Gate-scrambling Revisited - or: The TinyTable protocol for 2-Party Secure Computation

Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci

Department of Computer Science, Aarhus University

Abstract. We propose a new protocol, nicknamed TinyTable, for maliciously secure 2-party computation in the preprocessing model. One version of the protocol is useful in practice and allows, for instance, secure AES encryption with latency about 1ms and amortized time about 0.5 μs per AES block on a fast cloud set-up. Another version is interesting from a theoretical point of view: we achieve a maliciously and unconditionally secure 2-party protocol in the preprocessing model for computing a Boolean circuit, where both the communication complexity and preprocessed data size needed is $O(s)$ where s is the circuit size, while the computational complexity is $O(k^{\epsilon}s)$ where k is the statistical security parameter and $\epsilon < 1$ is a constant. For general circuits with no assumption on their structure, this is the best asymptotic performance achieved so far in this model.

1 Introduction

In 2-party secure computation, two parties A and B want to compute an agreed function securely on privately held inputs, and we want to construct protocols ensuring that the only new information a party learns is the intended output.

In this paper we will focus on malicious security: one of the parties is under control of an adversary and may behave arbitrarily. As is well known, this means that we cannot guarantee that the protocol always gives output to the honest party, but we can make sure that the output, if delivered, is correct. It is also well known that we cannot accomplish this task without using a computational assumption, and in fact heavy public-key machinery must be used to some extent.

However, as observed in several works[BDOZ11,DPSZ12,NNOB12,DZ13], one can confine the use of cryptography to a preprocessing phase where the inputs need not be known and can therefore be done at any time prior to the actual computation. The preprocessing produces "raw material" for the on-line phase which is executed once the inputs are known, and this phase can be information theoretically secure, and usually has very small computational and communication complexity, but round complexity proportional to the depth of the computation in question. An incomparable alternative (which is not our focus here) is to use Yao-garbled circuits. This approach gives constant round complexity but has inherently larger communication complexity and seems to be harder to make maliciously secure.

We will focus on the case where the desired computation is specified as a Boolean circuit. The case of arithmetic circuits over large fields was handled in [DPSZ12] which gave a solution where communication and computational complexities as well as the size of the preprocessed data (called data complexity in the following) are proportional to the circuit size. The requirement is that the field has $2^{\Omega(k)}$ elements where k is the security parameter and the allowed error probability is $2^{-\Omega(k)}$.

On the other hand, for Boolean circuits, state of the art is the protocol from [DZ13], nick-named MiniMac which achieves data and communication complexity $O(s)$ where s is the circuit size, and computational complexity $O(k^{\epsilon}s)$, where $\epsilon < 1$ is a constant. For an alternative variant of the protocol, all complexities are $O(\text{polylog}(k)s)$. However, the construction only works for circuits with a sufficiently nice structure, called "well formed" circuits in [DZ13]. Informally, a well-formed circuit allows a modest amount of parallelization throughout the computation – for instance, very tall and skinny circuits are not allowed.

On the practical side, many of the protocols in the preprocessing model are potentially practical and several of them have been implemented. In particular, implementations of the the MiniMac protocol were reported in [DLT14] and [DZ16]. In [DLT14], MiniMac was optimised and used for computing many instances of the same Boolean circuit in parallel, while in [DZ16] the protocol was adapted specifically for computing the AES circuit, which resulted in an implementation with latency about 6 ms and an amortised time of 0,4 ms per AES block.

Our Contribution In this paper, we introduce a new protocol for the preprocessing model, nick-named TinyTable. The idea is to implement each (non-linear) gate by a scrambled version of its truthtable. Players will do look-ups in the tables using bits that are masked by uniformly random bits that are chosen in the preprocessing phase together with the tables. This approach is related to the protocol from [DZ16] where (a different form of) table look-up was used to implement the AES S-boxes. However, the idea of gate-scrambling goes back at least to [CDvdG87] where a (much less efficient) protocol based on quadratic residuosity was proposed. What we do here is to observe that the idea of scrambled truth tables makes sense also in the more recent preprocessing model and, more importantly, if we combine this with the "right" authentication scheme, we get an extremely practical and maliciously secure protocol. This first version of our protocol has data and communication complexity $O(s)$ and computational complexity $O(ks)$. Although this is asymptotically inferior to previous protocols when counting elementary bit operations, it works much better in practice: XOR and NOT gates require no communication, and for each non-linear gate, each player sends and receives 1 bit and XORs 1 word from memory into a register. This means that TinyTable saves a factor of at least 2 in communication in comparison to standard *passively* secure protocols in the preprocessing model, such as GMW with precomputed OT's or the protocol using circuit randomization via Beaver-triples.

We implemented a version of this protocol that was optimised for AES computation, by using tables for each S-box. This is more costly in preprocessing but, compared to a Boolean circuit implementation, it reduces the round complexity of the on-line phase dramatically (to about 10). On a fast cloud set-up, we obtain on-line latency of 1ms and a time of about 0.5 μs per AES block. To the best of our knowledge, this is the fastest on-line time obtained for secure AES.

We describe how one can do the preprocessing we require based on the preprocessing phase of the TinyOT protocol from [NNOB12]. This protocol is basically maliciously secure OT extension with some extra processing on top. In the case of Boolean circuits, the work needed per AND gate roughly equals the work needed per AND gate in TinyOT. For the case of AES S-box tables, we show how to use a method from [DK10] to preprocess such tables. This requires 7 *binary* multiplications per table entry and local computation.

As for the speeds of preprocessing one can obtain, the best approaches and implementations of this type of protocol are from [FKOS15]. They do not have an implementation of the exact case we need here (2-party TinyOT) but they estimate that one can obtain certainly more than 10.000 AND gates per second and most likely up to 100.000 per second [Kel].

Our final contribution is a version of our protocol that has better asymptotic performance. We get data and communication complexity $O(s)$, and computational complexity $O(k^\epsilon s)$, where $\epsilon < 1$ is a constant. Alternatively we can also have all complexities be $O(\text{polylog}(k)s)$. While this is the same result that was obtained for the original MiniMac protocol, note that we get this for *any* circuit, not just for well-formed ones. Roughly speaking, the idea is to use the MiniMac protocol to authenticate the bits that players send during the on-line phase. This task is very simple: it parallelizes easily and can be done in constant depth. Therefore we get a better result than if we had used MiniMac directly on the circuit in question.

