A Secret-Sharing Based MPC Protocol for Boolean Circuits with Good Amortized Complexity

Ignacio Cascudo[1] and Jaron Skovsted Gundersen[2]

[1] IMDEA Software Institute, Madrid, Spain, ignacio.cascudo@imdea.org
[2] Department of Mathematical Sciences, Aalborg University, Denmark, jaron@math.aau.dk

Abstract We present a new secure multiparty computation protocol that allows for evaluating a number of instances of a boolean circuit in parallel, with a small online communication complexity per instance of 10 bits per party and multiplication gate. Our protocol is secure against an active adversary corrupting a dishonest majority. The protocol uses an approach introduced recently in the setting of honest majority and information-theoretically security, based on the algebraic notion known as reverse multiplication friendly embeddings, which essentially transforms a batch of evaluations of an arithmetic circuit over a small field into one evaluation of another arithmetic circuit over a larger field. To obtain security against a dishonest majority, we combine this approach with the well-known SPDZ protocol that provides security against a dishonest majority but operates over a large field. As SPDZ and its variants, our protocol operates in the preprocessing model. Structurally our protocol is most similar to MiniMAC, a protocol which bases its security on the use of error-correcting codes, but our protocol has a communication complexity which is half of that of MiniMAC when the best available binary codes are used. With respect to certain variant of MiniMAC that utilizes codes over larger fields, our communication complexity is slightly worse; however, that variant of MiniMAC needs a much larger preprocessing than ours. We also show that our protocol also has smaller amortized communication complexity than Committed MPC, a protocol for general fields based on homomorphic commitments, if we use the best available constructions for those commitments. Finally, we construct a preprocessing phase from oblivious transfer based on ideas from MASCOT and Committed MPC.

1 Introduction

The area of secure multiparty computation (MPC) studies how to design protocols that allow for a number of parties to jointly perform computations on private inputs in such a way that each party learns a private output, but nothing else than that.

In the last decade efficient MPC protocols have been developed that can be used in practical applications. In particular, in this work we focus on secret-sharing based MPC protocols, which are among the most used in practice. In secret-sharing based MPC, the target computation is represented as an arithmetic circuit consisting of sum and multiplication gates over some algebraic ring; each of the parties initially shares her input among the set of parties, and the secure computation proceeds gate by gate, where at every gate a sharing of the output of the gate is created; in this manner eventually parties obtain shares of the output of the computation, which can be then reconstructed.

A common practice is to use the preprocessing model, where the computation is divided in two stages: a preprocessing phase, that is completely independent from the inputs and

whose purpose is to distribute some correlated randomness among the parties; and an online phase, where the actual computation is performed with the help of the preprocessing data. This approach allows for pushing much of the complexity of the protocol into the preprocessing phase and having very efficient online computations in return.

Some secret sharing based MPC protocols obtain security against any static adversary which actively corrupts all but one of the parties in the computation, assuming that the adversary is computationally bounded. Since in the active setting corrupted parties can arbitrarily deviate from the protocol, some kind of mechanism is needed to detect such malicious behaviour, and one possibility is the use of information-theoretic MACs to authenticate the secret shared data, which is used in protocols such as BeDOZa [2] and SPDZ [13].

In SPDZ this works as follows: the computation to be performed is given by an arithmetic circuit over a large finite field $\mathbb{F}$. There is a global key $\alpha \in \mathbb{F}$ which is secret shared among the parties. Then for every value $x \in \mathbb{F}$ in the computation, parties obtain not only additive shares for that value, but also for the product $\alpha \cdot x$ which acts as a MAC for x. The idea is that if a set of corrupt parties change their shares and pretend that this value is $x + e$, for some nonzero error e, then they would also need to guess the correction value $\alpha \cdot e$ for the MAC, which amounts to guessing α since $\mathbb{F}$ is a field. In turn this happens with probability $1/|\mathbb{F}|$ which is small when the field is large.

Unfortunately, the same strategy does not work for other domains: over other rings of interest, for example over the modular ring $\mathbb{Z}/2^k\mathbb{Z}$ of integers modulo 2^k, this fails because of the ring having many divisors of zero (guessing $\alpha \cdot e$ for a chosen e is much easier than guessing α) so alternative authentication methods have been developed, see [11].

Over small fields, for example $\mathbb{F} = \mathbb{F}_2$ the problem is simply that the cheating success probability $1/|\mathbb{F}|$ is large. Of course, one can always take a large enough field $\mathbb{L}$ which is an extension field of $\mathbb{F}$ and embed the whole computation into that field. But this is wasteful: if we want to securely compute a boolean circuit, and we want the cheating success probability to be 2^{-s}, we need to do all our computations in the field of size 2^s, so communication is blown up by a factor of s.

An alternative was proposed in MiniMAC [14]. MiniMAC uses a batch authentication idea, that we describe now: if we are willing to securely compute k instances of the same arithmetic circuit over a small field at once, we can bundle these computations together and see them as a computation of an arithmetic circuit over the ring $\mathbb{F}^k$, where the sum and multiplication operations are considered coordinatewise. Note the same authentication technique as in SPDZ does not however directly work over this ring, again because of it having divisors of zero: now share data are vectors $\mathbf{x}$ in $\mathbb{F}^k$, and if we define their MACs as $\boldsymbol{\alpha} * \mathbf{x}$ where the key $\boldsymbol{\alpha}$ is now also a vector in $\mathbb{F}^k$ and * is the coordinatewise product, this authentication method can be fooled with probability $1/|\mathbb{F}|$, since the adversary only needs to introduce an error in one coordinate, which can be achieved successfully by guessing the corresponding coordinate of $\boldsymbol{\alpha}$. In order to solve this, MiniMAC first encodes every vector $\mathbf{x}$ as a larger vector $C(\mathbf{x})$ by means of a linear error-correcting code C with large minimum distance d, and then defines the MAC as $\boldsymbol{\alpha} * C(\mathbf{x})$. The point is that now introducing an error in the vector $\mathbf{x}$ requires to change at least d coordinates of the corresponding encoding to stay within the error correcting code. Therefore fooling this MAC requires to guess at least d different coordinates of $\boldsymbol{\alpha}$, which can be successfully done with probability only $1/|\mathbb{F}|^d$, so we can make this probability as small as we want by increasing d. An important observation, however, is that, when processing multiplication gates, some data needs to be temporarily authenticated with the so-called square C^* (or Schur square) of the code C and hence we

also need the minimum distance d^* of this other code to be large. These requirements on the minimum distance of these two codes have an effect on the communication overhead of the protocol, because the larger the distance of a code is, the worse the relation between dimension (the length of messages k) and length (the length of the encoding) of the code.

At this point, one can wonder why not to just to "see the vector $\mathbf{x}$ in $\mathbb{F}^k$ as an element in a extension field of $\mathbb{F}$ of degree k" and use SPDZ in that extension. That is, fixing from now on $\mathbb{F} = \mathbb{F}_2$ for simplicity, the ring $\mathbb{F}_2^k$ consisting of vectors of k elements in $\mathbb{F}_2$ (with coordinatewise sum) and the field $\mathbb{F}_{2^k}$ (with its usual field sum) are isomorphic as vector spaces over $\mathbb{F}_2$. So one can fix such an isomorphism ι satisfying that $\iota(\mathbf{x}) + \iota(\mathbf{y}) = \iota(\mathbf{x} + \mathbf{y})$. Unfortunately, however, this map cannot "preserve" multiplications, i.e., the two sets are not isomorphic as rings (for example, the ring $\mathbb{F}_2^k$ has divisors of zero, while $\mathbb{F}_{2^k}$ does not). This causes that $\iota(\mathbf{x}) \cdot \iota(\mathbf{y})$ (where $\cdot$ is the field product) will necessarily "destroy" information about $\mathbf{x} * \mathbf{y}$ (where $*$ is coordinatewise product on vectors), which would be our target computation when we are to compute k instances of a boolean multiplication gate in parallel.

However, an idea recently introduced in the honest majority, information-theoretically secure setting [7], can provide us with an alternative approach. The point is that while $\mathbb{F}_2^k$ is not isomorphic as a ring to $\mathbb{F}_{2^k}$, one can instead embed $\mathbb{F}_2^k$ into a slightly larger field $\mathbb{F}_{2^m}$ with some dedicated linear "embedding" map ϕ, that satisfies that for any two vectors $\mathbf{x}, \mathbf{y}$ in $\mathbb{F}_2^k$ the field product $\phi(\mathbf{x}) \cdot \phi(\mathbf{y})$ contains all information about $\mathbf{x} * \mathbf{y}$, in fact there exists a "recovery" linear map ψ such that $\mathbf{x} * \mathbf{y} = \psi(\phi(\mathbf{x}) \cdot \phi(\mathbf{y}))$. The pair (ϕ, ψ) is called an (k, m)-reverse multiplication friendly embedding (RMFE). With such tool, [7] proceeds to embed a simultaneous computation of k evaluation of a boolean circuit (i.e. an evaluation of an arithmetic circuit over $\mathbb{F}_2^k$ with coordinatewise operations) into one evaluation of a related circuit over $\mathbb{F}_{2^m}$, which is securely computed via an MPC protocol for arithmetic circuits over that larger field (more precisely the Beerliova-Hirt protocol [1]). The use of that MPC protocol over $\mathbb{F}_{2^m}$ is not black-box, however, as there a number of modifications that need to be done, which we will explain later, and that require certain additional correlated information to be created in the preprocessing phase. Note that the motivation for the introduction of this technique in the information-theoretical setting [7] was not related to authentication via MACs: the goal of that paper was to match, over small fields and in an amortized sense, the communication complexity of the Beerliova-Hirt protocol which uses two tools (Shamir secret sharing and hyperinvertible matrices) which are only available when the size of the field is at least the number of parties involved in the protocol.

1.1 Our contributions

In this paper we construct a new secure computation protocol in the dishonest majority setting that allows to compute several instances of a boolean circuit at an amortized cost.[1] We do this by combining the embedding techniques from [7] with the SPDZ methodology. As opposed to [7], where one of the points of the embedding was to use Shamir secret sharing, in our construction, vectors $\mathbf{x} \in \mathbb{F}_2^k$ are additively shared in $\mathbb{F}_2^k$, and it is only the MACs which are constructed and shared in the field $\mathbb{F}_{2^m}$. More precisely, the MAC of $\mathbf{x}$ will be $\alpha \cdot \phi(\mathbf{x})$ where ϕ is the embedding map from the RMFE. Only when processing a multiplication gate, authenticated sharings where the data are shared as elements in $\mathbb{F}_{2^m}$ are temporarily used. Throughout most of the online phase, authenticated sharings are partially opened (i.e. only the data values, not the MACs, are revealed). Only at the output phase, MACs are checked in a batched fashion, at which point the protocol aborts if discrepancies are found.

[1] Our ideas can of course be extended to arithmetic circuits over other small fields.

By this method we obtain a very efficient online phase where processing multiplication gates need each party to communicate around 10 bits[2] per evaluation of the circuit, for statistical security parameters like $s = 64, 128$ (meaning the adversary can succcessfully cheat with probability at most 2^{-s}, for which in our protocols we need to set $m \geq s$).

Our online phase ends up following a similar pattern to that of MiniMAC. Up to the output phase, every partial opening of a value in $\mathbb{F}_2^k$ takes place when a partial opening of a C-encoding occurs in MiniMAC. Respectively, we need to open values in $\mathbb{F}_{2^m}$ whenever MiniMAC opens C^*-encodings. At every multiplication gate, both protocols need to apply reencoding functions to convert encodings back to the base authentication scheme, which requires a preprocessed pair of authenticated sharings of random correlated elements.

However, the encoding via RMFE we are using is more compact than using binary linear error correcting codes with large minimum distance of the Schur square, and the comparison boils down to comparing the "expansion factor" m/k of RMFEs with the ratio k^*/k between the dimensions of C^* and C for the best binary codes with good distances of C^* [6]. We find that we can cut the communication cost of multiplication gates by about half with respect to MiniMAC where those binary codes are used. We achieve even better savings in the case of the output gates since in this case MiniMAC needs to communicate full vectors of the same length as the code, while the input and addition gates have the same cost.

We also compare the results with a modified version of MiniMAC proposed by Damgård, Lauritsen and Toft [12], that allows to save communication cost of multiplication gates, by essentially using MiniMAC over the field of 256 elements, at the cost of a much larger amount of preprocessing that essentially provides authenticated sharings of bit decompositions of the $\mathbb{F}_{256}$-coordinates of the elements in a triple, so that parties can compute bitwise operations. This version achieves a communication complexity that is around 80% of that of our protocol, due to the fact that this construction can make use of Reed-Solomon codes. However, it requires to have created authenticated sharings of 19 elements, while ours need 5 and as far as we know there is no explicit preprocessing protocol that has been proposed for this version of MiniMAC.

Finally we compare the results with Committed MPC, a secret-sharing based protocol which uses (UC-secure) homomorphic commitments for authentication, rather than information-theoretical MACs. In particular, this protocol can also be used for boolean circuits, given that efficient constructions of homomorphic commitments [16,9,8] over $\mathbb{F}_2$ have been proposed. These constructions of homomorphic commitments also use error-correcting codes. We find that, again, the smaller expansion m/k of RMFE compared to the relations between the parameters for binary error-correcting codes provides an improvement in the communication complexity, of about a factor 3 for security parameter 64.

We also provide a preprocessing phase producing all authenticated sharings of random correlated data that we need. The preprocessing follows the steps of MASCOT [19] (see also [17]) based on OT extension, with some modifications due to the slightly different authentication mechanisms we have and the different format of our preprocessing. All these modifications are easily to carry out based on the fact that ϕ and ψ are linear maps over $\mathbb{F}_2$. Nevertheless, using the "triple sacrificing steps" from MASCOT that assure that preprocessed triples are not malformed presents problems in our case for a number of reasons. Instead, we use the techniques from Committed MPC [15] in that part of the triple generation.

[2] Here we assume that broadcasting messages of M bits requires to send M bits to every other player, which one can achieve with small overhead that vanishes for large messages [13, full version]

1.2 Related Work

The use of information-theoretical MACs in secret-sharing based multiparty computation dates back to BeDOZa (Bendlin et al., [2]), where such MACs where established between every pair of players. Later SPDZ (Damgård et al., [13]) introduced the strategy consisting of a global MAC for every element of which every party has a share, and whose key is likewise shared among parties. Tiny OT (Nielsen et al., [21]), a 2-party protocol for binary circuits, introduced the idea of using OT extension in the preprocessing phase. Larraia et al. [20] extended these ideas to a multi-party protocol by using the SPDZ global shared MAC approach. MiniMAC (Damgård and Zakarias, [14]), as explained above, used error-correcting codes in order to authenticate vectors of bits, allowing for efficient parallel computation of several evaluations of the same binary circuits on possibly different inputs. Damgård et al. [12] proposed several improvements for the implementation of MiniMAC, among them the use of an error correcting code over an extension field, trading smaller communication complexity for a larger amount of preprocessing. Frederiksen et al. [17] gave new protocols for the construction of preprocessed multiplication triples in fields of characteristic two, based on OT extension, and in particular provided the first preprocessing phase for MiniMAC. MASCOT (Keller et al., [19]) built on some of these ideas to create preprocessing protocols for SPDZ based on OT extension. Committed MPC (Frederiksen et al. [15]) is a secret-sharing based secure computation protocol that relies on UC-secure homomorphic commitments instead of homomorphic MACs for authentication, but other than that, it follows a similar pattern to the protocols above. Efficient constructions of UC-secure homomorphic commitments from OT have been proposed by Frederiksen et al. [16] and Cascudo et al. [9] based on error correcting codes. Later, Cascudo et al. [8] proposed a modified construction from extractable commitments, still using error-correcting codes, that presents an important advantage for its use in Committed MPC, namely the commitment schemes are multi-verifier, see [8].

