantages translate to greater efficiency in practice. Indeed, TinyOT has been implemented with promising results, and the $\Theta(\kappa)$ bit operations required per gate can in most cases be performed in parallel using a single or small number of CPU instructions. On the other hand, the fact that TinyOT has larger storage requirements and communication complexity remains even on a massively parallel machine.

Our Contribution In this paper we present the first implementation of MiniMac. We compare this to a TinyOT implementation running on the same hardware with the goal of making a meaningful comparison of the two approaches in practice. As benchmark, we use parallel computation of many AES encryptions using a binary circuit; AES is often used as the de facto standard benchmark and performing multiple parallel executions has practical relevance, e.g., when encrypting data in counter mode. Additionally, we propose a new modification of MiniMac which increases the efficiency of binary circuit evaluation in both theory and practice. Our implementation is optimized for the two party case but works for any number of players.
MiniMac is based on an error correcting code of length n and dimension k over some finite field, $\mathbb{F}$, and allows k parallel evaluations of some arithmetic circuit over $\mathbb{F}$. It further uses an IT-secure authentication scheme that is based on the code; a forged message authentication code (MAC) will be accepted with probability 2^{-t} where $t = \log(|\mathbb{F}|) \cdot (n - 2k + 1)$.
Our implementation uses a Reed-Solomon code over $\mathbb{F}_{2^8}$ with $(n, k) = (256, 120)$ or $(255, 85)$, depending on the underlying implementation of operations on codewords; this implies (at least) 128-bit security. We note that [7] suggests alternative constructions of binary codes based on algebraic geometry, however, the constants involved are very large and no truly efficient encoding algorithm is known, thus, using these were not seen as a viable approach.
Using $\mathbb{F}_{2^8}$ as the underlying field is a natural choice; elements can be encoded as bytes and addressed separately, while the field is sufficiently small to allow efficient multiplication through table-lookup. Further, $\mathbb{F}_{2^8}$ has characteristic two implying that binary-XOR is simply addition.
Regarding the choice of (n, k), we strove to maximize parallel computation, i.e., maximize k. Since n is bounded by the cardinality of the field, $\mathbb{F}_{2^8}$, we have $(n, k) = (256, 120)$ when aiming for security level $t \geq 128$. However, for these

parameters, our best encoding algorithm is quadratic: the naive multiplication by the generator matrix. We found this matrix multiplication to be quite costly in practice. As an alternative, we suggest to implement it using a variant of Fast Fourier Transform (FFT) – 85 divides 255 which is the order of the multiplicative group implying that roots of unity of order 255 and 85 exist in $\mathbb{F}_{2^8}$. This reduces encoding- and decoding-time, but requires that the input-size divides 255. This sets $(n, k) = (255, 85)$, however, despite the reduced parallelization we found that this still pays off.

In addition to the above implementation suggestions, we also present an improvement of the protocol when the overall goal is Boolean circuits. Taking a closer look at the choice of field, we see that using a code over $\mathbb{F}_{2^8}$ with the original MiniMac protocol to compute a Boolean circuit, implies that every bit is encoded as a byte, i.e., we "waste" a factor of 8 in terms of space. We propose an optimization that allows us to use every bit in every data byte; this increases parallelization to $8 \cdot 120 = 960$ or $8 \cdot 85 = 680$ instances. To reach this goal, we redesign the multiplication operation in MiniMac: the original protocol implements multiplication of data bytes as multiplication in $\mathbb{F}_{2^8}$. Instead, we compute bit-wise AND of two bytes. Our solution for this generalizes to other characteristic 2 fields (and even other protocols, e.g., [6]) and while it requires a small amount of extra local computation, it saves communication and storage compared to the original MiniMac. More precisely, in [7] they compute the cost per player per gate of their protocol, where the cost can be the amount of data stored from the pre-processing, the communication and the computational work. For the case of two players, these costs can be $O(1), O(1), O(\text{poly}(n))$, or all three can be $O(\text{polylog}(n))$, depending on the underlying code used. For the case of computing the same function many times in parallel, our solution obtains $O(\log(n)), O(1), O(\text{polylog}(n))$ and so is better than both previous solutions on the parameters that matter most for efficiency. This is not only a theoretical improvement: in our implementation, we obtain more than an 8-fold speedup.

Another performance boost was obtained by exploiting the structure of the AES circuit: only AND gates require communication, and we form a number of batches of gates, such that gates of one batch can all be executed in parallel. We collect the required communication in packets that each span the entire batch. This way we send fewer but larger packets and this turns out to reduce the time we wait for communication to happen. However, profiling showed that significant time was still spent waiting for communication. We therefore tried a new setup, where several instances of the program were started at slightly different times. The idea was that this would keep both CPUs busy almost all the time and give us larger throughput. Indeed, we gained a factor almost 2 from this. Finally experimenting with the best setting for compiler optimizations gave another factor of 2.

Combining all the tricks we came up with, we obtain an amortized time of about 4 ms per AES encryption with (at least) 128 bit security and about 9 seconds latency. This is almost 10 times faster than the best time reported for TinyOT evaluating the same circuit on the same hardware, where we note that this TinyOT implementation runs with only 64 bit security and much larger latency.

In [8], a different secure AES implementation is reported on, based on the SPDZ protocol [6]. It is a bit faster than ours, but is incomparable as the hardware is different and the security level lower (40 bit). Most importantly, however, the circuit used there is an arithmetic circuit over $\mathbb{F}_{2^8}$ plus some "bit-decomposition" tricks to evaluate the S-boxes. Our study should be seen primarily as targeted against at using MiniMac as efficiently as possible to evaluate a Boolean circuit. Finally, [8] reports that much of their efficiency comes from a very careful scheduling of operations and messages. We have optimized our implementation to some extent w.r.t. scheduling but based on measurements of the time we spend waiting for messages, we believe there is further potential.

2 MiniMac

The MiniMac protocol supports the operations described in Figure 1. It does this by representing values occurring in the computation in a certain format. In the original MiniMac paper this representation is optimized for the case of many players. In this paper, however, we concentrate on the two-player case and therefore we set up the representation in a way that resembles more the way it is done in the TinyOT protocol (which is inherently a two-player protocol). More precisely, whereas in the original MiniMac, each secret value, as well a Message Authentication Code (MAC), is additively secret shared among the players, we keep the additive secret sharing of the value, but add instead a MAC on each share. Whereas this is sub-optimal for the multiple player case, it makes the check of Macs simpler and adds no significant other cost for two players.

Functionality $\mathcal{F}_{\mathrm{MPC}}$

Initialize: On input $(init, k)$ from all parties, each initialize the store for block length k.