2 Construction

We show a 2PC protocol for securely computing a Boolean circuit C for two players A and B. The circuit may contain arbitrary gates taking two inputs and giving one output. We let $G_1, \ldots, G_N$ be the gates of the circuit, and let $w_1, \ldots, w_W$ be the wires. We note that arbitrary fan-out is allowed, and we do not assume special fan-out gates for this, we simply allow that an arbitrary number of copies of a wire leave a gate, and all these copies are assigned the same wire index. We assume for simplicity that both parties are to learn the output.

We will fix some arbitrary order in which the circuit can be evaluated gate by gate, such that the output gates come last, and such that when we are about to evaluate gate i, its inputs have already been computed. We assume throughout that the indexing of the gates satisfy this constraint. We call the wires coming out of the output gates the *output wires*.

We first specify a functionality for preparing preprocessing material that will allow computation of the circuit with semi-honest security, see Figure 1[1].

Preprocessing Functionality F^{pre}_{sem}.

1. On input C from both players, do the following: For each wire w_u, choose a random masking bit r_u. This bit will be used to mask the bit b_u that will actually be on w_u when we do the computation, i.e., $e_u = b_u \oplus r_u$ will become known to the players. If w_u is an input wire, give r_u to the player who owns w_u.
2. For each gate G_i, with input wires w_u, w_v and output wire w_o, we will construct two tables A_i, B_i each with 4 entries, indexed by bits (c, d). This is done as follows: for each of the 4 possible values of bits (c, d), do:
   (a) If both player are honest, choose a random bit $s_{c,d}$. Otherwise take $s_{c,d}$ as input from the adversary. Let $G_i(\cdot, \cdot)$ denotes the function computed by gate G_i.
   (b) If both parties are honest, or if A is corrupt, set $A_i[c, d] = s_{c,d}$ and $B_i[c, d] = s_{c,d} \oplus (r_o \oplus G_i(c \oplus r_u, d \oplus r_v))$.
   (c) If B is corrupt, set $B_i[c, d] = s_{c,d}$ and $A_i[c, d] = s_{c,d} \oplus (r_o \oplus G_i(c \oplus r_u, d \oplus r_v))$.
3. For each gate G_i, hand A_i to player A and B_i to player B. For each output wire w_u, send r_u to both players.

Fig. 1. Functionality for preprocessing, semi-honest security.

The idea behind the construction of the tables is that when the time comes to compute gate G_i, both players will know "encrypted bits" $e_u = b_u \oplus r_u$ and $e_v = b_v \oplus r_v$, for the input wires, where b_u, b_v are the actual "cleartext" bits going into G_i, and r_u, r_v are random masking bits that are chosen in the preprocessing. In addition, the preprocessing sets up two tables A_i, B_i for each gate, one held by A and one held by B. These tables are used to get hold of a similar encrypted bit for the output wire: $e_o = b_o \oplus r_o$, where $b_o = G_i(b_u, b_v)$. This works because the tables are set up such that $A_i[e_u, e_v], B_i[e_u, e_v]$ is an additive sharing of $b_o \oplus r_o$, i.e.,

$$A_i[b_u \oplus r_u, b_v \oplus r_v] \oplus B_i[b_u \oplus r_u, b_v \oplus r_v] = b_o \oplus r_o \ .$$

[1] For this functionality as well as for the other preprocessing functionalities we define, whenever players are to receive shares of a secret value, the functionality lets the adversary choose shares for the corrupt player. This is a standard trick to make sure that the functionality is easier to implement: the simulator can simply run a fake instance of the protocol with the adversary and give to the functionality the shares that the corrupt player gets out of this. If we had let the functionality make all the choices, the simulator would have to force the protocol into producing the shares that the functionality wants. This weaker functionality is still useful: as long as the shared secret is safe, we don't care which shares the corrupt player gets.

These considerations lead naturally to the protocol for computing C securely shown in Figure 2. For the security of this construction, note that since the masking bits for the wires are uniformly random, each bit e_u is random as well (except when w_u is an output wire), and so the positions in which we look up in the tables are also random. With these observations, it is easy to see that we have:

Protocol π_{sem}.

1. A and B send C as input to F and get a set of tables $\{A_i, B_i | \ i = 1 \dots N\}$, as well as a bit b_u for each input wire w_u.
2. For each input wire w_u, if A holds input x_u for this wire, send $e_u = x_u \oplus b_u$ to B. If B holds input x_u, send $e_u = x_u \oplus b_u$ to A.
3. For $i = 1$ to N, do: Let G_i have input wires w_u, w_v and output wire w_o (so that e_u, e_v have been computed). A sends $A_i[e_u, e_v]$ to B, and B sends $B_i[e_u, e_v]$ to A. Set $e_o = A_i[e_u, e_v] \oplus B_i[e_u, e_v]$.
4. The parties output the bits $\{e_o \oplus r_o | \ w_o \text{ is an output wire}\}$.

Fig. 2. Protocol for semi-honest security.

Proposition 1. *F_{sem}^{pre} composed with π_{sem} implements with semi-honest security the ideal functionality F_{SFE} for secure function evaluation.*

Functionality F_{mal}^{pre}.

1. On input C from both players, execute the same algorithm as F_{sem}^{pre} would on input C.
2. In addition, for $i = 1, \dots, N$ and for all 4 values of (c, d), do:
   If both players are honest, select random k-bit strings $a_{i,c,d}^0, a_{i,c,d}^1, b_{i,c,d}^0, b_{i,c,d}^1$.
   If player A is corrupt, take $b_{i,c,d}^0, b_{i,c,d}^1$ as input from the adversary and choose $a_{i,c,d}^0, a_{i,c,d}^1$ at random.
   If player B is corrupt, take $a_{i,c,d}^0, a_{i,c,d}^1$ as input from from the adversary and choose $b_{i,c,d}^0, b_{i,c,d}^1$ at random.
3. Hand $a_{i,c,d}^0, a_{i,c,d}^1$ to B and $b_{i,c,d}^0, b_{i,c,d}^1$ to A.
   Hand $a_{i,c,d}^{A_i[c,d]}$ to A, and $b_{i,c,d}^{B_i[c,d]}$ to B.

Fig. 3. Preprocessing functionality, malicious security.

Here, F_{SFE} is a standard functionality that accepts C as input from both parties, then gets x from A, y from B and finally outputs $C(x, y)$ to both parties, in case of semi-honest corruption. In case of malicious corruption, it will send the output to the corrupted party and let the adversary decide whether to abort or not. Only in the latter case will it send the output to the honest party.