The notion of reverse multiplication friendly embedding was first explicitly defined and studied in the context of secure computation by Cascudo et al. in [7] and independently by Block et al. in [4]. The former work is in the context of information-theoretically secure protocols, as explained above, while the latter studied 2-party protocols over small fields where the assumed resource is OLE over an extension field. This latter work is partially based on a previous one by the same authors [3] where (asymptotically less efficient) constructions of multiplication friendly embeddings were implicitly used.

2 Preliminaries and Notation

Let $\mathbb{F}_q$ denote a finite fields with q elements. Vectors are denoted with bold letters as $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ and componentwise products of two vectors are denoted by $\mathbf{x} * \mathbf{y} = (x_1 \cdot y_1, x_2 \cdot y_2, \ldots, x_n \cdot y_n)$. Fixing an irreducible polynomial f of degree m in $\mathbb{F}_q[X]$, the elements in the finite field with q^m elements can be represented as polynomials with degree lower than m and coefficients from $\mathbb{F}_q$, i.e $\alpha = \alpha_0 + \alpha_1 \cdot X + \cdots + \alpha_{m-1} \cdot X^{m-1} \in \mathbb{F}_{q^m}$, where the sums and products are defined modulo f.

In our protocols we will assume a network of n parties who communicate by secure point-to-point channels, and an static adversary who can actively corrupt up to $n-1$ of these parties. Our proofs will be in the universal composable security model [5] (see Appendix A for a brief description of that model).

2.1 Reverse multiplication friendly embeddings

Our goal is a protocol that simultaneously evaluates k instances of a given boolean circuit securely against up to $(n-1)$ actively corrupted parties where n is the number of parties participating in the protocol.

For this we will bundle up together the k instances of the boolean circuit and embed them in one instance of an arithmetic circuit over $\mathbb{F}_{2^m}$, which we will compute securely using the structure of SPDZ. In order to do this, we will use the notion of reverse multiplication friendly embeddings from [7].

Definition 1. *Let $\mathbb{F}_q$ be the finite field with q elements, and let $k, m \in \mathbb{Z}^+$. A pair of $\mathbb{F}_q$-linear maps (ϕ, ψ), where $\phi\colon \mathbb{F}_q^k \to \mathbb{F}_{q^m}$ and $\psi\colon \mathbb{F}_{q^m} \to \mathbb{F}_q^k$ is called a $(k,m)_q$-reverse multiplication friendly embedding (RMFE) if they satisfy*

$$\mathbf{x} * \mathbf{y} = \psi(\phi(\mathbf{x}) \cdot \phi(\mathbf{y}))$$

for all $\mathbf{x}, \mathbf{y} \in \mathbb{F}_q^k$.

In other words, this tool allows to multiply coordinatewise two vectors over $\mathbb{F}_q$ by first embedding them in a larger field with ϕ, multiplying the resulting images and mapping the result back to a vector over $\mathbb{F}_q$ with the other map ψ.

The following results about the existence of such pairs can be found in [7].

Theorem 1 ([7]).

1. *(Concrete parameters, $q = 2$). For all $r \leq 33$, there exists a $(3r, 10r-5)_2$ -RMFE*
2. *(Concrete parameters, general q). For all $1 \leq k \leq q+1$, where q is a prime power, there exists a $(k, 2k-1)_q$-RMFE.*
3. *(Asymptotic parameters, $q = 2$). There exists a family of $(k,m)_2$-RMFE where $k \to \infty$ and $m/k \to 4.92...$*
4. *(Asymptotic parameters, general q). For every prime power q, there exists a family of $(k,m)_q$-RMFE where $m = \Theta(k)$.*

We remark that while the asymptotic results require algebraic geometry, the results 1 and 2 are constructed from elementary interpolation techniques.

2.2 The MPC embedding technique in [7]

The protocol follows the embedding strategy of the information-theoretically secure protocol from [7], and it is important to understand the general idea of that work, and the hurdles that are overcome there.

A parallel evaluation of k instances of a boolean circuit can equivalently be seen as one evaluation of an arithmetic circuit over the ring $\mathbb{F}_2^k$, where sums and products are componentwise (and additions with the constant 1 are mapped to additions with the constant all-one vector). Call this circuit C. Now we want to construct an arithmetic circuit $\widetilde{C}$ over the field $\mathbb{F}_{2^m}$ so that we can later embed an evaluation of C into an evaluation of $\widetilde{C}$ using the RMFE, meaning that we would take the inputs to C, map them into $\mathbb{F}_{2^m}$ using the map ϕ from the RMFE and evaluate $\widetilde{C}$ on them. Replacing (coordinatewise) sum and product gates in $\mathbb{F}_2^k$ by sum and products gates in $\mathbb{F}_{2^m}$ would fall short. The reason is that the multiplication property of the RMFE only works for products of exactly two elements. So

for example, in general $\mathbf{x} * \mathbf{y} * \mathbf{z} \neq \psi(\phi(\mathbf{x}) \cdot \phi(\mathbf{y}) \cdot \phi(\mathbf{z}))$ (and this naturally extends to more factors). Moreover, generally $\mathbf{x} \neq \psi(\phi(\mathbf{x}))$. See [7] for more details.

The solution to this is to replace every multiplication gate in $\mathbb{F}_2^k$ by a "multiplication gadget" that consists of the concatenation of a multiplication gate in $\mathbb{F}_{2^m}$, followed by a "refreshing" gate computing the concatenation map $\phi \circ \psi : \mathbb{F}_{2^m} \to \mathbb{F}_{2^m}$. The addition gates over $\mathbb{F}_2^k$ are simply replaced by addition gates over $\mathbb{F}_{2^m}$. This constitutes our new arithmetic circuit $\widetilde{C}$.

The evaluation strategy is then as follows: the inputs (as well as the constant all-one vectors) are mapped from $\mathbb{F}_2^k$ into $\mathbb{F}_{2^m}$ with ϕ. We evaluate the circuit $\widetilde{C}$ over $\mathbb{F}_{2^m}$ described above, and in the last step we apply ϕ^{-1}. The rationale is that, in such an evaluation, every wire of this circuit over $\mathbb{F}_{2^m}$ (consisting of additive gates and multiplication gadgets) contains a value of the form $\phi(\mathbf{x})$ where $\mathbf{x}$ is the value that would travel in that wire for the corresponding evaluation of the circuit C over $\mathbb{F}_2^k$. The justification for this invariant is that in the case of additive gates this holds by linearity of ϕ, and in the case of multiplication gadgets,

$$\phi(\mathbf{x} * \mathbf{y}) = \phi(\psi(\phi(\mathbf{x}) \cdot \phi(\mathbf{y}))) = (\phi \circ \psi)(\phi(\mathbf{x}) \cdot \phi(\mathbf{y})),$$

the first equality being a consequence of the RMFE definition.

Applying a secure computation protocol for an arithmetic circuit over $\mathbb{F}_{2^m}$ presents two obstacles, as discussed in [7]: one is computing the map $\phi \circ \psi$ from the multiplication gadgets, because this map is only linear over $\mathbb{F}_2$, and not over $\mathbb{F}_{2^m}$; the other is that parties need to prove that they have embedded their inputs correctly in $\mathbb{F}_{2^m}$ using ϕ, as a standard MPC protocol over $\mathbb{F}_{2^m}$ is oblivious to whether the inputs are in a particular subset of the field (in this case the image of ϕ). Both issues can be solved efficiently by use of an involved preprocessing step with generates randomness in $\mathbb{F}_2$-linear subspaces of $\mathbb{F}_{2^m}$, which involves the combination of hyperinvertible matrices [1] and tensoring of vector spaces.

3 The online phase

3.1 General idea

In this paper, we want to apply the ideas from [7] in order to construct a MPC protocol secure against an active adversary corrupting a majority (up to all but one) of the parties. As a basis we want to use the online phase of the SPDZ protocol. We have briefly discussed the authentication technique in SPDZ and the protocol is well known, so we refer the reader to [13] for more details, and we now explain how we combine both aforementioned elements.

Recall that our goal is to compute k instances of a boolean circuit in parallel, which we can in turn see as computing an instance of an arithmetic circuit over the ring $\mathbb{F}_2^k$ with coordinatewise sum and product, as explained above. While in [7], the inputs $\mathbf{x} \in \mathbb{F}_2^k$ would be immediately mapped into $\mathbb{F}_{2^m}$ via the map ϕ from the RMFE, and shared there (and subsequently the same would happen to all intermediate values in the computation), in our case this is not necessary, it is wasteful in communication, and in fact complicates the computation of multiplication gates. Instead, we will have mixed authenticated sharings, where inputs and intermediate values $\mathbf{x}$ are additively shared as vectors over $\mathbb{F}_2^k$, but in addition the MAC value $\alpha \cdot \phi(\mathbf{x}) \in \mathbb{F}_{2^m}$ will be additively shared in the field $\mathbb{F}_{2^m}$, where $\alpha \in \mathbb{F}_{2^m}$ is a global key. We will call this authentication $\langle \mathbf{x} \rangle$. The additivity of ϕ guarantees that linear operations can still be computed locally.

The question is how to process multiplication gates; given $\langle \mathbf{x} \rangle$, $\langle \mathbf{y} \rangle$ we need to compute $\langle \mathbf{x} * \mathbf{y} \rangle$ which implies not only obtaining an additive sharing of $\mathbf{x} * \mathbf{y}$ but also of its MAC

$\alpha \cdot \phi(\mathbf{x} * \mathbf{y})$. Now if try to apply directly Beaver's technique we have the following problem. Suppose we have obtained a random triple $\langle \mathbf{a} \rangle, \langle \mathbf{b} \rangle, \langle \mathbf{a} * \mathbf{b} \rangle$ from the preprocessing phase and, proceeding as usual, we open the values $\boldsymbol{\epsilon} = \mathbf{x} - \mathbf{a}$, $\boldsymbol{\delta} = \mathbf{y} - \mathbf{b}$. From here, computing sharings for $\mathbf{x} * \mathbf{y}$ is easy; however, the obstacle lies in computing shares of $\alpha \cdot \phi(\mathbf{x} * \mathbf{y})$. Indeed

$$\alpha \cdot \phi(\mathbf{x} * \mathbf{y}) = \alpha \cdot \phi(\mathbf{a} * \mathbf{b}) + \alpha \cdot \phi(\mathbf{a} * \boldsymbol{\delta}) + \alpha \cdot \phi(\boldsymbol{\epsilon} * \mathbf{b}) + \alpha \cdot \phi(\boldsymbol{\epsilon} * \boldsymbol{\delta}),$$

and while computing the last term is trivial (since it is public) and a sharing for the first term would have been created in the preprocessing, computing sharings for the middle terms is not possible with the information we have: by the properties of the RMFE

$$\alpha \cdot \phi(\mathbf{a} * \boldsymbol{\delta}) = \alpha \cdot \phi(\psi(\phi(\mathbf{a}) \cdot \phi(\boldsymbol{\delta}))) = \alpha \cdot (\phi \circ \psi)(\phi(\mathbf{a}) \cdot \phi(\boldsymbol{\delta}))$$

Now the same problem as in [7] arises: if $\phi \circ \psi : \mathbb{F}_{2^m} \to \mathbb{F}_{2^m}$ were linear over $\mathbb{F}_{2^m}$ we could "take α inside the argument" and use the additive sharing of $\alpha \cdot \phi(\mathbf{a})$ given in $\langle \mathbf{a} \rangle$ to compute a sharing of the expression above. However, $\phi \circ \psi$ is only linear over $\mathbb{F}_2$, not over $\mathbb{F}_{2^m}$, so this does not work.

Instead, we compute product gates using a two step process, for which we need to introduce authenticated sharings of elements in $x \in \mathbb{F}_{2^m}$. These are just regular SPDZ sharings: x and $\alpha \cdot x$ are both additively shared in $\mathbb{F}_{2^m}$. We denote those by $[x]$. In the first step of the multiplication we use a preprocessed triple which has the form $(\langle \mathbf{a} \rangle, \langle \mathbf{b} \rangle, [c])$ where $c = \phi(\mathbf{a}) \cdot \phi(\mathbf{b})$. It can be readily seen that by using this triple, and opening two values as above, parties can create $[\phi(\mathbf{x}) \cdot \phi(\mathbf{y})]$, because all it requires is the $\mathbb{F}_2$-linearity of ϕ (details will be given later). In the second step, we create $\langle \mathbf{x} * \mathbf{y} \rangle$ from $[\phi(\mathbf{x}) \cdot \phi(\mathbf{y})]$. We use the fact that $\mathbf{x} * \mathbf{y} = \psi(\phi(\mathbf{x}) \cdot \phi(\mathbf{y}))$ from the definition of the RMFE. Computing ψ can be done with the aid of a preprocessed pair of the form $[r], \langle \psi(r) \rangle$, by opening one value in $\mathbb{F}_{2^m}$.

The whole multiplication gate will cost 2 openings of sharings of vectors in $\mathbb{F}_2^k$ and one opening of a share of an element in $\mathbb{F}_{2^m}$. We remark that only sharings of the data are opened and not those of their MACs. Every multiplication gate requires fresh preprocessed correlated authenticated sharings $(\langle \mathbf{a} \rangle, \langle \mathbf{b} \rangle, [\phi(\mathbf{a}) \cdot \phi(\mathbf{b})])$ and $[r], \langle \psi(r) \rangle$ for random $\mathbf{a}, \mathbf{b}, r$.

On the other hand, input gates are easily handled if we have created in the preprocessing phase an authenticated sharing of a random value $\langle \mathbf{r} \rangle$ and opened $\mathbf{r}$ to the party that will provide the input. This party can just broadcast the difference between $\mathbf{r}$ and her input.

The output gate is handled as in SPDZ: the parties do a MAC check on a random linear combination of all opened values that ensures that parties have not cheated except with probability at most 2^{-m} (we need that $m \geq s$ if s is the statistical security parameter); then, they will open the result of the computation and will also check that the MAC of the result is correct and accept the result if that is the case.