Rand: On input $(rand, P_i, vid)$ from all parties P_i, with vid a fresh identifier, pick $\boldsymbol{r} \leftarrow \mathbb{F}^k$ and store $(vid, \boldsymbol{r})$.

Input: On input $(input, P_i, vid, \boldsymbol{x})$ from P_i and $(input, P_i, vid, ?)$ from all other parties, with vid a fresh identifier, store $(vid, \boldsymbol{x})$.

Add: On command $(add, vid_1, vid_2, vid_3)$ from all parties (if vid_1, vid_2 are present in memory and vid_3 is not), retrieve $(vid_1, \boldsymbol{x})$, $(vid_2, \boldsymbol{y})$ and store $(vid_3, \boldsymbol{x}+\boldsymbol{y})$.

Multiply: On input $(mult, vid_1, vid_2, vid_3)$ from all parties (if vid_1, vid_2 are present in memory and vid_3 is not), retrieve $(vid_1, \boldsymbol{x})$, $(vid_2, \boldsymbol{y})$ and store $(vid_3, \boldsymbol{x} * \boldsymbol{y})$.

Output: On input $(output, vid)$ from all honest parties (if vid is present in memory), retrieve $(vid, \boldsymbol{x})$ and output it to the environment. If the environment returns "OK", then output $(vid, \boldsymbol{x})$ to all players, else output $\perp$ to all players.

Fig. 1: The ideal functionality for MPC.

Representation of values A clear text value $\boldsymbol{x}$ is k-element vector over some finite field, in our case always $\mathbb{F}_{2^8}$. We consider a systematic linear error correcting code C of length n and dimension k, and let $C(\boldsymbol{x})$ denote the encoding of $\boldsymbol{x}$ in C. We also need to consider the Schur-transform C^* of C, which is the linear span of all products of form $\boldsymbol{c}_1 * \boldsymbol{c}_2$, where $\boldsymbol{c}_1, \boldsymbol{c}_2 \in C$ and $*$ denote the coordinate-wise product of vectors.

In our set-up, players P_1 and P_2 hold random n-element vectors $\boldsymbol{\alpha}_1$ and $\boldsymbol{\alpha}_2$, serving as (parts of the) keys for the MACs. A representation of $\boldsymbol{x}$ has the following form:

$$[\![\boldsymbol{x}]\!] = \big((C(\boldsymbol{x}_1), \boldsymbol{m}_1, \boldsymbol{\beta}_{x_2}), (C(\boldsymbol{x}_2), \boldsymbol{m}_2, \boldsymbol{\beta}_{x_1})\big)$$

where the first component is held by P_1 and the second by P_2. It should hold that $\boldsymbol{x} = \boldsymbol{x}_1 + \boldsymbol{x}_2$, that $\boldsymbol{m}_1 = MAC(\boldsymbol{x}_1) = \boldsymbol{\alpha}_2 * C(\boldsymbol{x}_1) + \boldsymbol{\beta}_{x_1}$, and by symmetry that $\boldsymbol{m}_2 = MAC(\boldsymbol{x}_2) = \boldsymbol{\alpha}_1 * C(\boldsymbol{x}_2) + \boldsymbol{\beta}_{x_2}$.

A representation can be opened if players exchange $C(\boldsymbol{x}_1), \boldsymbol{m}_1$ and $C(\boldsymbol{x}_2), \boldsymbol{m}_2$. P_1 checks that $C(\boldsymbol{x}_2)$ is indeed in C, and that $\boldsymbol{m}_2 = MAC(\boldsymbol{x}_2) = \boldsymbol{\alpha}_1 * \boldsymbol{x}_2 + \boldsymbol{\beta}_{x_2}$ holds; P_2 does the symmetric check on what he receives. Then both players can add the additive shares to get $C(\boldsymbol{x})$ and hence $\boldsymbol{x}$.

This way to open a representation is secure against a corrupt P_2 if $\boldsymbol{\beta}_{x_2}$ is uniformly chosen, independently for each representation since then P_2 has no a priory information on $\boldsymbol{\alpha}_1$. From this it is easy to see, using essentially the same argument as in [7], that P_1 will accept an incorrect value of $C(\boldsymbol{x}_2)$ with probability at most S^{-d} where S is the size of the field used and d is the minimum distance of C. The point is that P_2 wold have to switch to a different code word and hence change $C(\boldsymbol{x}_2)$ in at least d positions. But then he could only produce the required MAC value by guessing $\boldsymbol{\alpha}_1$ in d positions. Of course, a similar arguments works for a corrupt P_1.

It is trivial to verify that $[\![\boldsymbol{x}]\!] + [\![\boldsymbol{y}]\!] = [\![\boldsymbol{x} + \boldsymbol{y}]\!]$ where the +-symbol denotes that each player locally adds corresponding components he knows from the representations.

We can easily define a similar representation $[\![\boldsymbol{x}]\!]^*$, this is exactly the same as $[\![\boldsymbol{x}]\!]$, except that the code C^* is used for encoding. This will mean that the MACs can now be cheated with probability S^{-d^*} where d^* is the minimum distance of C^*. This is a potential problem since in general $d^* < d$, so we need to take care when we choose C.

Now, to multiply a value on the representation with a publicly known k-vector $\boldsymbol{u}$, $\boldsymbol{u}$ is turned into a codeword, $C(\boldsymbol{u})$ and then we define

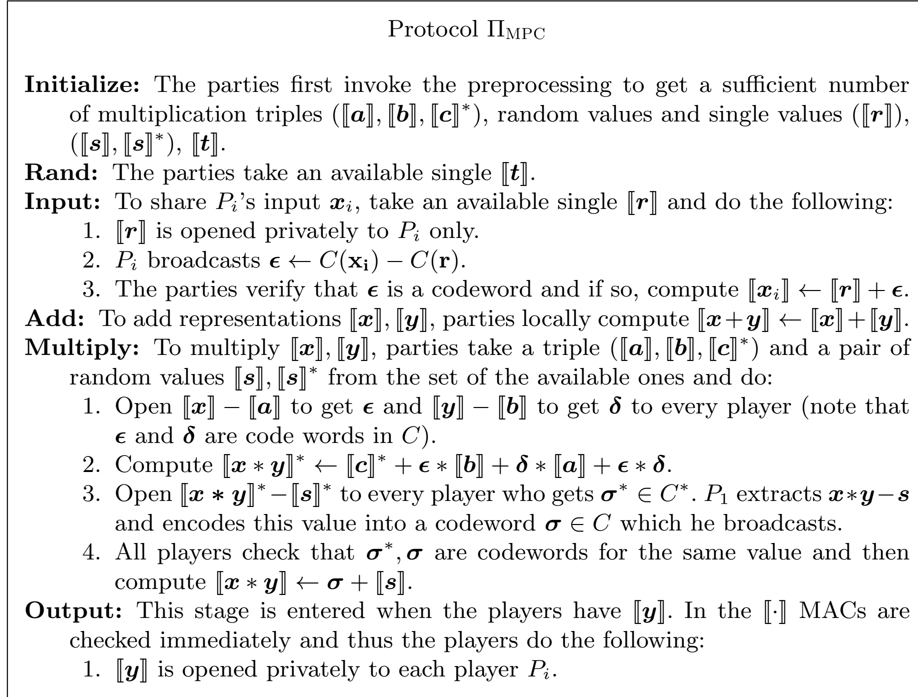
$$\begin{aligned} C(\boldsymbol{u}) * [\![\boldsymbol{x}]\!] = ((C(\boldsymbol{u}) * C(\boldsymbol{x}_1), C(\boldsymbol{u}) * \boldsymbol{m}_1, C(\boldsymbol{u}) * \boldsymbol{\beta}_{x_2}),\\ (C(\boldsymbol{u}) * C(\boldsymbol{x}_2), C(\boldsymbol{u}) * \boldsymbol{m}_2, C(\boldsymbol{u}) * \boldsymbol{\beta}_{x_1}). \end{aligned}$$