It is also very natural that we can get malicious security if we assume instead a functionality that commits A and B to the tables. We can get this effect by letting the functionality also store the tables and outputting bits from them to the other party on request. In other words, what we need is a functionality that allows us to commit players to correlated bit strings such that they can later open a subset of the bits. Note that the subset cannot be known in advance, which makes it more difficult to implement efficiently. Indeed, if we go for the simplest information theoretically secure solution in the OT-hybrid model, the storage requirement will be $O(\ell k)$ bits where ℓ is the string length and k is the security parameter. This is achieved, essentially by committing to each bit individually. In contrast, if we just need to open the entire string or nothing, $O(\ell + k)$ bits is sufficient using a number of standard techniques. However, this difference is not inherent: as we discuss in more detail in Section 6, one can in fact achieve both storage and communication complexity $O(\ell + k)$ bits also for opening a subset. On the other hand, the simple bit-by-bit solution works better in practice.

In Figure 3, we show a functionality that does preprocessing as in the semi-honest case, but in addition commits players bit by bit to the content of the tables. The idea is that, for entry $A_i[c,d]$ in a table, A is also given a random string $a_{i,c,d}^{A_i[c,d]}$ which serves as an authentication code that A can use to show that he sends the correct value of $A_i[c,d]$, while B is given the pair $a_{i,c,d}^0, a_{i,c,d}^1$ serving as a key that B can use to verify that he gets the correct value. Of course, using the authentication codes directly in this way, we would have to send k bits to open each bit. However, in our application, we can bring down the communication needed to (essentially) $\ell + k$ bits, because we can delay verification of most of the bits opened. The idea is that, instead of sending the authentication codes, players will accumulate the XOR of all of them and check for equality later at the end. The protocol shown in Figure 4 uses this idea to implement maliciously secure computation.

Theorem 1. *F_{mal}^{pre} composed with π_{mal} implements F_{SFE} with malicious and statistical security.*

This theorem follows easily from the observation that if a corrupt party, say A, sends an incorrect bit, he has no information about the authentication code that goes with this value. If this happens in Step 5, B will catch the error except with probability 2^{-k}. If it happens in Step 3 it will imply that the adversary has no information on the final value of t_B and then the check in Step 5 will fail except with probability 2^{-k}. On the other hand, if all bits sent are correct, security follows in the same way as for the semi-honest case.

2.1 Free XOR

It is easy to modify our construction to allow non-interactive processing of XOR gates. For simplicity, we only show how this works for the case of semi-honest

Protocol π_{mal}.

1. A and B send C as input to F_{mal}^{pre} and get a set of tables $\{A_i, B_i | \ i = 1 \dots N\}$, as well as masking bits for input and output wires. In addition, for all values of (i, c, d), A receives strings $b_{i,c,d}^0, b_{i,c,d}^1$ and $a_{i,c,d}^{A_i[c,d]}$. These are stored for later. (Symmetrically, B gets and stores a corresponding set of strings.)
2. Initialize variables: A sets $m_A = t_A = 0^k$, and B sets $m_B = t_B = 0^k$. Here m_A is an accumulative MAC on the values that A sends and t_B is the value that m_A should have if A was honest, and symmetrically for m_B and t_A.
3. For each input wire w_u, if A holds input x_u for this wire, send $e_u = x_u \oplus b_u$ to B. If B holds input x_u, send $e_u = x_u \oplus b_u$ to A.
4. Define M such that the last $N - M$ gates are output gates. For $i = 1$ to M do:
   (a) Let G_i have input wires w_u, w_v and output wire w_o (so that e_u, e_v have been computed).
   (b) A sends $e_A = A_i[e_u, e_v]$ to B, and sets $m_A = m_A \oplus a_{i,e_u,e_v}^{e_A}$. B sets $t_B = t_B \oplus a_{i,e_u,e_v}^{e_A}$. Note that B can do this, as he has both strings $a_{i,e_u,e_v}^0, a_{i,e_u,e_v}^1$.
   (c) B sends $e_B = B_i[e_u, e_v]$ to A, and sets $m_B = m_B \oplus b_{i,e_u,e_v}^{e_B}$. A sets $t_A = t_A \oplus b_{i,e_u,e_v}^{e_B}$.
   (d) Both parties set $e_o = e_A \oplus e_B$.

   Check if all bits opened until now were correct: A sends m_A to B and B sends m_B to A. A verify that $t_A = m_B$ and B verify that $t_B = m_A$. The parties abort if not the case.
6. Reveal the outputs. For $i = M + 1$ to N do:
   (a) Let G_i have input wires w_u, w_v and output wire w_o (so that e_u, e_v have been computed).
   (b) A sends $e_A = A_i[e_u, e_v]$ and $a_{i,e_u,e_v}^{e_A}$ to B. B checks that $a_{i,e_u,e_v}^{e_A}$ is correct. Note that B can do this, as he has both strings $a_{i,e_u,e_v}^0, a_{i,e_u,e_v}^1$.
   (c) B sends $e_B = B_i[e_u, e_v]$ and $b_{i,e_u,e_v}^{e_B}$ to A. A checks that $b_{i,e_u,e_v}^{e_B}$ is correct.
   (d) Both parties set $e_o = e_A \oplus e_B$.

   The correctness of the sent information is checked via an accumulative MAC as in Step 3
7. The parties output the bits $\{e_o \oplus r_o | \ w_o \text{ is an output wire}\}$.

Fig. 4. Protocol for malicious security.

security, malicious security is obtained in exactly the same way as in the previous section.

The idea is to select the wire masks in a different way, exploiting the homomorphic properties of XOR gates. Basically, if we encounter a NOT gate, we make sure the output wire mask is equal the input wire mask. For XOR gates we set the output wire mask to the XOR of the input wires masks. We ensure this invariant by traversing the circuit, since the wire masks can no longer be sampled independently at random. One could set the output wire mask to zero for all output gates and traverse the gates backwards $G_N, \dots, G_1$ ensuring the invariant on the wire masks. Another approach, which is the one we use in the the following, is to traverse the gates forwardly $G_1, \dots, G_N$ ensuring the invariants and then give output wire masks to the parties, who will then remove the mask before returning the actual output. In the online phase, we can now process XOR and NOT gates locally by computing the respective function directly on masked values. The resulting protocol is shown in Figure 6.

Preprocessing Functionality $F^{pre}_{freeXOR,sem}$.