We differ the description of the preprocessing for later, but we point out that due to the fact that ϕ and ψ are $\mathbb{F}_2$-linear maps, the creation of the needed elements (including their authentication) is well suited to be accomplished through the techniques from [17,19] relying on OT extension.

In comparison again to the information-theoretical protocol from [7] we avoid altogether the problem that parties need to prove that their inputs are computed correctly as images of ϕ (since here we keep the computation on $\mathbb{F}_2^k$ and the final opening will ensure that the MAC is correctly computed); while the obstacle occurring at multiplication gates as a consequence that $\phi \circ \psi$ is not $\mathbb{F}_{2^m}$-linear resurfaces in our case, but we overcome this with a

different preprocessing requirement. Finally, we do not have a need for the involved tensoring techniques from [7], which were needed because of the use of hyperinvertible matrices which were defined over the extension field $\mathbb{F}_{2^m}$. Here OT-extension based preprocessing techniques such as MASCOT are amenable to work with $\mathbb{F}_2$-linear operations directly.

3.2 Authenticated sharings and operations

We describe more precisely the two types of authenticated sharings we used, and the algebraic operations involving them.

In the case of vectors $\mathbf{x} \in \mathbb{F}_2^k$, we define

$$\langle \mathbf{x} \rangle = \left((\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(n)}), (m^{(1)}(\mathbf{x}), m^{(2)}(\mathbf{x}), \ldots, m^{(n)}(\mathbf{x})) \right), \tag{1}$$

where each party P_i will hold an additive share $\mathbf{x}^{(i)} \in \mathbb{F}_2^k$ and a MAC share $m^{(i)}(\mathbf{x}) \in \mathbb{F}_{2^m}$, such that

$$\sum_{i=1}^{n} m^{(i)}(\mathbf{x}) = \alpha \cdot \sum_{i=1}^{n} \phi(\mathbf{x}^{(i)}),$$

where $\phi \colon \mathbb{F}_2^k \to \mathbb{F}_{2^m}$ is the map from the RMFE.

On the other hand, in the case of the secret being a field element $x \in \mathbb{F}_{2^m}$, we define

$$[x] = \left((x^{(1)}, x^{(2)}, \ldots, x^{(n)})), (m^{(1)}(x), m^{(2)}(x), \ldots, m^{(n)}(x)) \right), \tag{2}$$

where P_i will hold $x^{(i)} \in \mathbb{F}_{2^m}$ and $m^{(i)}(x) \in \mathbb{F}_{2^m}$ such that

$$\sum_{i=1}^{n} m^{(i)}(x) = \alpha \cdot \sum_{i=1}^{n} x^{(i)}.$$

Namely, this is the SPDZ authenticated sharing.

We remark that we use the notation $m^{(i)}$ in both cases but the "input" tells what case we are in. Furthermore, we denote $m(\mathbf{x}) = \sum_{i=1}^{n} m^{(i)}(\mathbf{x})$ and similarly $m(x) = \sum_{i=1}^{n} m^{(i)}(x)$.

Note that the MAC shares in $\langle \mathbf{x} \rangle$ and $[\phi(\mathbf{x})]$ are distributed in the same way. However, the data shares of the former are vectors in $\mathbb{F}_2^k$ while the shares of the latter are in $\mathbb{F}_{2^m}$, and they are not even restricted to be in the image of ϕ (so there is more entropy in them). However, given $y \in \mathbb{F}_{2^m}$, if we define the following operation:

$$\begin{aligned} \langle \mathbf{x} \rangle + [y] = \Big(& (\phi(\mathbf{x}^{(1)}) + y^{(1)}, \ldots, \phi(\mathbf{x}^{(n)}) + y^{(n)}), \\ & (m^{(1)}(\mathbf{x}) + m^{(1)}(y), \ldots, m^{(n)}(\mathbf{x}) + m^{(n)}(y)) \Big) \end{aligned}$$

then we can write $\langle \mathbf{x} \rangle + [y] = [\phi(\mathbf{x}) + y]$ in the sense that the data sharings are now equally distributed.

As usual (because $[\cdot]$ is the regular SPDZ MAC) we can also define the following operations involving only $[\cdot]$:

$$\begin{aligned}
[x] + [y] &= \Big((x^{(1)} + y^{(1)}, \ldots, x^{(n)} + y^{(n)}), \\
&\qquad (m^{(1)}(x) + m^{(1)}(y), \ldots, m^{(n)}(x) + m^{(n)}(y))\Big) \\
a + [x] &= \Big((x^{(1)} + a, x^{(2)}, \ldots, x^{(n)}), \\
&\qquad (\alpha^{(1)} \cdot a + m^{(1)}(x), \ldots, \alpha^{(n)} \cdot a + m^{(n)}(x))\Big) \\
a \cdot [x] &= \Big((a \cdot x^{(1)}, \ldots, a \cdot x^{(n)}), (a \cdot m^{(1)}(x), \ldots, a \cdot m^{(n)}(x))\Big).
\end{aligned}$$

for elements $x, y, a \in \mathbb{F}_{2^m}$, where we recall that $\alpha^{(1)}, \ldots, \alpha^{(n)}$ are the additive shares for the global MAC key α.

In the case of vectors authenticated with $\langle\cdot\rangle$, we can in first place define the following operations:

$$\begin{aligned}
\langle \mathbf{x} \rangle + \langle \mathbf{y} \rangle &= \Big((\mathbf{x}^{(1)} + \mathbf{y}^{(1)}, \ldots, \mathbf{x}^{(n)} + \mathbf{y}^{(n)}), \\
&\qquad (m^{(1)}(\mathbf{x}) + m^{(1)}(\mathbf{y}), \ldots, m^{(n)}(\mathbf{x}) + m^{(n)}(\mathbf{y}))\Big) \\
&\qquad = \langle \mathbf{x} + \mathbf{y} \rangle, \\
\mathbf{a} + \langle \mathbf{x} \rangle &= \Big((\mathbf{x}^{(1)} + \mathbf{a}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(n)}), \\
&\qquad (\alpha^{(1)} \cdot \phi(\mathbf{a}) + m^{(1)}(\mathbf{x}), \ldots, \alpha^{(n)} \cdot \phi(\mathbf{a}) + m^{(n)}(\mathbf{x}))\Big) \\
&\qquad = \langle \mathbf{a} + \mathbf{x} \rangle,
\end{aligned}$$

for vectors $\mathbf{x}, \mathbf{y}, \mathbf{a} \in \mathbb{F}_2^k$. Finally, we need the following operations:

$$\begin{aligned}
y \cdot \langle \mathbf{x} \rangle &= \Big((y \cdot \phi(\mathbf{x}^{(1)}), \ldots, y \cdot \phi(\mathbf{x}^{(n)})), (y \cdot m^{(1)}(\mathbf{x}), \ldots, y \cdot m^{(n)}(\mathbf{x}))\Big), \\
\mathbf{a} * \langle \mathbf{x} \rangle &= \Big((\phi(\mathbf{a}) \cdot \phi(\mathbf{x}^{(1)}), \ldots, \phi(\mathbf{a}) \cdot \phi(\mathbf{x}^{(n)})), \\
&\qquad (\phi(\mathbf{a}) \cdot m^{(1)}(\mathbf{x}), \ldots, \phi(\mathbf{a}) \cdot m^{(n)}(\mathbf{x}))\Big)
\end{aligned}$$

for vectors $\mathbf{x}, \mathbf{a} \in \mathbb{F}_2^k$ and an element $y \in \mathbb{F}_{2^m}$. These expressions are "almost" equal respectively to $[y \cdot \phi(\mathbf{x})]$ and $[\phi(\mathbf{a}) \cdot \phi(\mathbf{x})]$ except that again the data sharings of the corresponding elements are not uniform in $\mathbb{F}_{2^m}$. However, given an element $z \in \mathbb{F}_{2^m}$ we have:

$$\begin{aligned}
y \cdot \langle \mathbf{x} \rangle + [z] &= [y \cdot \phi(\mathbf{x}) + z] \\
\mathbf{a} * \langle \mathbf{x} \rangle + [z] &= [\phi(\mathbf{a}) \cdot \phi(\mathbf{x}) + z].
\end{aligned} \tag{3}$$

By a partial opening, we will mean that only the shared values and not the MACs are opened. To partially open $\langle \mathbf{x} \rangle$ party P_i sends $\mathbf{x}^{(i)}$ to P_1 who can reconstruct and broadcast $\mathbf{x}$. Opening $[x]$ is analogous.

3.3 Functionalities and protocols for the online phase

We describe now more formally the functionalities and protocols involved in the online phase.

As we described before, for each input vector from party P_i we need to have obtained from the preprocessing a pair $(\mathbf{r}, \langle \mathbf{r} \rangle)$ where $\mathbf{r} \in \mathbb{F}_2^k$ is a random vector known by P_i; furthermore, in order to process multiplication gates we need pairs $(\langle \psi(r) \rangle, [r])$ for random elements $r \in \mathbb{F}_{2^m}$ and multiplication triples $(\langle \mathbf{a} \rangle, \langle \mathbf{b} \rangle, [c])$ where $c = \phi(\mathbf{a}) \cdot \phi(\mathbf{b})$. The functionality constructing the required preprocessed randomness is given in Figure 2, and relies on the authentication functionality in Figure 1. The latter augments the one in MASCOT [19] allowing to also authenticate vectors and to compute linear combinations involving the two different types of authenticated values (which capture the operations described above for the SPDZ-like MACs). We will later realize this authentication functionality by means of the $[\cdot]$- and $\langle \cdot \rangle$-representations, which allow for this type of algebraic operations as described before.

Functionality $\mathcal{F}_{\text{Auth}}$

The functionality maintains two dictionaries Val and ValField, to keep track of authenticated values. We remark that we can store elements from $\mathbb{F}_2^k$ in Val and elements from $\mathbb{F}_{2^m}$ in ValField. Entries in the dictionaries cannot be changed.

1. **Input:** On input
$$(\text{Input}, (\text{id}_1, \text{id}_2, \dots \text{id}_s), (\text{id}'_1, \text{id}'_2, \dots \text{id}'_t), (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_s), (x_1, x_2, \dots, x_t), P_i)$$
from P_i and $(\text{Input}, (\text{id}_1, \text{id}_2, \dots \text{id}_s), (\text{id}'_1, \text{id}'_2, \dots \text{id}'_t), P_i)$ from all other parties, set $\text{Val}[\text{id}_j] = \mathbf{x}_j$ for $j = 1, 2, \dots, s$ and $\text{ValField}[\text{id}'_j] = x_j$ for $j = 1, 2, \dots, t$.
2. **Add:** On input $(\text{Add}, \bar{\text{id}}, \text{id}, a))$ from all parties. If a is an id store $\text{Val}[\bar{\text{id}}] = \text{Val}[\text{id}] + \text{Val}[a]$. If a is a vector in $\mathbb{F}_2^k$ store $\text{Val}[\bar{\text{id}}] = \text{Val}[\text{id}] + a$.
3. **LinComb:** On input
$$(\text{LinComb}, \bar{\text{id}}, (\text{id}_1, \text{id}_2, \dots \text{id}_s), (\text{id}'_1, \text{id}'_2, \dots \text{id}'_t), a_1, a_2, \dots, a_{s+t}, a)$$
from all parties, where a_j is in $\mathbb{F}_{2^m}$ or $\mathbb{F}_2^k$ and $t \geq 1$. Define $\tilde{a}_j$ to be a_j if $a_j \in \mathbb{F}_{2^m}$, and $\phi(a_j)$ if $a_j \in \mathbb{F}_2^k$, and store $\text{ValField}[\bar{\text{id}}] = \sum_{j=1}^{s} \tilde{a}_j \cdot \phi(\text{Val}[\text{id}_j]) + \sum_{j=1}^{t} \tilde{a}_{s+j} \cdot \text{ValField}[\text{id}'_j] + \tilde{a}$.
4. **Open:** On input $(\text{Open}, \text{Dict}, \text{id}, S)$ from all parties, where S is a non-empty subset of parties. If $\text{Dict} = \text{Val}$ and $\text{Val}[\text{id}] \neq \perp$ wait for an $\mathbf{x}$ from the adversary and send $\mathbf{x}$ to the honest parties in S. If $\text{Dict} = \text{ValField}$ and $\text{ValField}[\text{id}] \neq \perp$ wait for an x from the adversary and send x to the parties in S.
5. **Check:** On input
$$(\text{Check}, (\text{id}_1, \text{id}_2, \dots, \text{id}_s), (\text{id}'_1, \text{id}'_2, \dots, \text{id}'_t), (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_s), (x_1, x_2, \dots, x_t))$$
from every party wait for an input from the adversary. If they input OK, $\text{Val}[\text{id}_j] = \mathbf{x}_j$ for $j = 1, 2, \dots, s$ and $\text{ValField}[\text{id}'_j] = x_j$ for $j = 1, 2, \dots, t$ return OK to all parties. Otherwise abort.

Notation: We will use the notation $\langle \mathbf{x} \rangle$ to refer to a value $\mathbf{x} \in \mathbb{F}_2^k$ stored in Val, and the notation $[x]$ to refer to a value $x \in \mathbb{F}_{2^m}$ stored in ValField.

Figure 1. Functionality – Authentication

Functionality $\mathcal{F}_{\text{Prep}}$

This functionality has the same features as $\mathcal{F}_{\text{Auth}}$ along with the following commands.

1. **InputTuple:** On input $(\text{InputTuple}, \text{id}, P_i)$ from all parties let P_i choose $\mathbf{r} \in \mathbb{F}_2^k$ at random and call $\mathcal{F}_{\text{Auth}}$ with input $(\text{Input}, \text{id}, \mathbf{r}, P_i)$ to obtain $\langle \mathbf{r} \rangle$. Output $\langle \mathbf{r} \rangle$ to all parties and $\mathbf{r}$ to P_i.
2. **ReEncodeTuple:** On input $(\text{ReEncodeTuple}, \text{id}_1, \text{id}_2)$ sample a random field element $r \in \mathbb{F}_{2^m}$ and set $\text{Val}[\text{id}_1] = \psi(r)$ and $\text{ValField}[\text{id}_2] = r$.
3. **Triple:** On input $(\text{Triple}, \text{id}_a, \text{id}_b, \text{id}_c)$ from all parties, sample two random vectors $\mathbf{a}, \mathbf{b} \in \mathbb{F}_2^k$ and set $(\text{Val}[\text{id}_a], \text{Val}[\text{id}_b], \text{ValField}[\text{id}_c]) = (\mathbf{a}, \mathbf{b}, \phi(\mathbf{a}) \cdot \phi(\mathbf{b}))$.