It is easy to see that this is indeed a well-formed representation of $\boldsymbol{u} * \boldsymbol{x}$, however, using the code C^*, so we can write this as $C(\boldsymbol{u}) * [\![\boldsymbol{x}]\!] = [\![\boldsymbol{u} * \boldsymbol{x}]\!]^*$.

We can also add a public constant $\boldsymbol{u}$ to a representation, namely we define

$$C(\boldsymbol{u}) + [\![\boldsymbol{x}]\!] = \big((C(\boldsymbol{u}) + C(\boldsymbol{x}_1), \boldsymbol{m}_1, \boldsymbol{\beta}_{x_2}), (C(\boldsymbol{x}_2), \boldsymbol{m}_2, \boldsymbol{\beta}_{x_1} - \boldsymbol{\alpha}_2 * \boldsymbol{u})\big) = [\![\boldsymbol{u} + \boldsymbol{x}]\!].$$

The protocol Π_{MPC} using this representation and its linear properties is described in Figure 2. In the original protocol [7] there was also a sub-protocol for permuting entries internally in the represented vectors. However, since we only want to do several instance of one computation in parallel, we do not need this step.



Protocol Π_{MPC}

Initialize: The parties first invoke the preprocessing to get a sufficient number of multiplication triples $([\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!], [\![\boldsymbol{c}]\!]^*)$, random values and single values $([\![\boldsymbol{r}]\!])$, $([\![\boldsymbol{s}]\!], [\![\boldsymbol{s}]\!]^*)$, $[\![\boldsymbol{t}]\!]$.

Rand: The parties take an available single $[\![\boldsymbol{t}]\!]$.

Input: To share P_i's input $\boldsymbol{x}_i$, take an available single $[\![\boldsymbol{r}]\!]$ and do the following:
1. $[\![\boldsymbol{r}]\!]$ is opened privately to P_i only.
2. P_i broadcasts $\boldsymbol{\epsilon} \leftarrow C(\mathbf{x_i}) - C(\mathbf{r})$.
3. The parties verify that $\boldsymbol{\epsilon}$ is a codeword and if so, compute $[\![\boldsymbol{x}_i]\!] \leftarrow [\![\boldsymbol{r}]\!] + \boldsymbol{\epsilon}$.

Add: To add representations $[\![\boldsymbol{x}]\!], [\![\boldsymbol{y}]\!]$, parties locally compute $[\![\boldsymbol{x}+\boldsymbol{y}]\!] \leftarrow [\![\boldsymbol{x}]\!] + [\![\boldsymbol{y}]\!]$.

Multiply: To multiply $[\![\boldsymbol{x}]\!], [\![\boldsymbol{y}]\!]$, parties take a triple $([\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!], [\![\boldsymbol{c}]\!]^*)$ and a pair of random values $[\![\boldsymbol{s}]\!], [\![\boldsymbol{s}]\!]^*$ from the set of the available ones and do:
1. Open $[\![\boldsymbol{x}]\!] - [\![\boldsymbol{a}]\!]$ to get $\boldsymbol{\epsilon}$ and $[\![\boldsymbol{y}]\!] - [\![\boldsymbol{b}]\!]$ to get $\boldsymbol{\delta}$ to every player (note that $\boldsymbol{\epsilon}$ and $\boldsymbol{\delta}$ are code words in C).
2. Compute $[\![\boldsymbol{x} * \boldsymbol{y}]\!]^* \leftarrow [\![\boldsymbol{c}]\!]^* + \boldsymbol{\epsilon} * [\![\boldsymbol{b}]\!] + \boldsymbol{\delta} * [\![\boldsymbol{a}]\!] + \boldsymbol{\epsilon} * \boldsymbol{\delta}$.
3. Open $[\![\boldsymbol{x} * \boldsymbol{y}]\!]^* - [\![\boldsymbol{s}]\!]^*$ to every player who gets $\boldsymbol{\sigma}^* \in C^*$. P_1 extracts $\boldsymbol{x} * \boldsymbol{y} - \boldsymbol{s}$ and encodes this value into a codeword $\boldsymbol{\sigma} \in C$ which he broadcasts.
4. All players check that $\boldsymbol{\sigma}^*, \boldsymbol{\sigma}$ are codewords for the same value and then compute $[\![\boldsymbol{x} * \boldsymbol{y}]\!] \leftarrow \boldsymbol{\sigma} + [\![\boldsymbol{s}]\!]$.

Output: This stage is entered when the players have $[\![\boldsymbol{y}]\!]$. In the $[\![\cdot]\!]$ MACs are checked immediately and thus the players do the following:
1. $[\![\boldsymbol{y}]\!]$ is opened privately to each player P_i.

Fig. 2: The online protocol.

2.1 Reed-Solomon codes

MiniMac[7] requires that the code C and its Schur transform C^* are systematic. In this section we recall how Reed-Solomon works, what it means for a code to be systematic and how it is achieved for Reed-Solomon which is not systematic out of the box.