1. On input C from both players, for each input wire w_u, choose a random bit r_u, and if w_u is an input wire, give r_u to the player who owns that wire.
   For $i = 1$ to N, do: Let w_u, w_v be the input wires of G_i and w_o the output wire. Note that we have already chosen masking bits r_u, r_v for w_u, w_v. If G_i is an XOR gate, set $r_o = r_u \oplus r_v$. If G_i is a NOT gate (so that w_u is the only input wire), set $r_o = r_v$ If G_i is any other gate, choose r_o at random.
2. For each gate G_i which is not an XOR or NOT gate, with input wires w_u, w_v and output wire w_o, we will construct two tables A_i, B_i each with 4 entries, indexed by bits (c, d). This is done as follows: for each of the 4 possible values of bits (c, d), do:
   (a) If both player are honest, choose a random bit $s_{c,d}$. Otherwise take $s_{c,d}$ as input from the adversary. Let $G_i(\cdot, \cdot)$ denote the function computed by gate G_i.
   (b) If both parties are honest, or if A is corrupt, set
   $A_i[c, d] = s_{c,d}$ and $B_i[c, d] = s_{c,d} \oplus (r_o \oplus G_i(c \oplus r_u, d \oplus r_v))$.
   (c) If B is corrupt, set
   $B_i[c, d] = s_{c,d}$ and $A_i[c, d] = s_{c,d} \oplus (r_o \oplus G_i(c \oplus r_u, d \oplus r_v))$.
3. For each gate G_i, where we built tables, hand A_i to player A and B_i to player B. Send wire masks for the output wires to both players.

Fig. 5. Functionality for preprocessing, semi-honest security with free XOR.

Protocol π_{sem}.

1. A and B send C as input to F and get a set of tables $\{A_i, B_i | \ i = 1 \dots N\}$, as well as a bit b_u for each input wire w_u.
2. For each input wire w_u, if A holds input x_u for this wire, send $e_u = x_u \oplus b_u$ to B. If B holds input x_u, send $e_u = x_u \oplus b_u$ to A.
3. For $i = 1$ to N, do: Let G_i have input wires w_u, w_v and output wire w_o (so that e_u, e_v have been computed). If G_i is an XOR gate, set $e_o = e_u \oplus e_v$. If G_i is a NOT gate (so that w_u is the only input wire), set $e_o = 1 - e_u$. Otherwise (we have tables for this gate): A sends $A_i[e_u, e_v]$ to B, and B sends $B_i[e_u, e_v]$ to A. Both set $e_o = A_i[e_u, e_v] \oplus B_i[e_u, e_v]$.
4. The parties output the bits $\{e_o \oplus r_o | \ w_o \text{ is an output wire}\}$.

Fig. 6. Protocol for semi-honest security, free XOR.

2.2 Generalisation to Bigger Tables

If the circuit contains a part that evaluates a non-linear function f on a small input, it is natural to implement computation of this function as a table. If the input is small, such a table is not prohibitively large. Suppose, for instance, that the input and output is 8 bits, as is the case for the AES S-Box. Then we will store tables A_f, B_f each indexed by 8-bit value M such that $A_f[M] \oplus B_f[M] = f(x \oplus M) \oplus O$, where O is an 8-bit output mask chosen in the preprocessing. We make sure in the preprocessing that the i'th bit of B denoted $B[i]$ equals the wire mask for the i'th wire going into the evaluation of f, whereas $O[i]$ equals the wire mask for the i'th wire receiving output from f. We can then use the table for f exactly as we use the tables for the AND gates.

To get malicious security we must note that the simple authentication scheme we used in the binary case will be less practical here: as we have 256 possible output values, each party would need to store 256 strings per table entry. It turns out this can be optimized considerably, as described in the following section.

3 The Linear MAC Scheme

There is a committer C, a verifier V and a preprocessor P. There is a security parameter k. Some of the computations are done over the finite field $\mathbb{F} = \mathrm{GF}(2^k)$. Let $p(\mathtt{X})$ be the irreducible polynomial of degree k used to do the computation in $\mathbb{F} = \mathrm{GF}(2^k)$, i.e, elements $x, y \in \mathbb{F}$ are polynomials of degree at most $k-1$ and multiplication is computed as $z = xy \bmod p$. We will also be doing computations in the finite field $\mathbb{G} = \mathrm{GF}(2^{2k-1})$. Let $q(\mathtt{X})$ be the irreducible polynomial of degree $2k-1$ used to do the computation in $\mathbb{G}$. Notice that elements $x, y \in \mathbb{F}$ are polynomials of degree at most $k-1$, so xy is a polynomial of degree at most $2k-2$. We can therefore think of xy as an element of $\mathbb{G}$. Note in particular that $xy \bmod q = xy$ when $x, y \in \mathbb{F}$.

3.1 Basic Version

The MAC scheme has message space $\mathbb{F}$. We denote a generic message by $x \in \mathbb{F}$. The MAC scheme has key space $\mathbb{F} \times \mathbb{G}$. We denote a generic key by $K = (\alpha, \beta) \in \mathbb{F} \times \mathbb{G}$. The tag space is $\mathbb{G}$. We denote a generic tag by $y \in \mathbb{G}$. The tag is computed as $y = \mathrm{mac}_K(x) = \alpha x + \beta$. Note that $\alpha \in \mathbb{F}$ and $x \in \mathbb{F}$, so $\alpha x \in \mathbb{G}$ as described above and hence can be added to β in $\mathbb{G}$. We use this particular scheme because it can be computed very efficiently using the `PCLMULQDQ` instruction on modern Intel processors. With one `PCLMULQDQ` instruction we can compute αx from which we can compute $\alpha x + \beta$ using one additional XOR.

The intended use of the MAC scheme is as follows. The preprocessor samples a message x and a uniformly random key K and computes $y = \mathrm{mac}_K(x)$. It gives K to V and gives (x, y) to C. To reveal the message C sends (x, y) to V who accepts if and only if $y = \mathrm{mac}_K(x)$. Since K is sampled independently of x, the scheme is clearly hiding in the sense that V gets no information on x before receiving the opening information. We now show that the scheme is binding.