Figure 2. Functionality – Preprocessing

As explained before, at every wire of the circuit parties hold authenticated sharings $\langle \mathbf{x} \rangle$ of the vectors $\mathbf{x}$ whose coordinates would be the bits in that wire in each of the k instances of the boolean computation. To have a party P_i input a value $\langle \mathbf{x} \rangle$ we use a preprocessed authenticated sharing of a random element $\langle \mathbf{r} \rangle$ known by P_i, who can then broadcast $\mathbf{x} - \mathbf{r}$ to allow the parties to obtain $\langle \mathbf{x} \rangle$ from $\langle \mathbf{r} \rangle$. Note that this way of proceeding ensures that the shared MAC of $\langle \mathbf{x} \rangle$ is well constructed as long as $\langle \mathbf{r} \rangle$ is.

Additions $\langle \mathbf{x} \rangle + \langle \mathbf{y} \rangle = \langle \mathbf{x} + \mathbf{y} \rangle$ can be performed locally due to linearity of the secret sharing and MAC, as it is usual for this type of protocols.

The two-step process to handle multiplications $\langle \mathbf{x} \rangle * \langle \mathbf{y} \rangle$ is described next. For a preprocessed multiplication triple $(\langle \mathbf{a} \rangle, \langle \mathbf{b} \rangle, [c])$, using linearity parties can locally compute and partially open $\langle \boldsymbol{\epsilon} \rangle = \langle \mathbf{x} \rangle - \langle \mathbf{a} \rangle$ and $\langle \boldsymbol{\delta} \rangle = \langle \mathbf{y} \rangle - \langle \mathbf{b} \rangle$ publicly. Once again linearity allows parties to compute

$$[c] + \boldsymbol{\epsilon} * \langle \mathbf{y} \rangle + \boldsymbol{\delta} * \langle \mathbf{x} \rangle - \phi(\boldsymbol{\epsilon}) \cdot \phi(\boldsymbol{\delta}) = [\phi(\mathbf{x}) \cdot \phi(\mathbf{y})],$$

assuming all operations have been done honestly.

Now, parties take a tuple $(\langle \psi(r) \rangle, [r])$ and partially open $[\sigma] = [\phi(\mathbf{x}) \cdot \phi(\mathbf{y})] - [r]$. To complete the conversion back to $\langle \cdot \rangle$ they compute $\psi(\sigma) + \langle \psi(r) \rangle = \langle \mathbf{x} * \mathbf{y} \rangle$.

From this description we present the functionality for our MPC protocol in Figure 3 and the protocol implementing the online phase in Figure 4.

Theorem 2. *Π_{Online} securely implements $\mathcal{F}_{\text{MPC}}$ in the $\mathcal{F}_{\text{Prep}}$-hybrid model.*

Proof. The correctness follows from the explanation above. We argue security in Appendix C but we note that the online phase from this protocol is similar to the online phases of protocols such as [13,14,19,15], except that in every multiplication we additionally need to use the tuple $(\langle \psi(r) \rangle, [r])$ in order to transform $[\phi(\mathbf{x}) \cdot \phi(\mathbf{y})]$ into $\langle \mathbf{x} * \mathbf{y} \rangle$. Since r is uniformly random in the field $\mathbb{F}_{2^m}$, the opened value σ masks any information on $\mathbf{x}$, $\mathbf{y}$.

4 Comparison with online phases of MiniMAC and Committed MPC

In this section we compare the communication complexity of our online phase with that of MiniMAC [14] and Committed MPC [15], two secret-sharing based MPC protocols which

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

1. **Initialize:** On input Init from all players setup an empty dictionary Val.
2. **Input:** On input $(\text{Input}, \text{id}, \mathbf{x}, P_i)$ from P_i and $(\text{Input}, \text{id}, P_i)$ from all other parties where $\mathbf{x} \in \mathbb{F}_2^k$ and $\text{Val}[\text{id}] = \perp$ set $\text{Val}[\text{id}] = \mathbf{x}$.
3. **Add:** On input $(\text{Add}, \text{id}_1, \text{id}_2, \text{id}_3)$ from all parties where $\text{Val}[\text{id}_1] \neq \perp$ and $\text{Val}[\text{id}_2] \neq \perp$, set $\text{Val}[\text{id}_3] = \text{Val}[\text{id}_1] + \text{Val}[\text{id}_2]$.
4. **Multiply:** On input $(\text{Mult}, \text{id}_1, \text{id}_2, \text{id}_3)$ from all parties where $\text{Val}[\text{id}_1] \neq \perp$ and $\text{Val}[\text{id}_2] \neq \perp$, set $\text{Val}[\text{id}_3] = \text{Val}[\text{id}_1] * \text{Val}[\text{id}_2]$.
5. **Output:** On input $(\text{Output}, \text{id})$ from all parties when $\text{Val}[\text{id}] \neq \perp$ retrieve $z = \text{Val}[\text{id}]$ and send z to the adversary. Wait for an input from the adversary, if the adversary inputs OK send z to the honest parties. Otherwise abort.

Figure 3. Functionality – MPC

Protocol Π_{Online}

1. **Initialize:** The parties call the preprocessing functionality $\mathcal{F}_{\text{Prep}}$ to obtain input tuples $(\mathbf{r}, \langle \mathbf{r} \rangle)$ for each party, re-encode tuples $(\langle \psi(r) \rangle, [r])$, and multiplication triples $(\langle \mathbf{a} \rangle, \langle \mathbf{b} \rangle, [c])$.
2. **Input:** For an input gate belonging to P_i having input $\mathbf{x} \in \mathbb{F}_2^k$ the parties do the following
   (a) P_i takes a tuple $(\mathbf{r}, \langle \mathbf{r} \rangle)$ and broadcasts $\boldsymbol{\epsilon} = \mathbf{x} - \mathbf{r}$.
   (b) The parties compute $\langle \mathbf{x} \rangle = \boldsymbol{\epsilon} + \langle \mathbf{r} \rangle$.
3. **Add:** To compute componentwise addition of $\langle \mathbf{x} \rangle$ and $\langle \mathbf{y} \rangle$ the parties locally compute $\langle \mathbf{x}+\mathbf{y} \rangle = \langle \mathbf{x} \rangle + \langle \mathbf{y} \rangle$.
4. **Multiply:** To compute a componentwise multiplication of $\langle \mathbf{x} \rangle$ and $\langle \mathbf{y} \rangle$, take the next available multiplication triple $(\langle \mathbf{a} \rangle, \langle \mathbf{b} \rangle, [c])$ and tuple $(\langle \psi(r) \rangle, [r])$.
   (a) Set $\langle \boldsymbol{\epsilon} \rangle = \langle \mathbf{x} \rangle - \langle \mathbf{a} \rangle$ and $\langle \boldsymbol{\delta} \rangle = \langle \mathbf{y} \rangle - \langle \mathbf{b} \rangle$ and partially open $\boldsymbol{\epsilon}$ and $\boldsymbol{\delta}$.
   (b) Compute[3] $[c] + \boldsymbol{\epsilon} * \langle \mathbf{y} \rangle + \boldsymbol{\delta} * \langle \mathbf{x} \rangle - \phi(\boldsymbol{\epsilon}) \cdot \phi(\boldsymbol{\delta}) = [\phi(\mathbf{x}) \cdot \phi(\mathbf{y})]$.
   (c) Compute $[\sigma] = [\phi(\mathbf{x}) \cdot \phi(\mathbf{y})] - [r]$ and partially open this value to obtain σ.
   (d) Compute $\psi(\sigma) + \langle \psi(r) \rangle = \langle \mathbf{x} * \mathbf{y} \rangle$ and output this value.
5. **Output:** This stage is entered when the players have an unopened sharing $\langle \mathbf{z} \rangle$ which they want to output. Let $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_s$ be all opened $\langle \cdot \rangle$-sharings, i.e. $\mathbf{x}_j \in \mathbb{F}_2^k$ and let $x_1, x_2, \ldots, x_t$ be all opened $[\cdot]$-sharings, i.e. $x_j \in \mathbb{F}_{2^m}$. The parties do the following:
   (a) Call $\mathcal{F}_{\text{Auth.Check}}$ with inputs $(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_s)$ and $(x_1, x_2, \ldots, x_t)$.
   (b) If the check passes, partially open $\mathbf{z}$.
   (c) Call $\mathcal{F}_{\text{Auth.Check}}$ with input $\mathbf{z}$
   (d) If the check passes, output $\mathbf{z}$ to all parties.

Figure 4. Online phase

are well-suited for simultaneously evaluating k instances of the same boolean circuit. We will count broadcasting a message of M bits by a party as having that party communicate

[3] Throughout the protocol, operations with authenticated sharings are formally executed by calling $\mathcal{F}_{\text{Auth.LinComb}}$ and $\mathcal{F}_{\text{Auth.Add}}$ on the appropriate inputs. E.g. the step $[c]+\boldsymbol{\epsilon}*\langle \mathbf{y} \rangle+\boldsymbol{\delta}*\langle \mathbf{x} \rangle-\phi(\boldsymbol{\epsilon})\cdot\phi(\boldsymbol{\delta})$ is shorthand for calling $\mathcal{F}_{\text{Auth}}$ on input $(\text{LinComb}, \overline{\text{id}}, (\text{id}_1, \text{id}_2), (\text{id}'), \boldsymbol{\epsilon}, \boldsymbol{\delta}, c, -\phi(\boldsymbol{\epsilon}) \cdot \phi(\boldsymbol{\delta}))$, where id_1, id_2 are the indices corresponding to $\langle \mathbf{x} \rangle$ and $\langle \mathbf{y} \rangle$, id' is the index corresponding to $[c]$ and $\bar{\text{id}}$ is a still unused index in ValField. Similarly, $\langle \boldsymbol{\epsilon} \rangle = \langle \mathbf{x} \rangle - \langle \mathbf{a} \rangle$ is computed by calling $\mathcal{F}_{\text{Auth.Add}}$ on input $(\text{Add}, \bar{\text{id}}, \text{id}_1, \text{id}_2)$ where id_1 and id_2 are the indices corresponding to $\mathbf{x}$ and $\mathbf{a}$.

$M(n-1)$ bits (M bits to each other party). In an amortized sense, this can be achieved using point-to-point channels by a technique described in the full version of [13].

Communication complexity of our protocol.

Partially opening a $\langle\cdot\rangle$-authenticated secret involves $2k(n-1)$ bits of communication, since we have one selected party receive the share of each other party and broadcast the reconstructed value. Likewise, partially opening a $[\cdot]$-authenticated value communicates $2m(n-1)$ bits. In our online phase, every input gate requires $k(n-1)$ bits of communication. Multiplication gates require the partial opening of two $\langle\cdot\rangle$-authenticated values and one $[\cdot]$-authenticated value, hence $(4k+2m)(n-1)$ bits of communication. An output gate requires to do a MAC-check on (a linear combination of) previously partially opened values, then partially opening the output, and finally doing a MAC check on the output. A MAC check require every party to communicate a MAC share in $\mathbb{F}_{2^m}$, for a total of mn bits communicated. Hence output gates require $2k(n-1)+2mn$ bits of communication.

MiniMAC.

We have described in the introduction the idea of MiniMAC and for reference a more detailed description is in the Appendix B. In short, MiniMAC utilizes a linear error correcting code C with parameters $[\ell, k, d]$ (i.e., it allows for linear encoding of messages from $\mathbb{F}_2^k$ into $\mathbb{F}_2^\ell$ with minimum Hamming distance d). It is also important to consider another code, C^* defined as $C^* = \text{span}\{\mathbf{x} * \mathbf{y} \mid \mathbf{x}, \mathbf{y} \in C\}$, the smallest linear code containing the coordinatewise product of every pair of codewords in C^*. This is another code with parameters $[\ell, k^*, d^*]$. We always have $d \geq d^*$, and the cheating success probability of the adversary in the MiniMAC protocol is 2^{-d^*}, so we need $d^* \geq s$ for the statistical parameter s. In MiniMAC, parties have additive shares of encodings $C(\mathbf{x})$, where the shares are also codewords, and shares of the MAC $\boldsymbol{\alpha} * C(\mathbf{x})$, which can be arbitrary vectors in $\mathbb{F}_2^\ell$. In addition, at multiplication gates C^*-encodings of information are temporarily created. The online phase of MiniMAC has a very similar communication pattern to our protocol: in particular, a multiplication requires to partially open two elements encoded with C (coming from the use of Beaver's technique) and one element encoded with C^*. Since shares of C-(resp C^*-)encodings are codewords in C (resp C^*), and describing such codewords require k bits (resp. k^* bits)[4] the total communication complexity is $(4k+2k^*)(n-1)$, so the difference with our protocol depends on the difference between the achievable parameters for their k^* and our m, which we compare later. Input gates require $k(n-1)$ bits, as in our case, and for output gates, since MAC shares are arbitrary vectors in $\mathbb{F}_2^\ell$, a total of $2k(n-1)+2\ell n$ bits are sent.

Committed MPC.

Committed MPC [15] is a secret-sharing based MPC protocol that relies on UC-secure additively homomorphic commitments for authentication, rather than on MACs. Efficient commitments of this type have been proposed in works such as [16,9,8], based on OT (in the first two cases) and extractable commitments (in the third), primitives that are only used in the preprocessing. The main ingredient of the construction is again a linear error correcting

[4] We observe that this is more lenient than the description of MiniMAC in [14,12] where it is implied that parties need to send vectors of ℓ bits in order to do these openings

code C with parameters $[\ell, k, d]$. In committed MPC, for every data vector $\mathbf{x} \in \mathbb{F}_2^k$, each party P_i holds an additive share $\mathbf{x}_i \in \mathbb{F}_2^k$ to which she commits towards every other party P_j (in the multi-receiver commitment from [8], this can be accomplished by only one commitment). During most of the online phase there are only partial openings of values (where only the shares are revealed) and only at output gates the commitments are checked. Multiplication is done through Beaver's technique. In this case only two values $\boldsymbol{\epsilon}$, $\boldsymbol{\delta}$ need to be partially opened. In exchange, parties need to communicate in order to compute commitments to $\boldsymbol{\delta} * \mathbf{a}$ (resp. $\boldsymbol{\epsilon} * \mathbf{b}$) given $\boldsymbol{\epsilon}$, and commitments to $\mathbf{a}$ (resp. $\boldsymbol{\epsilon}$ and commitments to $\mathbf{b}$) at least with current constructions for UC-secure homomorphic commitments. [15, full version, fig. 16] provides a protocol where each of these products with known constant vectors requires to communicate one full vector of length ℓ and two vectors of k^* components (again ℓ is the length of C and k^* is the dimension of C^*. In total the communication complexity of a multiplication is $(4k+2k^*+\ell)(n-1)$ bits. Output gates require to open all the commitments to the shares of the output. Since opening a commitments in [16,9,8] requires to send two vectors of length ℓ to every other party, which has a total complexity of $2\ell(n-1)n$. Input gates have the same cost as for the other two protocols.

Concrete parameters Summing up we have the communication costs given in Table 1.

	MiniMAC	Committed MPC	Our protocol
Input	$k(n-1)$	$k(n-1)$	$k(n-1)$
Add	0	0	0
Multiply	$(4k+2k^*)(n-1)$	$(4k+2k^*+\ell)(n-1)$	$(4k+2m)(n-1)$
Output	$2\cdot\ell\cdot n + 2k(n-1)$	$2\cdot\ell\cdot(n-1)n$	$2\cdot m\cdot n + 2k(n-1)$

Table 1. Bits sent in the different gates in the online phases, when computing k instances of a boolean circuit in parallel.