A Reed-Solomon code is an error correcting code described by three parameters: (n, k, d) where n is the length of the code, k is the dimension (the length of messages that can be encoded) and d the distance of the code (the amount

of redundancy). There are two algorithms, one for encoding a message into a codeword and one for checking a codeword is valid.
To *encode* a message of k field elements say $\boldsymbol{a} = [a_0, ..., a_{k-1}]$ as a Reed-Solomon codeword we consider the polynomial $f_{\boldsymbol{a}}(x) = a_0 + a_1 x + \cdots + a_{k-1}x^{k-1}$ of degree $k-1$. For $n \geq k$ distinct points $v_j, j = 0, \ldots, n-1$ we define the vector $\boldsymbol{c} = (f_{\boldsymbol{a}}(v_0), \ldots, f_{\boldsymbol{a}}(v_{n-1}))$ to be a codeword for $\boldsymbol{a}$. Evaluating a polynomial $f_{\boldsymbol{a}}$ in n distinct points is strongly connected to n by k Vandermonde matrices, $V_{n\times k}$. Encoding a polynomial $f_{\boldsymbol{a}}$ corresponds to multiplying such a Vandermonde matrix with the column vector $\boldsymbol{a} = (a_0, ..., a_{k-1})^T$ as follows:

$$V_{n\times k} \cdot \boldsymbol{a} = \begin{bmatrix} v_0^0 & v_0 & \cdots & v_0^{k-1} \\ v_1^0 & v_1 & \cdots & v_1^{k-1} \\ & \vdots & & \\ v_n^0 & v_n & \cdots & v_n^{k-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{k-1} \end{bmatrix} =$$

$$\begin{bmatrix} a_0 + a_1 v_0 + \cdots + a_{k-1} v_0^{k-1} \\ a_0 + a_1 v_1 + \cdots + a_{k-1} v_1^{k-1} \\ \vdots \\ a_0 + a_1 + v_n + \cdots + a_{k-1} v_n^{k-1} \end{bmatrix} = \begin{bmatrix} f_{\boldsymbol{a}}(v_0) \\ f_{\boldsymbol{a}}(v_1) \\ \vdots \\ f_{\boldsymbol{a}}(v_n) \end{bmatrix}$$

Observe that from k points from a codeword we can find our message $\boldsymbol{a}$ (the coefficient of $f_{\boldsymbol{a}}$) again using interpolation. Thus, one naive way to *validate* a given codeword, $\boldsymbol{c}$ is to go through each of the 2^d subsets of k elements from $\boldsymbol{c}$ and check that each interpolate to the same message $\boldsymbol{a}$.
As noted in the beginning of this section this way of encoding messages is not systematic in the following sense: A *systematic* code is a code where the encoded message directly appears in fixed positions of the codeword. In our case we will have the first k positions of a codeword to be the actual encoded message.
To encode a systematic Reed-Solomon codeword we fix an encoding matrix $V_{n\times k}$ where n is the desired codeword length and k the length of our messages. Then we take the upper $k \times k$ matrix $V_{k\times k}$ of $V_{n\times k}$ which by construction is guaranteed to be invertable and do the following:

$$V_{n\times k}(V_{k\times k})^{-1}\boldsymbol{a}^T = \boldsymbol{c} = \begin{bmatrix} c_0 = a_0 \\ \vdots \\ c_{k-1} = a_{k-1} \\ c_k \\ \vdots \\ c_{n-1} \end{bmatrix}$$

We take $E = V_{n\times k}(V_{k\times k})^{-1}$ to define our encoding matrix. Notice that the first k rows of E will yield the $k \times k$ identity matrix, ensuring systematic codewords as desired.

Now it is easy to check whether a vector is a codeword, namely given $\boldsymbol{c}$ we multiply E with the column vector consisting of the first k entries in $\boldsymbol{c}$, and check the result yields $\boldsymbol{c}$ again e.g. $E[c_0, ..., c_{k-1}]^T \stackrel{?}{=} \boldsymbol{c}$. If not the codeword is invalid.
As noted in the introduction, $\mathbb{F}_{2^8}$ is a natural choice as an underlying field. In particular as efficient encoding is important for MiniMac, we benefit from $\mathbb{F}_{2^8}$ being small enough that multiplication can be done by table look-up. With respect to Reed-Solomon the choice of $\mathbb{F}_{2^8}$ introduces some restrictions on the parameters (n, k, d). Namely, that as there are only 256 elements, the length of a (checkable) Reed-Solomon code can be at most 256, as we need a distinct evaluation point for each codeword position. Furthermore, the error probability for the MACs is at most 256^{-d^*}; aiming for 128-bit security implies a minimum distance of (at least) 16 field elements for C^*. We can summarize the restrictions we get on the Reed-Solomon code parameters as follows:

- $n \leq |\mathbb{F}_{2^8}| = 256$ - The length of the codewords can be at most 256.
- $d' = min(d, d^*) \geq 16$ - Altering a message to another codeword implies modifying d' positions.
- $k \leq (n - d')/2 + 1$ - If C has dimension k it is generated from polynomials of degree $k - 1$, so the Schur transform is generated from polynomials of degree $2(k - 1)$ and hence has minimum distance $d^* = n - 2(k - 1)$. For $d^* = 16$ we get $k \leq 121$ when $n = 256$. We have chosen to round this to $k = 120$ as this slightly simplifies implementation.

2.2 Preprocessing

We will not explicitly consider the preprocessing in this paper. Note, however, that since MACs are set up as in [9] and our field is characteristic 2, it is straightforward to modify the TinyOT preprocessing to obtain the one presently needed.

3 Evaluation and Comparison with TinyOT

In this section we evaluate MiniMac in practice. To put our work in perspective we wish to compare MiniMac to the previous state of the art protocol, TinyOT, in [9]. We present performance numbers for AES encryption for both protocols in our framework and their interpretation.

3.1 Empirical setup and performance measurements

In the following empirical study of TinyOT and MiniMac, we use the same two computers on an Aarhus University's network. Both machines have the following specifications:
In many of our experiments we start multiple processes on the two test machines that pair-wise execute the protocol to maximize CPU and network utilization. When more pairs of processes are executing the protocol a third machine, the

CPU	Intel(R) Xeon(R) CPU X3430 @ 2.40 GHz
CPU op-mode(s)	32-bit, 64-bit
CPU(s)	4
Thread(s) per core	1
CPU MHz	2393.859
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	8192K
RAM	32 GB
Net	Gigabit LAN with 0.215 ms avg. latency measured using the ping tool.

Fig. 3: Hardware spec.

Monitor, listens for all of them to report ready. When all processes are ready the Monitor broadcasts a "Go!" signal and records the time. When the pairs of processes running the protocol reports back to the Monitor that they are done the Monitor records the time of the *last* process. These two numbers are recorded as the start and end time of the experiment and their difference is taken to be the measured elapsed time of the experiment.

In summary performance numbers are presented in tables with these columns:

Instances This column report how many protocol instances are run in parallel. That is, the number of processes started on each of the two test machines pair-wise carrying out the protocol.

Total (ms) This column reports on the total execution time in milliseconds. For one instance, this is also the latency, e.g. the time from initiating computation until some result is ready.

No AES This reports the total number of single block 128-bit AES encryptions carried out.

Time per AES (ms) This column shows the amortized time in milliseconds used per AES circuit.