Let $\mathcal{A}$ be an unbounded adversary. Run $\mathcal{A}$ to get a message $x \in \mathbb{F}$. Sample a uniformly random key $(\alpha, \beta) \in \mathbb{F} \times \mathbb{G}$ and compute $y = \alpha x + \beta$. Given y to $\mathcal{A}$. Run $\mathcal{A}$ to get an output $(y', x') \in \mathbb{G} \times \mathbb{F}$. We say that $\mathcal{A}$ wins if $x' \neq x$ and $y' = \mathrm{mac}_K(x')$. We can show that no $\mathcal{A}$ wins with probability better than 2^{-k}. To see this, notice that if $\mathcal{A}$ wins then he knows x, y, x', y' such that $y = \alpha x + \beta$ and $y' = \alpha x' + \beta$. This implies that $y' - y = \alpha(x' - x)$, from which it follows that $\alpha = (y' - y)(x' - x)^{-1}$. This means that if $\mathcal{A}$ can win with probability r then $\mathcal{A}$ can guess α with probability at least r. It is then sufficient to prove that no adversary can guess α with probability better than 2^{-k}. This follows from the fact that α is uniformly random given $\alpha x + \beta$, because αx is some element of $\mathbb{G}$ and β is a uniformly random element of $\mathbb{G}$ independent of αx.

3.2 The Homomorphic Vector Version

We now describe a vector version of the scheme which allows to commit to multiple message using a single key.

The MAC scheme has message space $\mathbb{F}^\ell$. We denote a generic message by $\boldsymbol{x} \in \mathbb{F}^\ell$. The MAC scheme has key space $\mathbb{F} \times \mathbb{G}^\ell$. We denote a generic key by $K = (\alpha, \boldsymbol{\beta}) \in \mathbb{F} \times \mathbb{G}^\ell$. The tag space is $\mathbb{G}^\ell$. We denote a generic tag by $\boldsymbol{y} \in \mathbb{G}^\ell$. The tag is computed as $\boldsymbol{y} = \mathrm{mac}_K(\boldsymbol{x}) = \alpha \boldsymbol{x} + \boldsymbol{\beta}$, i.e., $y_i = \alpha x_i + \beta_i$. Note that $\alpha \in \mathbb{F}$ and $x_i \in \mathbb{F}$, so $\alpha x_i \in \mathbb{G}$.

The intended use of the MAC scheme is as follows. The preprocessor samples a message $\boldsymbol{x}$ and a uniformly random key K and computes $\boldsymbol{y} = \mathrm{mac}_K(\boldsymbol{x})$. It gives K to V and gives $(\boldsymbol{x}, \boldsymbol{y})$ to C. To reveal the message x_i the comitter C sends (x_i, y_i) to V, who accepts if and only if $y_i = \alpha x_i + \beta_i$.

The preprocessed information also allows to open to any sum of a subset of the x_i's. Let $\boldsymbol{\lambda} \in \mathbb{F}^\ell$ with $\lambda_i \in \{0, 1\}$. Let $x_\lambda = \sum_i \lambda_i x_i \pmod{\mathbb{F}}$, let $y_\lambda = \sum_i \lambda_i y_i \pmod{\mathbb{G}}$, and let let $\beta_\lambda = \sum_i \lambda_i \beta_i \pmod{\mathbb{G}}$. To reveal x_λ the committer C sends (x_λ, y_λ) and the verifier V checks that $y_\lambda = \alpha x_\lambda + \beta_\lambda$. If both players are honest, this is clearly the case. The only non-trivial thing to notice is that since $\sum_i \lambda_i x_i \pmod{\mathbb{F}}$ does not involve any reduction modulo p we have that $\sum_i \lambda_i x_i \pmod{\mathbb{F}} = \sum_i \lambda_i x_i = \sum_i \lambda_i x_i \pmod{\mathbb{G}}$.

The scheme is hiding in the sense that after a number of openings to elements x_λ the verifier learns nothing more than what can be computed from the received values x_λ. To see this notice that K is independent of $\boldsymbol{x}$ and hence could be simulated by V. Also the openings can be simulated. Namely, whenever V received an opening (x_λ, y_λ) from an honest C, we know that $y_\lambda = \alpha x_\lambda + \beta_\lambda$, so V could have computed y_λ itself from x_λ and K. Hence no information extra to x_λ is transmitted by transmitting (x_λ, y_λ).

We then prove that the scheme is binding. Let $\mathcal{A}$ be an unbounded adversary. Run $\mathcal{A}$ to get a message $\boldsymbol{x} \in \mathbb{F}^\ell$. Sample a uniformly random key $(\alpha, \boldsymbol{\beta}) \in \mathbb{F} \times \mathbb{G}^\ell$ and compute $y_i = \alpha x_i + \beta_i$ for $i = 1, \ldots, \ell$. Give $\boldsymbol{y}$ to $\mathcal{A}$. Run $\mathcal{A}$ to get an output $(y', x', \boldsymbol{\lambda}) \in \mathbb{G} \times \mathbb{F} \times \{0, 1\}^\ell$. We say that $\mathcal{A}$ wins if $x' \neq x_\lambda$ and $y' = \alpha x' + \beta_\lambda$. We can show that no $\mathcal{A}$ wins with probability better than 2^{-k}. To see this, notice that if $\mathcal{A}$ wins then he knows x' and y' such that $y' = \alpha x' + \beta_\lambda$. He also knows x_λ and y_λ as these can be computed from $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{\lambda}$, which he knows already.

And, it holds that $y_\lambda = \alpha x_\lambda + \beta_\lambda$. Therefore $y' - y_\lambda = \alpha(x' - x_\lambda)$, and it follows as above that $\mathcal{A}$ can compute α. Since no information is leaked on α by the value $\alpha\boldsymbol{x} + \boldsymbol{\beta}$ given to $\mathcal{A}$ it follows that it is uniform in $\mathbb{F}$ in the view of $\mathcal{A}$. Therefore $\mathcal{A}$ can compute α with probability at most 2^{-k}.

We note for completeness that the scheme could be extended to arbitrary linear combinations. In that case one would however have to send $x_\lambda = \sum_i \lambda_i x_i \pmod{\mathbb{F}}$ and $y_\lambda = \sum_i \lambda_i x_i \pmod{\mathbb{F}}$ which would involve doing reductions modulo p. The advantage of the above scheme where $\lambda_i \in \{0,1\}$ is that no polynomial reductions are needed, allowing full use of the efficiency of the `PCLMULQDQ` instruction.

3.3 Batched Opening

We now describe a technique to opening a large number of commitments by sending only k bits. For notational simplicity we assume that C wants to reveal the values $(x_1, \ldots, x_n)$, but the scheme trivially extends to opening arbitrary subsets and linear combinations. To reveal $(x_1, \ldots, x_n)$ as described above, C would send $Y^{\mathsf{C}} = (y_1, \ldots, y_n)$ and V would compute $y_i^{\mathsf{V}} = \alpha x_i + \beta$ for $i = 1, \ldots, n$ and $Y^{\mathsf{V}} = (y_1^{\mathsf{V}}, \ldots, y_n^{\mathsf{V}})$ and check that $Y^{\mathsf{V}} = Y^{\mathsf{C}}$. Consider now the following optimization where C and V is given a hash function H from P. They could then compare Y^{C} and Y^{V} by sending $h^{\mathsf{C}} = H(Y^{\mathsf{C}})$ and checking that $h^{\mathsf{C}} = H(Y^{\mathsf{V}})$.