We see that the comparison between the different protocols depends on the relation between m/k (in our case) and k^*/k and ℓ/k in the other two protocols. While the possible parameters ℓ, k, d of linear codes have been studied exhaustively in the theory of error-correcting codes, relations between those parameters and k^*, d^* are much less studied, at least in the case of binary codes. As far as we know, the only concrete non-asymptotic results are given in [6,10]. In particular, the parameters in Table 2 are achievable.

ℓ	k	$d \geq$	k^*	$d^* \geq$	k^*/k	ℓ/k
511	31	219	232	73	7.48	16.48
1023	46	439	441	147	9.59	22.24
2047	210	463	1695	67	8.07	9.75
4095	338	927	3293	135	9.74	12.11

Table 2. Parameters for C and C^{*2} from [6].

k	m	m/k
21	65	3.10
24	75	3.13
42	135	3.21
45	145	3.22

Table 3. Parameters for RMFE from [7].

On the other hand, the parameters for our protocol depend on parameters achievable by RMFEs. By Theorem 1 for all $1 \leq r \leq 33$, there exists a RMFE with $k = 3r$ and $m = 10r - 5$. Some specific values are shown in Table 3.

For security parameter $s = 64$, we need to select a RMFE where $m \geq 64$ in the case of our protocol (we choose the first entry), and a code for which $d^* \geq 64$, in the case of the other two protocols. The best selection for optimizing multiplication gates is the first code for MiniMAC and the third one for Committed MPC. The communication complexity *per computed instance of the boolean circuit* is then given in Table 4.

	MiniMAC	Committed MPC	Our protocol
Input	$(n-1)$	$(n-1)$	$(n-1)$
Add	0	0	0
Multiply	$20.96 \cdot (n-1)$	$29.89 \cdot (n-1)$	$10.2 \cdot (n-1)$
Output	$32.96n + 2(n-1)$	$19.5 \cdot (n-1)n$	$6.2 \cdot n + 2(n-1)$

Table 4. number of bits per instance sent in the different gates in the online phases, for $s = 64$.

We can see that the communication complexity of computing multiplication gates in our protocol is less than half of that of MiniMAC and Committed MPC. Similar numbers are obtained for security parameter $s = 128$.

Comparison with an online communication-efficient version of MiniMAC In [12], a version of MiniMAC is proposed which uses linear codes over the extension field $\mathbb{F}_{256}$. The larger field enables to use a Reed-Solomon code, for which $k^* = 2k - 1$. However, because this only gives coordinatewise operations in $\mathbb{F}_{256}^k$, the protocol needs to be modified in order to allow for bitwise operations instead. The modified version requires the opening of two C^*-encodings at every multiplication gate and a more complicated and much larger preprocessing, where in addition to creating certain type of multiplication triple, the preprocessing phase needs to provide authenticated sharings of 16 other vectors created from the bit decompositions of the coordinates of the two "factor" vectors in the triple[5]. As far as we know, no preprocessing phase that creates these authenticated elements has been proposed.

The amortized communication complexity of that protocol is of $8(n-1)$ bits per multiplication gate, per instance of the circuit, which is slightly less than 80% of ours. On the other hand, we estimate that the complexity of the preprocessing would be at least 4 times as that of our protocol and possibly larger, based on the number of preprocessed elements and their correlation. It is an interesting question whether we can adapt our protocol to enjoy a similar trade-off between communication complexity and amount of preprocessing.

5 Preprocessing phase

In the preprocessing phase we need to generate input tuples $(\mathbf{r}, \langle \mathbf{r} \rangle)$, re-encoding pairs (of the form $(\langle \psi(r) \rangle, [r])$), and multiplication triples $(\langle \mathbf{a} \rangle, \langle \mathbf{b} \rangle, [c])$ where $c = \phi(\mathbf{a}) \cdot \phi(\mathbf{b})$. We

[5] Namely if $\mathbf{a} = (a_1, ..., a_k) \in \mathbb{F}_{256}^k$ is one of these vectors the preprocessing needs to create authentications for all $\mathbf{a}^j = (a_1^j, ...a_k^j) \in \mathbb{F}_{256}^k$, $j = 1, ..., 7$, where if $\{1, Y, Y^2, \ldots, Y^7\}$ is a basis of $\mathbb{F}_{256}$ over $\mathbb{F}_2$ and $a_i = \sum_{j=0}^{7} a_{i,j} Y^j$ then a_i^j is defined as $a_{i,j} Y^j$.

use the ideas from [19] and [15] to create this preprocessing data from oblivious transfer. Small modifications of these protocols suffice for this, the underlying reason being that the $\mathbb{F}_2$-linearity of ϕ and ψ fit well with the OT-extension based techniques in these papers.

We use the following basic ideal functionalities. In first place parties can generate uniform random elements in a finite set, using the functionality $\mathcal{F}_{\text{Rand}}$ (for the sake of notational simplicity we will omit referring to $\mathcal{F}_{\text{Rand}}$ in protocols). Moreover, parties have access to a commitment functionality $\mathcal{F}_{\text{Comm}}$.

Functionality $\mathcal{F}_{\text{Rand}}$

1. Upon receiving (Rand, S) from all parties, where S is a finite set, choose a uniform random number $r \in S$ and send it to all parties.

Functionality $\mathcal{F}_{\text{Comm}}$

1. Upon receiving (Comm, x, P_i) from P_i and (Comm, P_i) from all other parties the functionality stores x. When receiving an opening command from all parties, the functionality sends x to all parties.

Figure 5. Functionalities – Randomness generation and Commitment

We will also make use of a random 1-out-of-2 oblivious transfers functionality outputting k-bit strings. We use the notation $\mathcal{F}_{\text{ROT}}^{n,k}$ to denote n instances of this functionality which is presented in Figure 6.

Functionality $\mathcal{F}_{\text{ROT}}^{n,k}$

1. Upon receiving (ROT, P_i, P_j) from party P_i and $(\text{ROT}, P_i, P_j, \mathbf{b})$ from party P_j, where $\mathbf{b} \in \{0,1\}^n$, the functionality chooses $\mathbf{r}_0^l, \mathbf{r}_1^l \in \{0,1\}^k$ uniformly at random and sends these to P_i, while it sends $\mathbf{r}_{b_l}^l$ to P_j for $l = 1, 2, \ldots, n$.

Figure 6. Functionality – Random OT

We adapt the correlated oblivious product evaluation functionality $\mathcal{F}_{\text{COPEe}}$ defined in MASCOT [19]. We recall how this functionality works: we see the field $\mathbb{F}_{2^m}$ as $\mathbb{F}_2[X]/(f)$ for some irreducible polynomial $f \in \mathbb{F}_2$. Then as a vector space over $\mathbb{F}_2$ the set $\{1, X, X^2, \ldots, X^{m-1}\}$ constitutes a basis for $\mathbb{F}_{2^m}$. The functionality as described in [19] takes an input $\alpha \in \mathbb{F}_{2^m}$ from one of the parties P_B in the initialization phase; then there is an arbitrary number of extend phases where on input $x \in \mathbb{F}_{2^m}$ from P_A, the functionality creates additive sharings of $\alpha \cdot x$ for the two parties. However, if P_A is corrupted it may instead decide to input a vector of elements $(x_0, x_1, \ldots, x_{m-1}) \in (\mathbb{F}_{2^m})^m$, and in that case the functionality outputs a sharing of $\sum_{i=0}^{m-1} x_i \cdot \alpha_i \cdot X^i$ (where α_i are the coordinates of α in the above basis). The honest case would correspond to all x_i being equal to x. This functionality from MASCOT corresponds to the steps Initialize and ExtendField in our version Figure 8.

We add the step ExtendVector, where party P_A can input a vector $\mathbf{x} \in \mathbb{F}_2^k$ and the functionality outputs an additive sharing of $\alpha \cdot \phi(\mathbf{x}) \in \mathbb{F}_{2^m}$. If party P_A is corrupted it may instead input $(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{m-1}) \in (\mathbb{F}_2^k)^m$. In that case the functionality outputs an additive sharing of $\sum_{i=0}^{m-1} \phi(\mathbf{x}_i) \cdot \alpha_i \cdot X^i$, and note that this is more restrictive for the corrupted adversary than ExtendField since the values $\phi(\mathbf{x}_i)$ are not free in $\mathbb{F}_{2^m}$ but confined to the image of ϕ.

We define the functionality $\mathcal{F}_{\text{COPEe}}$ in Figure 7 and present a protocol implementing the functionality in Figure 8.

Functionality $\mathcal{F}_{\text{COPEe}}$

This functionality runs with two parties P_A and P_B and an adversary $\mathcal{A}$. The Initialize phase is run once first. The ExtendVector and ExtendField may be run an arbitrary number of times, in arbitrary order.

1. **Initialize:** On input $\alpha \in \mathbb{F}_{2^m}$ from P_B the functionality stores this value. We identify α by the vector $(\alpha_0, \alpha_1, \ldots, \alpha_{m-1}) \in \mathbb{F}_2^m$, s.t. $\alpha = \sum_{i=0}^{m-1} \alpha_i \cdot X^i$.
2. **ExtendVector:** P_A inputs a vector $\mathbf{x} \in \mathbb{F}_2^k$.
   (a) If P_A is corrupt receive $t \in \mathbb{F}_{2^m}$ and $(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{m-1}) \in (\mathbb{F}_2^k)^m$ from $\mathcal{A}$, where the numbers indicate that $\mathbf{x}_i$ might be different from $\mathbf{x}$. Then compute q such that
   $$q + t = \sum_{i=0}^{m-1} \phi(\mathbf{x}_i) \cdot \alpha_i \cdot X^i$$
   (b) If both parties are honest sample $t \in \mathbb{F}_{2^m}$ at random and compute q such that
   $$q + t = \alpha \cdot \phi(\mathbf{x})$$
   (c) If only P_B is corrupt then receive $q \in \mathbb{F}_{2^m}$ from $\mathcal{A}$ and compute t s.t.
   $$q + t = \alpha \cdot \phi(\mathbf{x})$$
   (d) Output t to P_A and q to P_B.
3. **ExtendField:** P_A inputs a field element $x \in \mathbb{F}_{2^m}$.
   (a) If P_A is corrupt receive $t \in \mathbb{F}_{2^m}$ and $(x_0, x_1, \ldots, x_{m-1}) \in (\mathbb{F}_{2^m})^m$ from $\mathcal{A}$, where the numbers indicate that x_i might be different from x. Then compute q such that
   $$q + t = \sum_{i=0}^{m-1} x_i \cdot \alpha_i \cdot X^i$$
   (b) If both parties are honest sample $t \in \mathbb{F}_{2^m}$ at random and compute q such that
   $$q + t = \alpha \cdot x$$
   (c) If only P_B is corrupt then receive $q \in \mathbb{F}_{2^m}$ from $\mathcal{A}$ and compute t s.t.
   $$q + t = \alpha \cdot x$$
   (d) Output t to P_A and q to P_B.

Figure 7. Functionality – Correlated oblivious product evaluation with errors.

Protocol Π_{COPEe}

The protocol is a two party protocol with parties P_A and P_B that uses PRFs $F\colon \{0,1\}^\lambda \times \{0,1\}^\lambda \to \mathbb{F}_2^k$ and $F_{\text{Field}}\colon \{0,1\}^\lambda \times \{0,1\}^\lambda \to \mathbb{F}_{2^m}$, has access to the ideal functionality $\mathcal{F}_{\text{ROT}}^{m,\lambda}$, and maintains a global counter $j := 0$. The Initialize phase is run once first, and then the ExtendVector and ExtendField may be run an arbitrary number of times, in arbitrary order.

1. **Initialize:** On input $\alpha \in \mathbb{F}_{2^m}$ from P_B:
   (a) The parties engage in $\mathcal{F}_{\text{ROT}}^{m,\lambda}$ where P_B inputs $(\alpha_0, \alpha_1, \ldots, \alpha_{m-1}) \in \mathbb{F}_2^m$ s.t. $\alpha = \sum_{i=0}^{m-1} \alpha_i \cdot X^i \in \mathbb{F}_{2^m}$. P_A receives $\left\{(\mathbf{k}_0^i, \mathbf{k}_1^i)\right\}_{i=0}^{m-1}$ and P_B receives $\mathbf{k}_{\alpha_i}^i$ for $i = 0, 1, \ldots, m-1$.
2. **ExtendVector:** On input $\mathbf{x} \in \mathbb{F}_2^k$ from P_A:
   (a) For $i = 0, 1, \ldots, m-1$:
      i. Define
      $$\mathbf{t}_0^i = F(\mathbf{k}_0^i, j) \in \mathbb{F}_2^k, \quad \mathbf{t}_1^i = F(\mathbf{k}_1^i, j) \in \mathbb{F}_2^k,$$
      so P_A knows $(\mathbf{t}_0^i, \mathbf{t}_1^i)$ and P_B knows $\mathbf{t}_{\alpha_i}^i$.
      ii. P_A sends $\mathbf{u}_i = \mathbf{t}_0^i - \mathbf{t}_1^i + \mathbf{x}$ to P_B.
      iii. P_B computes
      $$\begin{aligned}\mathbf{q}_i &= \alpha_i \cdot \mathbf{u}_i + \mathbf{t}_{\alpha_i}^i \\ &= \mathbf{t}_0^i + \alpha_i \cdot \mathbf{x} \in \mathbb{F}_2^k.\end{aligned}$$
   (b) $j := j + 1$
   (c) P_B outputs $q = \sum_{i=0}^{m-1} \phi(\mathbf{q}_i) \cdot X^i$ and P_A outputs $t = -\sum_{i=0}^{m-1} \phi(\mathbf{t}_0^i) \cdot X^i$
   It holds that $t + q = \alpha \cdot \phi(\mathbf{x})$
3. **ExtendField:** On input $x \in \mathbb{F}_{2^m}$ from P_A:
   (a) For $i = 0, 1, \ldots, m-1$:
      i. Define
      $$t_0^i = F_{\text{Field}}(\mathbf{k}_0^i, j) \in \mathbb{F}_{2^m}, \quad t_1^i = F_{\text{Field}}(\mathbf{k}_1^i, j) \in \mathbb{F}_{2^m},$$
      so P_A knows (t_0^i, t_1^i) and P_B knows $t_{\alpha_i}^i$.
      ii. P_A sends $u_i = t_0^i - t_1^i + x$ to P_B.
      iii. P_B computes
      $$\begin{aligned}q_i &= \alpha_i \cdot u_i + t_{\alpha_i}^i \\ &= t_0^i + \alpha_i \cdot x \in \mathbb{F}_{2^m}.\end{aligned}$$
   (b) $j := j + 1$
   (c) P_B outputs $q = \sum_{i=0}^{m-1} q_i \cdot X^i$ and P_A outputs $t = -\sum_{i=0}^{m-1} t_0^i \cdot X^i$
   It holds that $t + q = \alpha \cdot x$

Figure 8. Correlated oblivious product evaluation with errors.