To even out fluctuations, every line in tables like the one below reports on the mean time from at least five runs. When appropriate, we present results with in a 95% confidence interval, indicated as $35 \pm 8\ ms$ for an experiment with mean 35 with a 95% confidence interval in $[27; 43]\ ms$.

Instances	**No AES**	**Total Time (ms)**	**Time per AES (ms)**
8	102000	960	380 $\pm 69\ ms$

Fig. 4: An example performance measurement, these numbers are made up.

3.2 The benchmark - AES circuit and relevance

We run our experiments with a binary AES circuit encrypting one block of plaintext with a 128 bits key. The key is additively secret shared between the two test machines. The plain-text to encrypt is publicly known as well as the circuit gates. Both test machines learns the cipher-text but learns nothing new about the key. There are several practical scenarios where key material split between several servers might mitigate a potential compromise of one server.
Our concrete circuit has 6800 AND-gates and 26816 XOR-gates.
Throughout all experiments MiniMac is run with at least 128-bit security.

3.3 MiniMac test runs with no optimizations

In this Section we run our implentation with all optmizations and tricks turned off to present a baseline of performance. This baseline is compared to TinyOT also implemented in our framework.
To only measure the AES circuit we exclude input and output gates. Thus when running our experiments all MiniMac processes run to the point where all parties has completed their input gates. Likewise, when the last XOR-gates of the circuit is computed the computation is recorded as complete before running any of the output gates.

Instances	No AES	Total Time (ms)	Time per AES (ms)
1	120	161651	1347 ± 1243
2	240	128283	644 ± 578
4	480	188824	393 ± 138
8	960	264596	144 ± 233
16	1920	598799	311 ± 108

Fig. 5: Running MiniMac without tricks and optimizations.

3.4 Introduction to TinyOT

The protocol in [9] is nicked named TinyOT. MiniMac and TinyOT are both targeted at evaluating binary circuits. TinyOT uses Oblivious Transfer to obtain multiplication triples as described in [1].
First a short piece of history about TinyOTs implementation. A great deal of work was put in to implementing of TinyOT in Java for the publication and the performance-numbers presented in [9]. The best times achieved there were on a gigantic circuit doing 16384 AES encryptions in parallel. With this set-up, they achieved 32 ms amortised time per AES block. This circuit was not available to us here as it was custom built into a circuit generator written in Java.
Later Nielsen et. al. did experimentation with implementing TinyOT in C++. This later C++ implementation has been adapted to our set-up in plain C.

This allows us to measure TinyOT performance on a single AES circuit in our framework. It is faster than the Java implementation used in [9] for one instance, probably because of the cost involved in starting up a Java process.

3.5 Empirical results with TinyOT

The table below presents TinyOT runs on the test circuit in our setup. To get some parallelism we try to run several TinyOT processes. From the numbers it is evident that running multiple TinyOT processes in parallel actually hurts performance making each AES circuit slower. The explanation for this is that TinyOT exhausts the CPU resource on our test machines, probably in part because the administration involved in running several separate processes hurts performance.

Instances	**No AES**	**Total Time (ms)**	**Time per AES (ms)**
1	1	1079	1079 ± 251
2	2	1414	707 ± 183
4	4	4740	1185 ± 229
8	8	10451	1306 ± 187
16	16	41998	2624 ± 261

Fig. 6: Running TinyOT (the C version) on one instance of our AES circuit.

4 MiniTrix

In this section we boost MiniMac in a number of ways.

4.1 Making the protocol symmetric for Multiplications

In MiniMac one player has a special role when transforming $[\![\boldsymbol{\sigma}]\!]^*$ back to $[\![\boldsymbol{\sigma}]\!]$ where all players wait on one player to do the re-encoding of $\boldsymbol{\sigma}$ as codeword in C. This happens both when multiplying with a public constant and when multiplying two secret values. This step, transforming a C^* codeword to C requires communication in the order of $2N$-codeword for N players.
With this trick we make the protocol symmetric, by opening $[\![\boldsymbol{\sigma}]\!]^*$ among all players. Now each party in parallel can extract $\langle\sigma\rangle$ and reencode it in C. This increases the overall communication complexity from $2N$ to N^2, however for small N and in particular in the two party case this makes no difference, and evens out the workload for the parties. However, we did not see any significant impact of this optimization.

4.2 Use Fast Fourier Transform for encoding

The naive MiniMac implementation spends most of its CPU-time in matrix by vector multiplications during the encoding of codewords.
Naive matrix by vector multiplication is quadratic in the length of the codeword, n, and hence there is a lot to gain by reducing this using the Fast Fourier Transform (FFT[4,3]).
In its basic form, FFT can be thought of as a recursive algorithm for multiplying a vector by an $n \times n$ Vandermonde matrix M, or its inverse. In order for this to work, the matrix must contain powers of an n'th root of unity. More specifically, starting indices from 0, the (i, j)'th entry in the matrix should be ω^{ij}, where $\omega^n = 1$. We call this matrix M_ω. If we think of a vector $\boldsymbol{x}$ as containing coefficients of a polynomial of degree (at most) $n-1$, then computing $M_\omega \boldsymbol{x}$ will evaluate the polynomial in the points $\omega^0, \omega^1, \ldots, \omega^{n-1}$, it can therefore be used for Reed-Solomon encoding.
FFT works by breaking the problem into two instance of size $n/2$ each, and in order for this to work recursively, it is usually assumed that n is a two-power. In this case, the algorithm takes time $O(n \log n)$ field multiplications. However, even if n is not a two-power but factors into smaller primes, variants of the algorithm can still be used to break the problem into smaller pieces and improve performance.
In our field $\mathbb{F}_{2^8}$, the multiplicative group has order 255, which factors into $3 \cdot 5 \cdot 17$. The field therefore contains a root of unity ω of order 255, which means that ω^3 is a root of unity of order 85. This allows us to use a Reed-Solomon code of length 255 and dimension 85, where all the required operations can be done using (a variant of) FFT:
Systematic encoding can be done by multiplying the data vector $\boldsymbol{x}$ by the 85×85 matrix $M_{\omega^3}^{-1}$ to get $\boldsymbol{y}$ containing coefficients of a polynomial that evaluates to the coordinates of $\boldsymbol{x}$ in the points $(\omega^3)^i,\ i = 0...84$. Then we compute $M_\omega \boldsymbol{y}$ to get the full codeword. Note that the protocol only requires to encode in C, not in C^*. We can test if $\boldsymbol{c}$ is in C (or C^*) by computing $M_\omega^{-1}\boldsymbol{c}$. This should reconstruct the polynomial underlying the purported codeword, and then we simply test if the degree is low enough.
Even using one level of recursion in FFT reduces significantly the work we need for these operations. Because 17 divides the size of both matrices, we can break both problems in 17 smaller pieces, and for multiplication by M_ω, for instance, this reduces the work from $256^2 = 65536$ field multiplications to $255 * 15 + 255 * 17 = 8160$.