We saw above that if C sends $(x_1', \ldots, x_n')$ and $(y_1', \ldots, y_n')$ where $x_i' \neq x_i$ for some i, then the probability that $y_i' = y_i^{\mathsf{V}}$ is at most 2^{-k}.

The optimization would therefore be secure if H is secure in the following game. Sample a random H and give it to a poly-time adversary $\mathcal{A}$ which will then submit a value h and distribution D over $\mathbb{G}^n$ and an index $i \in \{1, \ldots, n\}$ with the property that if we sample $(d_1, \ldots, d_n)$ from D, then the min-entropy of d_i in the view of $\mathcal{A}$ is k. The adversary wins if $h = H(d_1, \ldots, d_n)$. We say that H is secure if no poly-time $\mathcal{A}$ wins with better than negligible probability.

Any collision resistant hash function has this property as we can do a reduction by simply sampling D multiple times to get different $(d_1, \ldots, d_n)$ and $(d_1', \ldots, d_n')$ such that $h = H(d_1, \ldots, d_n)$ and $h = H(d_1', \ldots, d_n')$ for the h given by $\mathcal{A}$. It is easy to see that if P is a uniformly random permutation then $H(d_1, \ldots, d_n) = \bigoplus_{i=1}^n P(i \| id_i)$ is collision resistant, which gives a construction in the ideal permutation model.

4 Preprocessing

In this section we show how to securely implement the preprocessing. The idea is to generate the tables by a computation on linear secret shared values, which in case of malicious security also include MACs. We will consider additive secret sharing of x where A hold $x_A \in \{0,1\}$ and B hold $x_B \in \{0,1\}$ such that $x = x_A + x_B$.

In the case of malicious security the MACs takes place over a field $\mathbb{F}$ of characteristic 2 and of size at least 2^k where k is the security parameter. Here

A hold a key $\alpha_A \in \mathbb{F}$ and B a key $\alpha_B \in \mathbb{F}$. We denote a secret shared value with MACs $[\![x]\!]$ where A hold (x_A, y_A, β_A) and B hold (x_B, y_B, β_B) such that $y_A = \alpha_B x_A + \beta_B$ and $y_B = \alpha_A x_B + \beta_A$. If a value is to be opened the MAC is checked, e.g. if x is opened to A she receives x_B and y_B and checks that indeed $y_B = \alpha_A x_B + \beta_A$ or abort otherwise. This is also the format used in the TinyOT protocol [NNOB12], so we can use the preprocessing protocol from there to produce single values $[\![a]\!]$ for random a and triples of form $[\![x]\!], [\![y]\!], [\![z]\!]$ where x, y are random bits and $z = xy$.

Note that, by standard protocol, we can use one triple to produce from any $[\![a]\!], [\![b]\!]$ the product $[\![ab]\!]$, this just requires opening $a + x, b + y$ and local computation. Also, we can compute the sum $[\![a + b]\!]$ by only local computation.

In Figure 7, we describe a protocol that implements the preprocessing functionality F^{pre}_{mal} assuming a secure source of triples and single random values as described here, and also assuming that the circuit contains only AND, XOR and NOT gates. We use a permutation P that we model (for simplicity) as a random permutation. It is sufficient that $H(x_1, \ldots, x_n) = \bigoplus_i P(i \| x_i)$ is collision resistant as discussed in the previous section.

For simplicity the protocol is phrased as a loop that runs through all gates of the circuit, but we stress that it can be executed in constant round: we can execute step 3 in the protocol by first doing all the XOR and NOT gates, using only local operations. At this point wire masks have been chosen for all input and output wires of all AND gates, and they can now all be done in parallel.

Preprocessing for AES To preprocess an AES Sbox table, we can again make use of the TinyOT preprocessing. This can be combined with a method from [DK10]. Here, it is shown how to compute the Sbox function securely using 7 binary multiplications and local operations. We can then make the table by simply computing each entry (in parallel). It is also possible to compute the Sbox using arithmetic in $\mathbb{F}_{256}$, but if we have to build such multiplications from binary ones, as we would if using TinyOT, this most likely does not pay off.

5 Implementation

We implement two clients, Alice and Bob, securely evaluating the AES-128 encryption function E. Alice inputs the message and Bob inputs the key. Note that Bob locally computes the key schedule and inputs the expanded key to the protocol. The encryption function E contains several operations where all operations except SubBytes are linear. The parties compute the linear operations locally on masked values. We are left with replacing every S-box (lookup) with a tiny table (lookup and opening). For a passively secure evaluation we need $40kB$ of preprocessing data (160 tiny tables with 2^8 entries of one byte each). For malicious security we add MACs to obtain statistical security 2^{-k}. We test the simple lookup MAC scheme where we use $k + 256k$ bits extra per entry. For the linear MAC scheme we use a k bit key (reused) and $2k$ per entry.

Protocol π_{mal}^{pre}.

1. Invoke the TinyOT preprocessing such that for each AND gate we have a triple secret of shared values $[\![x]\!], [\![y]\!]$ and $[\![z]\!]$ such that $xy = z$.
2. Also using the TinyOT preprocessing, for each wire w_i that is an input wire, or an output wire from an AND gate, the players assign a random $[\![r_i]\!]$ to w_i, r_i will serve as the masking bit for w_i.
3. For each gate G_i with where masking bits $[\![r_u]\!], [\![r_v]\!]$ have been assigned to the input wires w_u, w_v the players do as follows, depending on the type of gate:
   NOT: Set the output wire mask $[\![r_o]\!] = [\![r_u]\!]$
   XOR: Set the output wire mask $[\![r_o]\!] = [\![r_u]\!] + [\![r_v]\!]$
   AND:
   - Compute $[\![r_u r_v]\!]$ from $[\![r_u]\!]$ and $[\![r_v]\!]$ using the triple assigned to this AND gate, using the protocol from [**?**].
   - For all $c, d \in \{0, 1\}$ define $t_{c,d} = (r_u + c)(r_v + d) + r_o$ and compute a secret sharing of it as follows: $[\![t_{c,d}]\!] = [\![r_u r_v]\!] + c[\![r_v]\!] + d[\![r_u]\!] + [\![cd]\!] + [\![r_o]\!]$. This requires only local computation.