Proposition 1. *Π_{COPEe} securely implements $\mathcal{F}_{\text{COPEe}}$ in the $\mathcal{F}_{OT}^{m,\lambda}$-hybrid model.*

Proof. The commands Initialize and ExtendField are as in [19] (the latter being called Extend there). The proof for our ExtendVector command is analogous to the one for the ExtendField. For completeness we give more details in Appendix C.

5.1 Authentication

In the next protocol, Π_{Auth} in Figure 9, 10, and 11, we use $\mathcal{F}_{\text{COPEe}}$ to implement $\mathcal{F}_{\text{Auth}}$ from Figure 1.

Protocol Π_{Auth} – Part 1

This protocol additively shares and authenticates elements in $\mathbb{F}_2^k$ or $\mathbb{F}_{2^m}$, and allows linear operations and openings to be carried out on these shares. Note that the **Initialize** procedure only needs to be called once, to set up the MAC key. We assume access to the ideal functionalities $\mathcal{F}_{\text{Rand}}$, $\mathcal{F}_{\text{Comm}}$, and $\mathcal{F}_{\text{COPEe}}$.

1. **Initialize:** Each party P_i samples a MAC key share $\alpha^{(i)} \in \mathbb{F}_{2^m}$. Each pair of parties (P_i, P_j) for $i \neq j$ calls $\mathcal{F}_{\text{COPEe}}$.Initialize where P_i inputs $\alpha^{(i)}$.
2. **Input:** On input $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_s \in \mathbb{F}_2^k$ and $x_1, x_2, \ldots, x_t \in \mathbb{F}_{2^m}$ from P_j the parties do the following:
   (a) P_j samples random element $x_{t+1} \in \mathbb{F}_{2^m}$.
   (b) For $h = 1, 2, \ldots, s$, P_j generates additive sharing $\sum_{i=1}^n \mathbf{x}_h^{(i)} = \mathbf{x}_h$ and sends $\mathbf{x}_h^{(i)}$ to P_i. Similarly, for $l = 1, 2, \ldots, t+1$, P_j generates additive sharing $\sum_{i=1}^n x_l^{(i)} = x_l$ and sends $x_l^{(i)}$ to P_i.
   (c) For every $i \neq j$, P_i and P_j call $\mathcal{F}_{\text{COPEe.ExtendVector}}$ s times where P_j inputs $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_s$ and $\mathcal{F}_{\text{COPEe.ExtendField}}$ $t+1$ times with inputs $x_1, x_2, \ldots, x_{t+1}$.
   (d) P_i receives $q_h^{(i,j)} \in \mathbb{F}_{2^m}$ and P_j receives $t_h^{(j,i)} \in \mathbb{F}_{2^m}$ such that

$$q_h^{(i,j)} + t_h^{(j,i)} = \alpha^{(i)} \cdot \phi(\mathbf{x}_h), \qquad \text{for } h = 1, 2, \ldots, s$$
$$q_{l+s}^{(i,j)} + t_{l+s}^{(j,i)} = \alpha^{(i)} \cdot x_l, \qquad \text{for } l = 1, 2, \ldots, t+1$$

   (e) Each P_i, $i \neq j$ defines the MAC shares $m^{(i)}(\mathbf{x}_h) = q_h^{(i,j)}$ for $h = 1, 2, \ldots, s$ and $m^{(i)}(x_l) = q_{l+s}^{(i,j)}$ for $l = 1, 2, \ldots, t+1$. P_j computes MAC share

$$m^{(j)}(\mathbf{x}_h) = \alpha^{(j)} \cdot \phi(\mathbf{x}_h) + \sum_{i \neq j} t_h^{(j,i)} \qquad \text{for } h = 1, 2, \ldots, s$$
$$m^{(j)}(x_l) = \alpha^{(j)} \cdot x_l + \sum_{i \neq j} t_{l+s}^{(j,i)} \qquad \text{for } l = 1, 2, \ldots, t+1$$

   This implies that we have $\langle \mathbf{x}_h \rangle$ *for* $h = 1, 2, \ldots, s$ *and* $[x_l]$ *for* $l = 1, 2, \ldots, t+1$
   (f) The parties call $\mathcal{F}_{\text{Rand}}(\mathbb{F}_{2^m}^{s+t+1})$ to obtain $(r_1, \ldots, r_{s+t+1})$.
   (g) Compute $[y] = \sum_{h=1}^s r_h \cdot \langle \mathbf{x}_h \rangle + \sum_{l=1}^{t+1} r_{s+l} \cdot [x_l]$ by calling $\Pi_{\text{Auth.LinComb}}$ and open y by calling $\Pi_{\text{Auth.Open}}$.
   (h) Call $\Pi_{\text{Auth.Check}}$ on y. If the check succeeds output $\langle \mathbf{x}_h \rangle$ for $h = 1, 2, \ldots, s$, and $[x_l]$ for $l = 1, 2, \ldots, t$.

Figure 9. Authenticated shares – Part 1.

In the initialize phase each pair of parties (P_i, P_j) call the initialize phase from $\mathcal{F}_{\text{COPEe}}$ where P_i inputs a MAC key. Afterwards P_j can create authenticated sharings to the desired values, both of boolean vectors and of elements in the larger field: namely P_j constructs additive random sharings of the individual values and uses the appropriate extend phase

Protocol Π_{Auth} – Part 2

3. **Add:** On input $(\text{Add}, \bar{\text{id}}, \text{id}, a)$ the parties do the following. If a is an index of Val they retrieve shares and MAC shares $\mathbf{x}^{(i)}, \mathbf{y}^{(i)}, m^{(i)}(\mathbf{x}), m^{(i)}(\mathbf{y})$ where $\mathbf{x}$ corresponds to id and $\mathbf{y}$ corresponds to the index a in Val. P_i computes

$$\mathbf{x}^{(i)} + \mathbf{y}^{(i)} \quad \text{and} \quad m^{(i)}(\mathbf{x}) + m^{(i)}(\mathbf{y})$$

and store these under $\bar{\text{id}}$. If a is a vector, i.e. $a = \mathbf{a}$, they retrieve the share and MAC share $\mathbf{x}^{(i)}, m^{(i)}(\mathbf{x})$ where $\mathbf{x}$ corresponds to id in Val. P_i computes

$$\mathbf{x}^{(i)} + \begin{cases} \mathbf{a} & \text{if } i = 1 \\ 0 & \text{if } i \neq 1 \end{cases} \quad \text{and} \quad m^{(i)}(\mathbf{x}) + \alpha^{(i)} \cdot \phi(\mathbf{a}).$$

and store these under $\text{Val}[\bar{\text{id}}]$.

4. **LinComb:** On input

$$(\text{LinComb}, \bar{\text{id}}, (\text{id}_1, \text{id}_2, \dots \text{id}_s), (\text{id}'_1, \text{id}'_2, \dots \text{id}'_t), c_1, c_2, \dots, c_{s+t}, c)$$

where $t \geq 1$, the P_i retrieves its shares and MAC shares $\left\{\mathbf{x}_j^{(i)}, m^{(i)}(\mathbf{x}_j)\right\}_{j=1,2,\dots,s}$ corresponding to id_j in Val and $\left\{x_j^{(i)}, m^{(i)}(x_j)\right\}_{j=1,2,\dots,t}$ corresponding to id'_j in ValField. P_i computes

$$y^{(i)} = \sum_{j=1}^{s} c_j \cdot \phi(\mathbf{x}_j^{(i)}) + \sum_{j=1}^{t} c_{s+j} \cdot x_j^{(i)} + \begin{cases} c & \text{if } i = 1 \\ 0 & \text{if } i \neq 1 \end{cases}$$

$$m^{(i)}(y) = \sum_{j=1}^{s} c_j \cdot m^{(i)}(\mathbf{x}_j) + \sum_{j=1}^{t} c_{s+j} \cdot m^{(i)}(x_j) + c \cdot \alpha^{(i)}$$

and stores these under $\bar{\text{id}}$ in ValField.

Figure 10. Authenticated shares – Part 2.

of $\mathcal{F}_{\text{COPEe}}$ to obtain additive sharings of the MACs. At last, a random linear combination of the values chosen by P_j is checked. Here privacy is achieved by letting P_j include a dummy input x_{t+1} to mask the other inputs. Note that we have already described how to compute linear combinations involving both authenticated sharings of boolean vectors and of elements in $\mathbb{F}_{2^m}$.

Proposition 2. *Π_{Auth} securely implements $\mathcal{F}_{\text{Auth}}$ in the $(\mathcal{F}_{\text{COPEe}}, \mathcal{F}_{\text{Rand}}, \mathcal{F}_{\text{Comm}})$-hybrid model*

Proof. Since the proof is similar to the proof of security for $\Pi_{[[.]]}$ in [19], we point out the differences and argue why it does not have an impact on the security in Appendix C.

5.2 Input tuples and reencoding pairs

The two functionalities $\mathcal{F}_{\text{COPEe}}$ and $\mathcal{F}_{\text{Auth}}$ are the building blocks for the preprocessing. They are in shape very similar to the MASCOT functionalities but with some few corrections to include that sharings can be of vectors instead of field elements in $\mathbb{F}_{2^m}$. With these building

Protocol Π_{Auth} – Part 3

5. **Open:** On input $(\text{Open}, \text{Dict}, \text{id}, S)$ party P_i retrieves the share corresponding to the dictionary and index, sends the share to P_j (the party with lowest index in S) who sums the shares and sends the sum back to the other parties in S.
6. **Check:**
   (a) On input

$$(\text{Check}, (\text{id}_1, \text{id}_2, \ldots, \text{id}_s), (\text{id}'_1, \text{id}'_2, \ldots, \text{id}'_t), (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_s), (x_1, x_2, \ldots, x_t))$$

   parties sample a random vector $(r_1, r_2, \ldots, r_{s+t}) \in \mathbb{F}_{2^m}^{s+t}$. P_i retrieves its MAC shares $m^{(i)}(\mathbf{x}_j)$ for $j = 1, 2, \ldots, s$ corresponding to id_j in Val and $m^{(i)}(x_j)$ for $j = 1, 2, \ldots, t$ corresponding to id'_j in ValField. Define

$$y = \sum_{j=1}^{s} r_j \cdot \phi(\mathbf{x}_j) + \sum_{j=1}^{t} r_{s+j} \cdot x_j$$

   and let P_i compute

$$m^{(i)}(y) = \sum_{j=1}^{s} r_j \cdot m^{(i)}(\mathbf{x}_j) + \sum_{j=1}^{t} r_{s+j} \cdot m^{(i)}(x_j)$$

   (b) P_i calls $\mathcal{F}_{\text{Comm}}$ to commit to $\sigma^{(i)} = m^{(i)}(y) - \alpha^{(i)} \cdot y$ and afterwards open the commitment.
   (c) The parties check if $\sigma^{(1)} + \sigma^{(2)} + \cdots + \sigma^{(n)} = 0$ and abort otherwise.

Figure 11. Authenticated shares – Part 3.

blocks we can produce the randomness needed for the online phase. First of all, we produce input tuples with protocol $\Pi_{\text{InputTuple}}$ in Figure 12. Proposition 3 is straightforward.

Proposition 3. *$\Pi_{\text{InputTuple}}$ securely implements $\mathcal{F}_{\text{Prep.InputTuple}}$ in the $\mathcal{F}_{\text{Auth}}$-hybrid model.*

Protocol $\Pi_{\text{InputTuple}}$

The protocol generates $(\mathbf{r}, \langle \mathbf{r} \rangle)$ where $\mathbf{r} \in \mathbb{F}_2^k$ is chosen randomly by P_i, the party calling the protocol.

1. **Construct:**
   (a) P_i chooses $\mathbf{r} \in \mathbb{F}_2^k$ uniformly at random.
   (b) P_i calls $\mathcal{F}_{\text{Auth}}$ to obtain $\langle \mathbf{r} \rangle$ and output this authenticated share.

Figure 12. Creating input tuples.

We also need to construct tuples to re-encode $[\cdot]$-sharings to $\langle \cdot \rangle$-sharings after a multiplication. A protocol $\Pi_{\text{ReEncodeTuple}}$ for producing the tuples $(\langle \psi(r) \rangle, [r])$ for random $r \in \mathbb{F}_{2^m}$ are shown in Figure 13.

Proposition 4. *$\Pi_{\text{ReEncodeTuple}}$ securely implements $\mathcal{F}_{\text{Prep.ReEncodeTuple}}$ in the $(\mathcal{F}_{\text{Auth}}, \mathcal{F}_{\text{Rand}})$-hybrid model with statistical security parameter s.*

Proof. Clearly the values r_j are random. The correctness of the output is based on the fact that the sacrificing step guarantees that if after the Construct Phase, there are values $[r_j]$, $\langle \mathbf{s}_j \rangle$ with $\psi(\mathbf{s}_j) \neq r_j$ this will be detected with probability $1 - 2^{-s}$ in the sacrificing part. We prove this in Appendix C.

Protocol $\Pi_{\text{ReEncodeTuple}}$

The protocol generates $(\langle \psi(r_j) \rangle, [r_j])$ for $j = 1, 2, \ldots, t$, where r_j is random in $\mathbb{F}_{2^m}$ and unknown to all parties. We assume access to the functionality $\mathcal{F}_{\text{Rand}}$.

1. **Construct:**
   (a) P_i chooses $r_j^{(i)}$ for $j = 1, 2, \ldots, t + s$ uniformly at random in $\mathbb{F}_{2^m}$.
   (b) P_i calls $\mathcal{F}_{\text{Auth}}$ to obtain $[r_j^{(i)}]$ and $\langle \psi(r_j^{(i)}) \rangle$.
   (c) Compute $[r_j] = \sum_{i=1}^{n} [r_j^{(i)}]$ and $\langle \psi(r_j) \rangle = \sum_{i=1}^{n} \langle \psi(r_j^{(i)}) \rangle$ for $j = 1, 2, \ldots, t + s$.
2. **Sacrifice:**
   (a) Call $\mathcal{F}_{\text{Rand}}(\mathbb{F}_2^t)$ to obtain $\mathbf{a}'_i$ for $i = 1, 2, \ldots, s$ and define $\mathbf{a}_i = (\mathbf{a}'_i, \mathbf{e}_i)$ where $\mathbf{e}_i$ is the i'th canonical basis vector of length s.
   (b) Compute $[b_i] = \sum_{j=1}^{t+s} a_{ij} [r_j]$ and $\langle \mathbf{b}_i \rangle = \sum_{j=1}^{t+s} a_{ij} \langle \psi(r_j) \rangle$ and partially open b_i and $\mathbf{b}_i$.
   (c) If $\psi(b_i) \neq \mathbf{b}_i$ for some $i \in \{1, 2, \ldots, s\}$ then abort.
   (d) Call $\mathcal{F}_{\text{Auth.Check}}$ on the opened values $\mathbf{b}_i$ and b_i
3. **Output:** Output $(\langle \psi(r_j) \rangle, [r_j])$ for $j = 1, 2, \ldots, t$

Figure 13. Re-encode tuples.