4.3 Preprocessing dedicated for Binary circuits

Minimac provides secure field arithmetic, and basing everything on $\mathbb{F}_2$ directly seems natural if the overall goal is secure Boolean circuit evaluation. However, constructing *binary* codes with the right properties is non-trivial as explained earlier. In this paper we solved this issue by using codes over $\mathbb{F}_{2^8}$. But this means that, although every data vector we encode will have binary entries, positions

Instances	No AES	Total Time (ms)	Time per AES (ms)
1	85	53699	631
2	170	59505	424
4	340	79092	207
8	680	107512	146
16	1360	257609	115

Fig. 7: Running a varying number MiniMac instances using FFT encoding.

in the resulting codeword will be bytes where only the least significant position contains data; all other bit positions are identically 0. It is natural to ask if we can exploit all eight positions in each data byte. Thus, the goal here will be to evaluate not k, but $8k$ ($\log |\mathbb{F}| \cdot k$ in general) identical circuits in parallel, using the same code as before and therefore (ideally) at the same cost.

We can certainly encode a vector $\boldsymbol{x}$ that contains data in all bit positions, to get $[\![\boldsymbol{x}]\!]$, and our addition $[\![\boldsymbol{x}]\!]+[\![\boldsymbol{y}]\!] = [\![\boldsymbol{x}+\boldsymbol{y}]\!]$ indeed implements $8k$ XOR operations in parallel, simply because the addition in $\mathbb{F}_{2^8}$ is the same as bit-wise XOR on bytes.

This means that we just need to redesign the multiplication protocol so that from $[\![\boldsymbol{x}]\!], [\![\boldsymbol{y}]\!]$, we can compute $[\![\boldsymbol{x} \wedge \boldsymbol{y}]\!]$, where for $\boldsymbol{x} = (x_1, ..., x_k), \boldsymbol{y} = (y_1, ..., y_k)$ we define $[\![\boldsymbol{x} \wedge \boldsymbol{y}]\!] = (x_1 \wedge y_1, ..., x_k \wedge y_k)$, and where $x_i \wedge y_i$ denotes bit-wise AND on bytes.

To get started, let us recall the MiniMac multiplication protocol (since we are in characteristic 2, we simplify some expressions by replacing some minus signs by plus).

To multiply $[\![\boldsymbol{x}]\!], [\![\boldsymbol{y}]\!]$, parties take a triple $([\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!], [\![\boldsymbol{c}]\!]^*)$ and a pair of random values $[\![\boldsymbol{s}]\!], [\![\boldsymbol{s}]\!]^*$ from the set of the available ones and do:

1. Open $[\![\boldsymbol{x}]\!] + [\![\boldsymbol{a}]\!]$ to get $\boldsymbol{\epsilon}$ and $[\![\boldsymbol{y}]\!] + [\![\boldsymbol{b}]\!]$ to get $\boldsymbol{\delta}$ to every player.
2. Compute $[\![\boldsymbol{x} * \boldsymbol{y}]\!]^* \leftarrow [\![\boldsymbol{c}]\!]^* + \boldsymbol{\epsilon} * [\![\boldsymbol{b}]\!] + \boldsymbol{\delta} * [\![\boldsymbol{a}]\!] + \boldsymbol{\epsilon} * \boldsymbol{\delta}$ to every player.
3. Open $[\![\boldsymbol{x} * \boldsymbol{y}]\!]^* + [\![\boldsymbol{s}]\!]^*$ to every player who gets $\boldsymbol{\sigma}^* \in C^*$. P_1 extracts $\boldsymbol{x} * \boldsymbol{y} + \boldsymbol{s}$ and encodes this value into a codeword $\boldsymbol{\sigma} \in C$ which he broadcasts.
4. All players check that $\boldsymbol{\sigma}^*, \boldsymbol{\sigma}$ are codewords for the same value and then compute $[\![\boldsymbol{x} * \boldsymbol{y}]\!] \leftarrow \boldsymbol{\sigma} + [\![\boldsymbol{s}]\!]$.

Note here that $\boldsymbol{\epsilon}$ and $\boldsymbol{\delta}$ are codewords. In general for $\boldsymbol{c} \in C$ we will let $C^{-1}(\boldsymbol{c})$ denote the k-vector that $\boldsymbol{c}$ encodes. The reader should notice that the reason why step 2 above works is that $C^{-1}(\boldsymbol{\epsilon}) = \boldsymbol{x} + \boldsymbol{a}$ and $C^{-1}(\boldsymbol{\delta}) = \boldsymbol{y} + \boldsymbol{b}$.

Now, the idea is to change the computation in step 2. above. Instead we will compute:

$$[\![\boldsymbol{x} \wedge \boldsymbol{y}]\!]^* = [\![C^{-1}(\boldsymbol{\delta}) \wedge C^{-1}(\boldsymbol{\epsilon})]\!] + [\![C^{-1}(\boldsymbol{\epsilon}) \wedge \boldsymbol{a}]\!]^* + [\![C^{-1}(\boldsymbol{\delta}) \wedge \boldsymbol{b}]\!]^* + [\![\boldsymbol{a} \wedge \boldsymbol{b}]\!]^*$$

It is easy to see that this equation is true, simply because the $\wedge$ operation distributes over bit-wise XOR just like the $*$-operation does. Essentially, we are simply using Beaver triples for $\mathbb{F}_2$, where the product is stored as a $\mathbb{F}_{2^8}$-element. Note that it makes sense to add $[\![]\!]$ and $[\![]\!]^*$-representations since our C

is contained in C^*. Therefore $[\![C^{-1}(\boldsymbol{\delta}) \wedge C^{-1}(\boldsymbol{\epsilon})]\!]$ is also a $[\![]\!]^*$-representation of the same vector (albeit not a random such representation).
Now we need to figure out how we can compute the 4 terms on the right-hand side. The last term can be obtained by requiring that the preprocessing supplies us with $[\![\boldsymbol{c}]\!]^* = [\![\boldsymbol{a} \wedge \boldsymbol{b}]\!]^*$. Also, the first term can be computed on public values by all parties.
The "mixed terms" $[\![C^{-1}(\boldsymbol{\epsilon}) \wedge \boldsymbol{a}]\!]^*, [\![C^{-1}(\boldsymbol{\delta}) \wedge \boldsymbol{b}]\!]^*$ constitute a challenge. To compute these, the multiplication triples from the preprocessing phase are extended so that in addition to $[\![\boldsymbol{a}]\!]$ and $[\![\boldsymbol{b}]\!]$, the "bit decomposition" of $\boldsymbol{a}$ and $\boldsymbol{b}$ is also provided.
To explain what this means we need some notation. Say $\boldsymbol{a} = (a_1, ..., a_k)$ where each $a_j \in \mathbb{F}_{2^8}$. Let $(a_j)_i$ be the byte that equals a_j in the i'th bit position and is 0 elsewhere. Furthermore, $(a_j)_{\downarrow i}$ denotes $(a_j)_i$ shifted down $i-1$ positions, so that the "important" bit is in the least significant position. Then we define $\boldsymbol{a}_i = ((a_1)_i, ..., (a_k)_i)$ and $\boldsymbol{a}_{\downarrow i} = ((a_1)_{\downarrow i} ...., (a_k)_{\downarrow i})$.
The assumption on the preprocessing now is that we that we are given $[\![\boldsymbol{a_i}]\!], [\![\boldsymbol{b_i}]\!]$ for $i = 0...7$.
One can now observe that we can compute the missing terms as follows:

$$\sum_{i=0}^{7} C(C^{-1}(\boldsymbol{\delta})_{\downarrow i}) * [\![\boldsymbol{a_i}]\!] = \sum_{i=0}^{7} [\![C^{-1}(\boldsymbol{\delta})_{\downarrow i} * \boldsymbol{a_i}]\!]^* = [\![C^{-1}(\boldsymbol{\delta}) \wedge \boldsymbol{a}]\!]^*$$

Note that we multiply by the public constant $C(C^{-1}(\boldsymbol{\delta})_{\downarrow i})$ using the normal MiniMac procedure that works over $\mathbb{F}_{2^8}$. Hence the result will indeed be a vector in the $[\![]\!]^*$-representation. That the vector inside the representation is indeed $C^{-1}(\boldsymbol{\delta}) \wedge \boldsymbol{a}$ can be seen from the fact that each byte in $C^{-1}(\boldsymbol{\delta})_{\downarrow i}$ has the value 0 or 1, depending on the value of the corresponding bits from $C^{-1}(\boldsymbol{\delta})$. It therefore acts as a "selector" that decides whether to include the bits from $\boldsymbol{a}_i$.
A final modification concerns that last two steps in the original protocol, where the result is converted from $[\![]\!]^*$ to $[\![]\!]$-representation. This costs communication that we would like to avoid. To do this, consider what happens if we simply omit this conversion. This would mean that data would be passed from one gate to another using the $[\![]\!]^*$-representation instead. This presents no problem for doing linear computation, as the $[\![]\!]^*$-representation is also linear.
For the multiplication protocol to take input in the $[\![]\!]^*$-representation, we can just modify the preprocessing so that it would supply $[\![\boldsymbol{a}]\!]^*, [\![\boldsymbol{b}]\!]^*$ in stead of $[\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!]$. The only effect of this is that then $\boldsymbol{\epsilon}, \boldsymbol{\delta}$ will be C^*-codewords, but this has no effect on the rest of the protocol, since we anyway need to decode them and re-encode the bits in C; thus, C^{*-1} will simply denote the extraction of the first k data-entries of a C^* codeword.
To summarize, the multiplication triples have been replaced by a different set of data, namely

$$[\![\boldsymbol{a}]\!]^*,\ [\![\boldsymbol{b}]\!]^*,\ [\![\boldsymbol{a} \wedge \boldsymbol{b}]\!]^*,\ \{[\![\boldsymbol{a}_i]\!] |\ i = 0...7\},\ \{[\![\boldsymbol{b}_i]\!] |\ i = 0...7\}$$

The multiplication protocol works as follows:

1. Open $[\![\boldsymbol{x}]\!]^* + [\![\boldsymbol{a}]\!]^*$ to get $\boldsymbol{\epsilon}$ and $[\![\boldsymbol{y}]\!]^* + [\![\boldsymbol{b}]\!]^*$ to get $\boldsymbol{\delta}$, for every player. The following steps are then done using local operations only.
2. Compute $[\![{C^*}^{-1}(\boldsymbol{\epsilon}) \wedge {C^*}^{-1}(\boldsymbol{\delta})]\!]$
3. Compute $\sum_{i=0}^{7} C({C^*}^{-1}(\boldsymbol{\delta})_{\downarrow i}) * [\![\boldsymbol{a_i}]\!] = [\![{C^*}^{-1}(\boldsymbol{\delta}) \wedge \boldsymbol{a}]\!]^*$.
4. Compute $\sum_{i=0}^{7} C({C^*}^{-1}(\boldsymbol{\epsilon})_{\downarrow i}) * [\![\boldsymbol{b_i}]\!] = [\![{C^*}^{-1}(\boldsymbol{\epsilon}) \wedge \boldsymbol{b}]\!]^*$.
5. Compute $[\![{C^*}^{-1}(\boldsymbol{\delta}) \wedge {C^*}^{-1}(\boldsymbol{\epsilon})]\!] + [\![{C^*}^{-1}(\boldsymbol{\epsilon}) \wedge \boldsymbol{a}]\!]^* + [\![{C^*}^{-1}(\boldsymbol{\delta}) \wedge \boldsymbol{b}]\!]^* + [\![\boldsymbol{a} \wedge \boldsymbol{b}]\!]^* = [\![\boldsymbol{x} \wedge \boldsymbol{y}]\!]^*$

Theoretical Analysis of the Binary Preprocessing It is easy to see that the above methods generalizes to any field of characteristic 2. In general the field should be of size $O(\log(n))$ to allow the use of Reed-Solomon codes and hence allow for efficient encoding using FFT.

Following [7], we compute the cost of the protocol per player per gate in the circuit we compute. By simple inspection, one sees that the storage needed from the preprocessing is $O(\log(n))$ (namely $O(1)$ field elements). The communication is $O(1)$ bits because we only do 2 openings in each multiplication protocol, and each such protocol does $\log(n)$ AND gates. Finally the computational work is $O(\text{polylog}(n))$ bit operations due to the use of FFT. As explained in the Introduction, this improves on the original MiniMac for communication and computational work. We also save one round of communication.

Experimentation and Performance numbers In the above we actually present two tricks. The first trick allows us to utilize all eight bits of each field element. We call this the *bit-packing trick*. In the second trick we reduce the communication complexity of the protocol by removing the down conversion from codewords in the Schur transform to codewords in C, we call this the *zero-degree constants* trick because it takes advantage of the fact that all constants in our circuit is the same value repeated many times and thus the underlying encoding polynomial must be constant.