   Note that $t_{c,d}$ is the bit that needs to be additively secret shared for entry (c, d) in the table for gate G_i.
   A sets $A_i[c, d]$ to be his share of $t_{c,d}$ (which he knows from $[\![t_{c,d}]\!]$), B defines $B_i[c, d]$ similarly.
   Note further that from his part of $[\![t_{c,d}]\!]$, A can compute valid MACs $mac_{i,c,d}^0, mac_{i,c,d}^1$ for both possible values of B's additive share in $t_{c,d}$.
   A sets $a_{i,c,d}^0 = P(i \| mac_{i,c,d}^0)$, $a_{i,c,d}^1 = P(i \| mac_{i,c,d}^1)$, while B defines $b_{i,c,d}^0, b_{i,c,d}^1$ similarly[a].
4. For each output gate open the wire mask r_o to both players. For each input wire w_i, open r_i to the player who owns that wire.
5. The parties return the opened input masks, output masks as well as tables A_i, B_i and verifications strings $a_{i,c,d}^0, a_{i,c,d}^1, b_{i,c,d}^0, b_{i,c,d}^1$ for each AND gate.

[a] We need to apply P because we want the verification strings to be independent, and this is not the case if we use the macs directly.

Fig. 7. Protocol for preprocessing with malicious security

In the test setups the parties receive preprocessed data generated by a trusted party. They proceed to compute E on a set of test vectors and verify correctness. We test the implementation on two setups: a basic LAN setup and on a cloud. The LAN setup consist of two PCs connected via 1GbE, where each machine has 8 CPU cores at 3.5GHz. For our cloud setup we use Amazon EC2 with two c4.8xlarge instances connected via 10GbE with 36 vCPUs (hyperthreads). We utilize the Intel SSE and AES instruction set for local operations and communicate over the TCP protocol.

The results are measured as the average over 30 seconds. On each setup we test different MAC schemes: the linear scheme, the simple lookup scheme and a passive scheme. We test the linear scheme with security $k = 64$, the lookup with $k = 64$ and $k = 32$ and the passive where $k = 0$. For the sequential tests the network delay is the bottleneck for timings as we communicate over 11 rounds for passive security and 13 rounds for malicious security. For the parallel tests the bandwidth is the bottleneck on the LAN setup, while computation begins to be the bottleneck on the cloud setup as we raise the security parameter. The results are summarized in Table 1.

Table 1. Execution times

	LAN		**Cloud**	
	Sequential	Parallel	Sequential	Parallel
Linear-64	1.03ms	3.15μs	1.09ms	0.47μs
Lookup-64	1.03ms	3.01μs	1.05ms	0.45μs
Lookup-32	1.02ms	2.95μs	1.05ms	0.32μs
Passive	0.88ms	2.89μs	0.97ms	0.29μs

6 An asymptotically better solution

Recall that the main problem we have with obtaining malicious security is that we must make sure that players reveal correct bits from the tables they are given, but on the other hand only the relevant bits should be revealed.

In this section we show an asymptotically better technique for committing players to their tables such that we can open only the relevant bits.

The idea is as follows: if player A is to commit to string $\boldsymbol{s}$, that is known at preprocessing time, then the preprocessing protocol will establish a (verifiable) secret sharing of $\boldsymbol{s}$ among the players. Concretely, we will use the representation introduced for the so-called MiniMac protocol in [DZ13]: we choose an appropriate linear error correcting (binary) code C. This code should be able to encode strings of length k bits, and have length and minimum distance linear in k.[2] The preprocessing additively secret shares the encoding $C(\boldsymbol{s})$ among the players. $\boldsymbol{s}$ has length ℓ which is comparable to the circuit size which we assume is much larger than k. So $C(\boldsymbol{s})$ is computed by splitting $\boldsymbol{s}$ in k-bit blocks, encoding each block in C and concatenating the encodings.

The preprocessing also chooses a random string $\boldsymbol{a}$, unknown to both players, and both $\boldsymbol{a}$ and $\boldsymbol{a} * C(\boldsymbol{s})$ are additively secret shared. Here $*$ denotes the bitwise (Schur) product. We will use the notation $[\boldsymbol{s}]$ as shorthand for all the additive shares held by the players relevant to $\boldsymbol{s}$. The idea is that $\boldsymbol{a} * C(\boldsymbol{s})$ serves as a message authentication code for authenticating $\boldsymbol{s}$, where $\boldsymbol{a}$ is the key. We note

[2] Furthermore its Schur transform should also have minimum distance linear in k. The Schur transform is the code obtained as the linear span of of all vectors in the set $\{\boldsymbol{c} * \boldsymbol{d} |\ \boldsymbol{c}, \boldsymbol{d} \in C\}$. See [DZ13] for further details on existence of such codes.

Functionality $F^{pre}_{MiniMac}$.

1. On input C from both players, execute the same algorithm as F^{pre}_{sem} would on input C.
2. Let $\boldsymbol{s}_A$ be the string obtained by concatenating all tables $\{A_i\}$ created for A. Similarly $\boldsymbol{s}_B$ contains all B's tables. Create MiniMac representations $[\boldsymbol{s}_A], [\boldsymbol{s}_B]$ and give the resulting additive shares to A and B.
3. Create correlated random strings $data_A, data_B$ for the MiniMac protocol to do $\lceil \ell/k \rceil$ multiplications of blocks, and hand $data_A$ to A and $data_B$ to B.

Fig. 8. Preprocessing functionality using the MiniMac protocol

that, as in [DZ13], $\boldsymbol{a}$ is in fact a global key that is also used for other data represented in this same format.

Later, when it is known (to both players) which substring A needs to open, we let I denote the characteristic vector of the substring, i.e., it is an ℓ bit string where the i'th bit is 1 if A is to reveal the i'th bit of $\boldsymbol{s}$ and 0 otherwise.

Functionality F_{Table}.

1. On input C from both players, execute the same algorithm as F^{pre}_{sem} would on input C.
2. Let $\boldsymbol{s}_A$ be the string obtained by concatenating all tables $\{A_i\}$ created for A. Similarly $\boldsymbol{s}_B$ contains all B's tables. At any later point, whenever both players input an index set pointing to a substring of $\boldsymbol{s}_A$ or $\boldsymbol{s}_B$, output the substring to both players.