We go more into details about the multiplication triples in the next section.

5.3 Multiplication triples

Our protocol Π_{Triple} for constructing triples is given in Figure 15. We note that $c = \phi(\mathbf{a}) \cdot \phi(\mathbf{b}) = \sum_{i,j} \phi(\mathbf{a}^{(i)}) \phi(\mathbf{b}^{(j)})$ and hence sharings of c can be obtained by adding sharings of the summands, where each of the summands only require two parties P_i and P_j to interact. Again, the construction step is much like the construction step from the protocol Π_{Triple} in [19]. where we have modified the protocol such that it produces triples $(\langle \mathbf{a} \rangle, \langle \mathbf{b} \rangle, [c])$ instead of $([a], [b], [c])$.

However, after authentication, we use some techniques from [15] to check correctness and avoid leakage on the produced triples. Indeed using the combine and sacrifice steps in MASCOT presents some problems in our case: for example, in the sacrificing step in MASCOT parties take two triples $([a], [b], [c])$ and $([\hat{a}], [b], [\hat{c}])$ and start by opening a random combination $s \cdot [a] - [\hat{a}]$ to some value ρ, so that they can later verify that $s \cdot [c] - [\hat{c}] - \rho \cdot [b]$ opens to 0. Since the second triple will be disregarded, and $s \cdot a - \hat{a}$ completely masks a since $\hat{a}$ is uniformly random, no information is revealed about a. In our case we would have triples $(\langle \mathbf{a} \rangle, \langle \mathbf{b} \rangle, [c])$ and $(\langle \hat{\mathbf{a}} \rangle, \langle \mathbf{b} \rangle, [\hat{c}])$ and sample a random $s \in \mathbb{F}_{2^m}$, it would not be the

Protocol $\Pi_{\text{TripleConstruct}}$

The protocol produces N multiplication triples.

1. **Construction:**
   (a) P_i samples $\mathbf{a}_l^{(i)}, \mathbf{b}_l^{(i)} \in \mathbb{F}_2^k$ for $l = 1, 2, \ldots, N$ and let $\phi(\mathbf{a}_l^{(i)}) = \sum_{h=0}^{m-1} a_{h,l}^{(i)} \cdot X^h$.
   (b) For $l = 1, 2, \ldots, N$ every ordered pair (P_i, P_j) does the following:
      i. The pair call $\mathcal{F}_{\text{ROT}}^{m,k}$ where P_i inputs $a_{h,l}^{(i)}$ for the h'th instance.
      ii. P_j receives $\mathbf{t}_{0,h,l}^{(j,i)}, \mathbf{t}_{1,h,l}^{(j,i)} \in \mathbb{F}_2^k$ and P_i receives $\mathbf{t}_{a_{h,l}^{(i)},h,l}^{(j,i)}$ for $h = 0, 1, \ldots, m-1$.
      iii. P_j sends $\mathbf{u}_{h,l}^{(j,i)} = \mathbf{t}_{0,h,l}^{(j,i)} - \mathbf{t}_{1,h,l}^{(j,i)} + \mathbf{b}_l^{(j)}$ for $h = 0, 1, \ldots, m-1$.
      iv. P_i sets $\mathbf{q}_{h,l}^{(j,i)} = \mathbf{t}_{a_{h,l}^{(i)},h,l}^{(j,i)} + a_{h,l}^{(i)} \cdot \mathbf{u}_{h,l}^{(j,i)} = \mathbf{t}_{0,h,l}^{(j,i)} + a_{h,l}^{(i)} \cdot \mathbf{b}_l^{(j)}$. Set $\mathbf{t}_{h,l}^{(j,i)} = \mathbf{t}_{0,h,l}^{(j,i)}$.
      v. P_i sets $c_{i,j,l}^{(i)} = \sum_{h=0}^{m-1} \phi(\mathbf{q}_{h,l}^{(j,i)}) \cdot X^h$ and P_j sets $c_{i,j,l}^{(j)} = -\sum_{h=0}^{m-1} \phi(\mathbf{t}_{h,l}^{(j,i)}) \cdot X^h$
      Now we have $c_{i,j,l}^{(i)} + c_{i,j,l}^{(j)} = \phi(\mathbf{a}_l^{(i)}) \cdot \phi(\mathbf{b}_l^{(j)})$
   (c) Each party P_i computes $c_l^{(i)} = \phi(\mathbf{a}_l^{(i)}) \cdot \phi(\mathbf{b}_l^{(i)}) + \sum_{j \neq i} c_{i,j,l}^{(i)} + c_{j,i,l}^{(i)}$
   Now we have $c_l = \sum_{i=1}^n c_l^{(i)} = \sum_{i=1}^n \phi(\mathbf{a}_l^{(i)}) \cdot \sum_{i=1}^n \phi(\mathbf{b}_l^{(i)}) = \phi(\mathbf{a}_l) \cdot \phi(\mathbf{b}_l)$ *for* $l = 1, 2, \ldots, N$
2. **Authenticate:**
   (a) P_i calls $\mathcal{F}_{\text{Auth}}$ to obtain $\langle \mathbf{a}_l^{(i)} \rangle$, $\langle \mathbf{b}_l^{(i)} \rangle$, and $[c_l^{(i)}]$.
   (b) Parties compute $\langle \mathbf{a}_l \rangle = \sum_{i=1}^n \langle \mathbf{a}_l^{(i)} \rangle$ and similarly to obtain $\langle \mathbf{b}_l \rangle$ and $[c_l]$.

Figure 14. Multiplication triples.

case that $\phi(\hat{\mathbf{a}})$ would act as a proper one-time pad for $s \cdot \phi(\mathbf{a})$[6]. A similar problem would arise for adapting the combine step in [19].

Therefore, we use instead a combine and sacrifice step more similar to [15]. In the protocol Π_{Triple} we start by constructing additive sharings of $N = \tau_1 + \tau_1 \cdot \tau_2^2 \cdot T$ triples. Then some of these triples are opened and it is checked that they are correct. This guarantees that most of the remaining triples are correct. The remaining triples are then organized in buckets and for each bucket all but one of the triples are sacrified in order to guarantee that the remaining triple is correct with very high probability. However, this step opens the door for a selective failure attack, where the adversary can guess some information about the remaining triples from the fact that the sacrifice step has not aborted (see Appendix D), so a final combining step is used to remove this leakage.

Proposition 5. *Π_{Triple} securely implements $\mathcal{F}_{\text{Prep.Triple}}$ in the $(\mathcal{F}_{\text{Auth}}, \mathcal{F}_{\text{ROT}}^{m,k}, \mathcal{F}_{\text{Rand}})$-hybrid model.*

Proof. The proof uses similar arguments as the one from [15]. For completeness we include the proof in Appendix D.

Proposition 6. *$\Pi_{\text{InputTuple}}$, $\Pi_{\text{ReEncodeTuple}}$, and Π_{Triple} securely implements $\mathcal{F}_{\text{Prep}}$ in the $(\mathcal{F}_{\text{Auth}}, \mathcal{F}_{\text{ROT}}^{m,k}, \mathcal{F}_{\text{Rand}})$-hybrid model.*

Proof. This follows directly from Propositions 3, 4, and 5.

[6] Sampling $\mathbf{s} \in \mathbb{F}_2^k$ instead would not solve the problem since $\mathbf{s} * \langle \mathbf{a} \rangle - \langle \hat{\mathbf{a}} \rangle$ is not a proper $[\cdot]$-sharing as described before (3)

Protocol Π_{Triple}

The protocol generates T multiplication triples $(\langle \mathbf{a} \rangle, \langle \mathbf{b} \rangle, [c])$ where $\mathbf{a}, \mathbf{b} \in \mathbb{F}_2^k$ are random vectors and $c = \phi(\mathbf{a}) \cdot \phi(\mathbf{b})$. The integers τ_1, τ_2 are bucket sizes and are for security reasons. Let $N = \tau_1 + \tau_1 \cdot \tau_2^2 \cdot T$.

1. **Construction:** Call $\Pi_{\text{TripleConstruct}}$ to produce N multiplication triples.
2. **Cut-and-choose:**
   (a) Call $\mathcal{F}_{\text{Rand}}$ to obtain $(l_1, l_2, \ldots, l_{\tau_1})$, where $l_i \neq l_j$ when $i \neq j$.
   (b) Open $\langle \mathbf{a}_{l_j} \rangle$, $\langle \mathbf{b}_{l_j} \rangle$, and $[c_{l_j}]$ for $j = 1, 2, \ldots, \tau_1$. Abort if $c_{l_j} \neq \phi(\mathbf{a}_{l_j}) \cdot \phi(\mathbf{b}_{l_j})$ for some j.
3. **Sacrifice:**
   (a) Use $\mathcal{F}_{\text{Rand}}$ to randomly divide the remaining $N - \tau_1$ triples into $\tau_2^2 \cdot T$ buckets with τ_1 triples in each.
   (b) In each bucket we denote the triples by $(\langle \mathbf{a}_l \rangle, \langle \mathbf{b}_l \rangle, [c_l])$ for $l = 1, \ldots, \tau_1$ and do the following:
      i. Compute $\langle \boldsymbol{\epsilon}_l \rangle = \langle \mathbf{a}_l \rangle - \langle \mathbf{a}_1 \rangle$ and $\langle \boldsymbol{\delta}_l \rangle = \langle \mathbf{b}_l \rangle - \langle \mathbf{b}_1 \rangle$ and open $\boldsymbol{\epsilon}_l$ and $\boldsymbol{\delta}_l$ for $l = 2, \ldots, \tau_1$
      ii. Compute $[\sigma_l] = [c_l] - [c_1] - \boldsymbol{\epsilon}_l * \langle \mathbf{b}_1 \rangle - \boldsymbol{\delta}_l * \langle \mathbf{a}_1 \rangle - \phi(\boldsymbol{\epsilon}_l) \cdot \phi(\boldsymbol{\delta}_l)$ and open σ_l for $l = 2, \ldots, \tau_2$. Abort if $\sigma_l \neq 0$. Otherwise, call $(\langle \mathbf{a}_1 \rangle, \langle \mathbf{b}_1 \rangle, [c_1])$ a correct triple.
4. **Combine:**
   (a) Combine on $\mathbf{a}$: Use $\mathcal{F}_{\text{Rand}}$ to randomly divide the remaining $\tau_2^2 \cdot T$ non-malformed triples into $\tau_2 \cdot T$ buckets with τ_2 in each. Denote the triples in each bucket by $(\langle \mathbf{a}_l \rangle, \langle \mathbf{b}_l \rangle, [c_l])$ for $l = 1, \ldots, \tau_2$. Combine the triples in each bucket in the following way:
      i. Compute $\langle \mathbf{a}' \rangle = \sum_{l=1}^{\tau_2} \langle \mathbf{a}_l \rangle$ and $\langle \mathbf{b}' \rangle = \langle \mathbf{b}_1 \rangle$
      ii. For $l = 2, 3, \ldots \tau_2$: Compute $\langle \boldsymbol{\epsilon}_l \rangle = \langle \mathbf{b}_1 \rangle - \langle \mathbf{b}_l \rangle$ and open $\boldsymbol{\epsilon}_l$
      iii. Compute $[c'] = [c_1] + \sum_{l=2}^{\tau_2} \boldsymbol{\epsilon}_l * \langle \mathbf{a}_l \rangle + [c_l] = [\phi(\mathbf{a}') \cdot \phi(\mathbf{b}')]$ and call $(\langle \mathbf{a}' \rangle, \langle \mathbf{b}' \rangle, [c'])$ a good triple.
   (b) Combine on $\mathbf{b}$: Use $\mathcal{F}_{\text{Rand}}$ to randomly divide the remaining $\tau_2 \cdot T$ non-malformed triples into T buckets with τ_2 in each. Denote the triples in each bucket by $(\langle \mathbf{a}_l \rangle, \langle \mathbf{b}_l \rangle, [c_l])$ for $l = 1, \ldots, \tau_2$. Combine the triples in each bucket in the following way:
      i. Compute $\langle \mathbf{b}' \rangle = \sum_{l=1}^{\tau_2} \langle \mathbf{b}_l \rangle$ and $\langle \mathbf{a}' \rangle = \langle \mathbf{a}_1 \rangle$
      ii. For $l = 2, 3, \ldots \tau_2$: Compute $\langle \boldsymbol{\epsilon}_l \rangle = \langle \mathbf{a}_1 \rangle - \langle \mathbf{a}_l \rangle$ and open $\boldsymbol{\epsilon}_l$
      iii. Compute $[c'] = [c_1] + \sum_{l=2}^{\tau_2} \boldsymbol{\epsilon}_l * \langle \mathbf{b}_l \rangle + [c_l] = [\phi(\mathbf{b}') \cdot \phi(\mathbf{a}')]$ and call $(\langle \mathbf{a}' \rangle, \langle \mathbf{b}' \rangle, [c'])$ a good triple.
   (c) Call $\mathcal{F}_{\text{Auth.Check}}$ on all opened values so far. If the check succeeds output the T good triples.

Figure 15. Multiplication triples.

Complexity of Preprocessing

We briefly describe the communication complexity for producing the randomness needed for the online phase. Starting by considering the construction of an input tuple the only communication we have to consider here is a single call to $\mathcal{F}_{\text{Auth}}$. The main cost of authentication is the call to Π_{COPEe} where the parties needs to send $mk(n-1)$ bits for each vector authenticated. In the case where a field element is authenticated instead they need to send $m^2(n-1)$ bits. Furthermore, the party who is authenticating needs to send the shares of the vector authenticating but this has only a cost of $k(n-1)$ bits. At last, the check is carried out but we assume that the parties authenticate several vectors/values in a batch and hence this cost is amortized away.

For the re-encoding pairs we assume that t is much larger than s. This means that in order to obtain a single pair the parties need to authenticate n field elements and n vectors. Once again we assume that the check is amortized away, so this gives a total cost of sending $(m^2 + mk)n(n-1)$ bits.

Regarding the communication in order to obtain a single multiplication triple we make $\tau_1\tau_2^2 n(n-1)$ calls to $\mathcal{F}_{\text{ROT}}^{m,k}$ and sends $\tau_1\tau_2^2 mkn(n-1)$ bits in the construction step. Afterwards, we authenticate $2\tau_1\tau_2^2 n$ vectors and $\tau_1\tau_2^2 n$ field elements. The cost in the remaining steps are not close to be as costly as this, so we ignore these.