Instances	**No AES**	**Total Time (ms)**	**Time per AES (ms)**
1	960	58127	60 ± 2
2	1920	104840	76 ± 6
4	3840	224146	58 ± 2
8	7680	591523	77 ± 7
16	15360	1094454	71 ± 6

Fig. 8: Running MiniMac with bit-packing-trick only, with standard matrix encoding

Instances	No AES	Total Time (ms)	Time per AES (ms)
1	680	19410	24 ± 4
2	1360	36235	26 ± 3
4	2720	71082	24 ± 8
8	5440	196322	34 ± 30
16	10880	428458	39 ± 1

Fig. 9: Running MiniMac with bit-packing-trick only, with Fast Fourier Transform encoding

4.4 Simultaneous multiplication gates

The present optimization is based on the observation that network utilization is much higher for large batches of data. Thus, collecting a larger amount of data before communication should give better performance.
In the previous sections our MiniMac implementation was general and works even if the circuit is streamed, e.g. without any knowledge of what is coming ahead. If our circuit description is extended with hints about which gates can be run simultaneously, then the protocol can be made to run faster. In particular, we extend our implementation to use such hints to compute blocks of multiplication (AND) gates. The hints that we allow in the circuit description describe how many of the following multiplications are independent (e.g. the number of AND gates in the following that will not read the results of each other).
Lets us recap the communication pattern in MiniMac during a multiplication from one players pointer of view:

1 The codewords for $\boldsymbol{\delta}$ and $\boldsymbol{\epsilon}$ are sent to every other player along with their MACs. This requires $4*n*N$ bytes of communication. Where n is the length of a codeword and N is the number of players.
2 Receive $\boldsymbol{\delta}$ and $\boldsymbol{\epsilon}$ with MACs from every other player.
3 Local computation, including checking the incoming MACs.

The idea is to collect say M independent AND gates before initiating communication. Then on the M'th gate the peers exchange M $\boldsymbol{\delta}$- and $\boldsymbol{\epsilon}$-values with MACs.

Instances	No AES	Total Time (ms)	Time per AES (ms)
1	960	55917	58 ± 20
2	1920	95321	70 ± 56
4	3840	205667	85 ± 148
8	7680	334529	43 ± 24
16	15360	282940	42 ± 20

Fig. 10: Running MiniMac with bit-packing-trick and simultaneous multiplication trick, with matrix encoding

Instances	No AES	Total Time (ms)	Time per AES (ms)
1	680	14313	21 ± 2
2	1360	27893	28 ± 22
4	2720	69452	25 ± 4
8	5440	140540	25 ± 6
16	10880	257556	23 ± 3

Fig. 11: Running MiniMac with bit-packing-trick and simultaneous multiplication trick, with FFT encoding.

4.5 Fast encoding of binary data

One should notice that in the above multiplication protocol we need to encode $C^{*-1}(\boldsymbol{\epsilon})_{\downarrow i}$ for $i = 0...7$. However, each entry in these vectors is 0 or 1, due to the shift down we did. Therefore, we can encode much faster than in general by simply XOR-ing together those rows of the generator matrix that correspond to positions in the vector that are 1. Packing bytes together in word-size chunk allows us to do this on several bytes using one CPU instruction.

Instances	No AES	Total Time (ms)	Time per AES (ms)
1	960	18060	18 ± 1
2	1920	37614	19 ± 6
4	3840	118878	28 ±10
8	7680	252111	18 ±10
16	15360	304846	19 ±7

Fig. 12: Running MiniMac with bit-packing-trick, zero-degree-constants and simultaneous multiplication trick, with matrix bit encoding trick.

Instances	No AES	Total Time (ms)	Time per AES (ms)
1	680	9962	14 ± 1
2	1360	17553	14 ± 5
4	2720	39886	15 ± 5
16	10880	195992	18 ± 7

Fig. 13: Running MiniMac with bit-packing-trick, zero-degree-constants and simultaneous multiplication trick, with FFT bit encoding trick.

4.6 Final Optimizations

Just before the deadline for this version of the paper, we did some final experiments that greatly improved the performance. First, we observed from profiling

the program that a lot of time was spent waiting for data to arrive on the communication line. In other words, both players had idle time we should be able to exploit. We therefore tried starting 8 copies of our original process with a time interval of 200ms in between them, hoping to allow some instances to compute while others were waiting. This improved the time per AES instance to about 9 ms (using 8 copies of the program seemed experimentally to be the best choice). Finally, we experimented with compiler flags and found a setting that allowed better exploitation of the CPU. This gave another factor of about 2, so that we end up with about 4 ms per AES.

5 Conclusion

We have proposed several optimisations of the MiniMac protocol, both on implementation and protocol level. In the fastest configuration of MiniMac using Fast Fourier transform and bit encoding trick with simultaneous multiplications gates we evaluate an AES circuit in 4 *ms* on our test setup. As far as we are aware, this is the fastest published actively secure two-party implementation of AES with 128-bit security based on a Boolean circuit.

6 Future directions

All experiments in this paper are run on the same AES circuit. To improve performance one may try to optimize the AES circuit, for instance by reducing further the number of AND gates. Since a lot of the description of AES uses arithmetic over $\mathbb{F}_{2^8}$ it seems natural to try an arithmetic circuit over $\mathbb{F}_{2^8}$. However, the circuits of this type that we know of are all significantly larger than the binary circuit we use here. However, it is likely that adding the bit decomposition trick used in [8] could help here, so this seems an obvious direction to try in future work. However, AES leads to very specialized circuits, so it natural to broaden the scope and consider other binary circuits, for instance for hash functions such as SHA-1.
During our work with MiniMac many Operating System dependent obstacles has been identified. In particular the handling of when the TCP actually sends data. It is possible that tweaking Linux network stack parameters, and/or other strategies for better scheduling may increase throughput further.

7 Acknowledgements

We would like to thank Nigel Smart, Stefan Tillich and their crew at Bristol for providing a selection of excellent circuits at `http://www.cs.bris.ac.uk/Research/CryptographySecurity/MPC/`. Also we would like to thank Jesper Buus Nielsen for providing source code for the TinyOT C++ implementation.

References

1. Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, pages 420–432, 1991.
2. Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT*, pages 169–188, 2011.
3. S.D. Conte and C. De Boor. *Elementary Numerical Analysis: An Algorithmic Approach*. International series in pure and applied mathematics. McGraw-Hill, 1980.
4. James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Math. comput*, 19(90):297–301, 1965.
5. Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure mpc for dishonest majority - or: Breaking the spdz limits. In *ESORICS*, pages 1–18, 2013.
6. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, pages 643–662, 2012.
7. Ivan Damgård and Sarah Zakarias. Constant-overhead secure computation of boolean circuits using preprocessing. In *TCC*, pages 621–641, 2013.
8. Marcel Keller, Peter Scholl, and Nigel P Smart. An architecture for practical actively secure mpc with dishonest majority. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 549–560. ACM, 2013.
9. Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 681–700. Springer Berlin Heidelberg, 2012.