Fig. 9. Functionality giving on-line access to tables

We then compute a representation $[I]$ (which is trivial using the additive shares of $\boldsymbol{a}$), use the MiniMac protocol to compute $[I * \boldsymbol{s}]$ and then open this representation to reveal $I * \boldsymbol{s}$ which gives B the string he needs to know and nothing more. This is possible if we let the preprocessing supply appropriate correlated randomness for the multiplication of I and $\boldsymbol{s}$.

As for the on-line efficiency of this protocol, note that in [DZ13] the MiniMac protocol is claimed to be efficient only for so-called well-formed circuits, but this is not a problem here since the circuit we need to compute is a completely regular depth 1 circuit. Indeed, bit-wise multiplication of strings is exactly the operation MiniMac was designed to do efficiently. Therefore, simple inspection of [DZ13] shows that the preprocessing data we need will be of size $O(\ell) = O(s)$, where s is the circuit size, and this is also the communication complexity. The computational complexity is dominated by the time spent on encoding in C. Unfortunately, we do not know codes with the right algebraic properties that

Protocol π_{mal}^{table}.

1. A and B send C as input to F_{table} and get a set of tables $\{A_i, B_i \mid i = 1 \dots N\}$, as well as a bit b_u for each input wire w_u.
2. For each input wire w_u, if A holds input x_u for this wire, send $e_u = x_u \oplus b_u$ to B. If B holds input x_u, send $e_u = x_u \oplus b_u$ to A.
3. Define M such that the last $N - M$ gates are output gates. For $i = 1$ to M do:
   (a) Let G_i have input wires w_u, w_v and output wire w_o (so that e_u, e_v have been computed).
   (b) A sends $e_A = A_i[e_u, e_v]$ to B.
   (c) B sends $e_B = B_i[e_u, e_v]$ to A.
   (d) Both parties set $e_o = e_A \oplus e_B$.

   Check if all bits opened until now were correct: A and B both ask F_{table} for the substring of A's tables that she was supposed to reveal in the previous steps. B checks against what he received from A and aborts if there is a mismatch. Symmetrically, A and B also ask F_{table} for the substring of B's tables that she was supposed to reveal in the previous steps. A checks against what he received from B and aborts if there is a mismatch.
5. Reveal the outputs. For $i = M + 1$ to N do:
   (a) Let G_i have input wires w_u, w_v and output wire w_o (so that e_u, e_v have been computed).
   (b) A sends $e_A = A_i[e_u, e_v]$ to B.
   (c) B sends $e_B = B_i[e_u, e_v]$ to A.
   (d) Both parties set $e_o = e_A \oplus e_B$.
6. The parties use F_{table} to confirm that the bits sent in the previous step were correct, in the same way as in Step 4. If there was no abort, the parties output the bits $\{e_o \mid G_o \text{ is an output gate}\}$.

Fig. 10. Simple protocol for malicious security.

also have smart encoding algorithms, so the only approach known is to simply multiply by the generator matrix. We can optimize by noting that if $\ell > k^2$ we will always be doing many encodings in parallel, so we can collect all vectors to encode in a matrix and use fast matrix multiplication. With current state of the art, this leads to computational complexity $O(sk^{\epsilon})$ where $\epsilon \approx 0.3727$.

Alternatively, we can let C be a Reed-Solomon code over an extension field with $\Omega(k)$ elements. We can then use FFT algorithms for encoding and then all complexities will be $O(\text{polylog}(k)s)$.

In Figure 8 we specify the preprocessing functionality $F_{MiniMac}^{pre}$ we assumed here, i.e., it outputs the tables as well as MiniMac representations of them. Let $\pi_{MiniMac}$ denote the protocol we just sketched here, and consider the functionality F_{table} from Figure 9 that simply stores the tables and outputs bits from them on request. Now, by trivial modification of the security proof for MiniMac, we have

Lemma 1. *The protocol $\pi_{MiniMac}$ composed with $F_{MiniMac}^{pre}$ implements F_{table} with statistical security against a malicious adversary.*

Finally, consider the protocol π_{mal}^{table} from Figure 10. By trivial adaptation of the proof for semi-honest security, we get that

Lemma 2. *The protocol π_{mal}^{table} composed with F_{table} implements F_{SFE} with malicious and statistical security.*

We can then combine the two lemmas to get a protocol for F_{SFE} in the preprocessing model, and we then have:

Theorem 2. *There exists 2-party protocol in the preprocessing model (using $F_{MiniMac}^{pre}$) for computing any Boolean circuit of size s with malicious and statistical security, where the preprocessed data size and communication complexity are $O(s)$ and the computational complexity is $O(k^{\epsilon} s)$ where k is the security parameter and $\epsilon < 1$. There also exists a protocol for which all complexities are $O(\text{polylog}(k)s)$.*

References

BDOZ11. Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic Encryption and Multiparty Computation. In *Proceedings of EuroCrypt*, pages 169–188, Springer Verlag 2011.

CDvdG87. David Chaum, Ivan Damgård, and Jeroen van de Graaf. Multiparty computations ensuring privacy of each party's input and correctness of the result. In *CRYPTO*, volume 293 of *Lecture Notes in Computer Science*, pages 87–119. Springer, 1987.

DK10. Ivan Damgård and Marcel Keller. Secure multiparty AES. In *Financial Cryptography*, volume 6052 of *Lecture Notes in Computer Science*, pages 367–374. Springer, 2010.

DLT14. Ivan Damgård, Rasmus Lauritsen, and Tomas Toft. An empirical study and some improvements of the minimac protocol for secure computation. In *SCN*, volume 8642 of *Lecture Notes in Computer Science*, pages 398–415. Springer, 2014.

DPSZ12. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In *Proceedings of Crypto*, pages 643–662, Springer Verlag 2012.

DZ13. Ivan Damgård and Sarah Zakarias. Constant-Overhead secure computation of boolean circuits using preprocessing. In *Theory of Cryptography*, pages 621–641. Springer, 2013.

DZ16. Ivan Damgård and Rasmus Winther Zakarias. Fast oblivious AES A dedicated application of the minimac protocol. In *AFRICACRYPT*, volume 9646 of *Lecture Notes in Computer Science*, pages 245–264. Springer, 2016.

FKOS15. Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl. A unified approach to MPC with preprocessing using OT. In *ASIACRYPT (1)*, volume 9452 of *Lecture Notes in Computer Science*, pages 711–735. Springer, 2015.

Kel. Marcel Keller. private communication.

NNOB12. Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A New Approach to Practical Active-Secure Two-Party Computation. In *Proceedings of Crypto*, pages 681–700, Springer Verlag 2012.