In [15] it is suggested to use $\tau_1 = \tau_2 = 3$, and we can see in Table 3 that we can achieve $m \approx 3.1k$. Thus, using these parameters and adding the cost of producing a re-encode pair and a multiplication triple, the communication complexity for preparing a multiplication gate is 27 calls to $\mathcal{F}_{\text{ROT}}^{m,k}$ and sending

$$\begin{aligned}&(9.61 + 3.1)k^2 n(n-1) + (1+2)83.7k^2 n(n-1) + 259.47k^2 n(n-1) \text{ bits}\\ &= 523.28k^2 n(n-1) \text{ bits}.\end{aligned}$$

References

1. Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure MPC with linear communication complexity. In *Theory of Cryptography*, pages 213–230, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
2. Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In *Advances in Cryptology – EUROCRYPT 2011*, pages 169–188, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
3. Alexander R. Block, Hemanta K. Maji, and Hai H. Nguyen. Secure computation based on leaky correlations: High resilience setting. In *Advances in Cryptology – CRYPTO 2017*, pages 3–32, Cham, 2017. Springer International Publishing.
4. Alexander R. Block, Hemanta K. Maji, and Hai H. Nguyen. Secure computation with constant communication overhead using multiplication embeddings. In *Progress in Cryptology – INDOCRYPT 2018*, pages 375–398, Cham, 2018. Springer International Publishing.
5. Ran Canetti. Universally composable security: a new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145, 10 2001.
6. Ignacio Cascudo. On squares of cyclic codes. *IEEE Transactions on Information Theory*, 65(2):1034–1047, 02 2019.
7. Ignacio Cascudo, Ronald Cramer, Chaoping Xing, and Chen Yuan. Amortized complexity of information-theoretically secure MPC revisited. In *Advances in Cryptology – CRYPTO 2018*, pages 395–426, Cham, 2018. Springer International Publishing.
8. Ignacio Cascudo, Ivan Damgård, Bernardo David, Nico Döttling, Rafael Dowsley, and Irene Giacomelli. Efficient uc commitment extension with homomorphism for free (and applications). In *Advances in Cryptology – ASIACRYPT 2019*, pages 606–635, Cham, 2019. Springer International Publishing.
9. Ignacio Cascudo, Ivan Damgård, Bernardo David, Nico Döttling, and Jesper Buus Nielsen. Rate-1, linear time and additively homomorphic uc commitments. In *Advances in Cryptology – CRYPTO 2016*, pages 179–207, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.
10. Ignacio Cascudo, Jaron Skovsted Gundersen, and Diego Ruano. Squares of matrix-product codes. *Finite Fields and Their Applications*, 62:101606, 2020.
11. Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPD$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In *Advances in Cryptology – CRYPTO 2018*, pages 769–798, Cham, 2018. Springer International Publishing.
12. Ivan Damgård, Rasmus Lauritsen, and Tomas Toft. An empirical study and some improvements of the minimac protocol for secure computation. In *Security and Cryptography for Networks*, pages 398–415, Cham, 2014. Springer International Publishing.
13. Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology – CRYPTO 2012*, pages 643–662, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

14. Ivan Damgård and Sarah Zakarias. Constant-overhead secure computation of boolean circuits using preprocessing. In *Theory of Cryptography*, pages 621–641, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
15. Tore K. Frederiksen, Benny Pinkas, and Avishay Yanai. Committed MPC. In *Public-Key Cryptography – PKC 2018*, pages 587–619, Cham, 2018. Springer International Publishing.
16. Tore Kasper Frederiksen, Thomas P. Jakobsen, Jesper Buus Nielsen, and Roberto Trifiletti. On the complexity of additively homomorphic uc commitments. In *Theory of Cryptography*, pages 542–565, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.
17. Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl. A unified approach to MPC with preprocessing using OT. In *Proceedings, Part I, of the 21st International Conference on Advances in Cryptology – ASIACRYPT 2015 - Volume 9452*, page 711–735, Berlin, Heidelberg, 2015. Springer-Verlag.
18. Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *Advances in Cryptology – EUROCRYPT 2017*, pages 225–255, Cham, 2017. Springer International Publishing.
19. Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 830–842, New York, NY, USA, 2016. ACM.
20. Enrique Larraia, Emmanuela Orsini, and Nigel P. Smart. Dishonest majority multi-party computation for binary circuits. In *Advances in Cryptology – CRYPTO 2014*, pages 495–512, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
21. Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *Advances in Cryptology – CRYPTO 2012*, pages 681–700, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

A Universal Composability

We will use the universal composability model by Canetti [5]. The model follows a real world-ideal world simulation paradigm. In order to prove security of a protocol in this model, we roughly need to show that for any adversary taking control of an allowed set of parties in Π, there exists a simulator, interacting with the ideal functionality $\mathcal{F}$ and corrupting the same parties, that produces an indistinguishable view towards any "environment", as described below. The environment (which intuitively captures everything that happens outside of the protocol) provides inputs and reads the outputs of individual parties, and can communicate with the adversary/simulator during the protocol. At the end of the protocol, the environment tries to distinguish whether it has interacted with the real protocol and the real adversary or with the ideal world and simulator, based on the view it has received. Π UC-securely implements $\mathcal{F}$ if the environment cannot distinguish with non-negligible probability.

In the ideal model, parties simply relay inputs from the environment to the functionality, and outputs from the functionality to the environment. On the other hand, the order of quantifiers indicates that the simulator needs to be constructed for a specific adversary, we can have the simulator interact in a black-box way with an internal copy of the adversary and so by abuse of language we say in the proofs that the simulator is interacting with the adversary.

In addition to these two worlds, the UC-composability framework also considers a $\mathcal{G}$-hybrid model, where the real protocol π makes use of one or more calls to an ideal functionality $\mathcal{G}$. We can extend the definition above and say that π UC-securely implements $\mathcal{F}$ if for any adversary for that protocol there is a simulator in the ideal world that can produce the same view for any environment. Then the fundamental result in the UC-framework asserts that if there is another protocol ρ that UC-securely implements $\mathcal{G}$, we can replace the calls in π to the functionality $\mathcal{G}$ by calls to ρ without affecting the security.

B MiniMAC

In this section, we describe the online phase of MiniMAC.

We use the notation π_k in this section to denote the projection map onto $\mathbb{F}_2^k$. Hence, for a vector $\mathbf{x} \in \mathbb{F}_2^{k^*}$ where $k^* \geq k$ let $\pi_k(\mathbf{x})$ be the vector consisting of the first k entries of $\mathbf{x}$.

The MACs make use of an $[\ell, k, d]$ linear code C and its square C^* with parameters $[\ell, k^*, d^*]$. The code C^* is defined as

$$C^* = \text{span}\{\mathbf{x} * \mathbf{y} \mid \mathbf{x}, \mathbf{y} \in C\}$$

If $\mathbf{x} \in \mathbb{F}_2^k$ we use the notation $\langle \mathbf{x} \rangle$ to denote that $\mathbf{x}$ is secretly shared along with MACs. I.e.

$$\langle \mathbf{x} \rangle = \left((\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(m)}), (m^{(1)}(\mathbf{x}), m^{(2)}(\mathbf{x}), \ldots, m^{(n)}(\mathbf{x})) \right),$$

where P_i holds $\mathbf{x}^{(i)}$ and $m^{(i)}(\mathbf{x})$ and it holds that

$$\mathbf{x} = \sum_{i=1}^{n} \mathbf{x}^{(i)},$$

$$m(\mathbf{x}) = \sum_{i=1}^{n} m^{(i)}(\mathbf{x}) = \boldsymbol{\alpha} * \sum_{i=1}^{n} C(\mathbf{x}^{(i)}) = \boldsymbol{\alpha} * C(\mathbf{x}),$$

where $C(\mathbf{x}^{(i)})$ is the encoding of $\mathbf{x}^{(i)}$ in C.

Similarly, by $\langle \mathbf{y} \rangle^*$, for $\mathbf{y} \in \mathbb{F}_2^{k^*}$, we represent the same type of authentication but using C^* instead of C.

We now describe the online phase of the MiniMAC protocol. The protocol is also presented in Figure 16. For an input gate corresponding to an input from P_i, the party takes a preprocessed tuple $(\mathbf{r}, \langle \mathbf{r} \rangle)$ and broadcasts $\boldsymbol{\epsilon} = \mathbf{x} - \mathbf{r}$. All the parties can by local computations compute $\boldsymbol{\epsilon} + \langle \mathbf{r} \rangle = \langle \mathbf{x} \rangle$. Parties do linear operations locally according to the following rules:

$$\begin{aligned}
\langle \mathbf{x} \rangle + \langle \mathbf{y} \rangle &= \Big((\mathbf{x}^{(1)} + \mathbf{y}^{(1)}, \ldots, \mathbf{x}^{(n)} + \mathbf{y}^{(n)}), \\
&\quad (m^{(1)}(\mathbf{x}) + m^{(1)}(\mathbf{y}), \ldots, m^{(n)}(\mathbf{x}) + m^{(n)}(\mathbf{y})) \Big) \\
&= \langle \mathbf{x} + \mathbf{y} \rangle, \\
\mathbf{a} + \langle \mathbf{x} \rangle &= \Big((\mathbf{x}^{(1)} + \mathbf{a}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(n)}), \\
&\quad (\boldsymbol{\alpha}^{(1)} * C(\mathbf{a}) + m^{(1)}(\mathbf{x}), \ldots, \boldsymbol{\alpha}^{(n)} * C(\mathbf{a}) + m^{(n)}(\mathbf{x})) \Big) \\
&= \langle \mathbf{a} + \mathbf{x} \rangle, \\
\mathbf{a} * \langle \mathbf{x} \rangle &= \Big((C^{*-1}(C(\mathbf{a}) * C(\mathbf{x}^{(1)})), \ldots, C^{*-1}(C(\mathbf{a}) * C(\mathbf{x}^{(n)})), \\
&\quad (C(\mathbf{a}) * m^{(1)}(\mathbf{x}), \ldots, C(\mathbf{a}) * m^{(n)}(\mathbf{x})) \Big) \\
&= \langle \mathbf{t} \rangle^*,
\end{aligned}$$

where $\pi_k(\mathbf{t}) = \mathbf{a} * \mathbf{x}$ and $C^{*-1}(\mathbf{y})$ means the vector in $\mathbb{F}_2^{k^*}$ which is encoded to $\mathbf{y}$. Notice that this also describes how the parties locally can compute an addition gate and furthermore, it shows that when multiplying by a constant vector we end up with a $\langle \cdot \rangle^*$-sharing. This is what happens in the multiply step, where we end up with $\langle \boldsymbol{\rho} \rangle^*$. Notice that $\pi_k(\boldsymbol{\rho}) = \mathbf{x} * \mathbf{y}$. However, we need to transform it back to a $\langle \cdot \rangle$-sharing and this is what we need the preprocessed reencode tuple for. To see that the steps in the protocol make this transformation correctly notice that

$$\pi_k(\boldsymbol{\sigma}) + \mathbf{r} = \pi_k(\boldsymbol{\rho}) - \pi_k(\mathbf{s}) + \mathbf{r} = \mathbf{x} * \mathbf{y} - \mathbf{r} + \mathbf{r} = \mathbf{x} * \mathbf{y}.$$

We will not go into details about the MAC check for MiniMAC but we will mention that they do a batch check where they check random $\mathbb{F}_2^{\ell}$-linear combinations of the encodings of all opened vectors. For more information about the check see [14].

C Proofs

Proof (Theorem 2). The initialization phase is just communication with $\mathcal{F}_{\text{Prep}}$. For simulating an input by a party P_i, if the party who inputs the value is not corrupted, then the simulator samples and broadcasts a random $\boldsymbol{\epsilon}$. If it is corrupted, then when the adversary broadcasts $\boldsymbol{\epsilon}$, then the simulator extracts $\mathbf{x} = \boldsymbol{\epsilon} + \mathbf{r}$ and inputs $(\text{Input}, \text{id}, \mathbf{x}, P_i)$ to $\mathcal{F}_{\text{MPC}}$. Additions consist of local computations and are trivial to simulate. For every multiplication, the simulator generates uniformly random vectors $\boldsymbol{\epsilon}$ and $\boldsymbol{\delta}$ and a uniformly random

Protocol Π_{MiniMAC}

1. **Initialize:** The parties call the preprocessing functionality $\mathcal{F}_{\text{Prep}}$ to obtain input tuples $(\mathbf{r}, \langle \mathbf{r} \rangle)$ for each party, re-encode tuples $(\langle \mathbf{r} \rangle, \langle \mathbf{s} \rangle^*)$, where $\pi_k(\mathbf{s}) = \mathbf{r}$, and multiplication triples $(\langle \mathbf{a} \rangle, \langle \mathbf{b} \rangle, \langle \mathbf{c} \rangle^*)$, where $\pi_k(\mathbf{c}) = \mathbf{a} * \mathbf{b}$.
2. **Input:** For an input gate belonging to P_i having input $\mathbf{x} \in \mathbb{F}_2^k$ the parties do the following
   (a) P_i takes a tuple $(\mathbf{r}, \langle \mathbf{r} \rangle)$ and broadcasts $\boldsymbol{\epsilon} = \mathbf{x} - \mathbf{r}$.
   (b) The parties compute $\langle \mathbf{x} \rangle = \boldsymbol{\epsilon} + \langle \mathbf{r} \rangle$.
3. **Add:** To compute componentwise addition of $\langle \mathbf{x} \rangle$ and $\langle \mathbf{y} \rangle$ the parties locally compute $\langle \mathbf{x}+\mathbf{y} \rangle = \langle \mathbf{x} \rangle + \langle \mathbf{y} \rangle$.
4. **Multiply:** To compute a componentwise multiplication of $\langle \mathbf{x} \rangle$ and $\langle \mathbf{y} \rangle$, take the next available multiplication triple $(\langle \mathbf{a} \rangle, \langle \mathbf{b} \rangle, \langle \mathbf{c} \rangle^*)$ and tuple $(\langle \mathbf{r} \rangle, \langle \mathbf{s} \rangle^*)$.
   (a) Set $\langle \boldsymbol{\epsilon} \rangle = \langle \mathbf{x} \rangle - \langle \mathbf{a} \rangle$ and $\langle \boldsymbol{\delta} \rangle = \langle \mathbf{y} \rangle - \langle \mathbf{b} \rangle$ and partially open $\boldsymbol{\epsilon}$ and $\boldsymbol{\delta}$.
   (b) Compute $\langle \boldsymbol{\rho} \rangle^* = \langle \mathbf{c} \rangle^* + \boldsymbol{\epsilon} * \langle \mathbf{y} \rangle + \boldsymbol{\delta} * \langle \mathbf{x} \rangle - \boldsymbol{\epsilon} * \boldsymbol{\delta}$
   (c) Compute $\langle \boldsymbol{\sigma} \rangle^* = \langle \boldsymbol{\rho} \rangle^* - \langle \mathbf{s} \rangle^*$ and partially open this value to obtain $\boldsymbol{\sigma} \in \mathbb{F}_2^{k^*}$.
   (d) Compute $\pi_k(\boldsymbol{\sigma}) + \langle \mathbf{r} \rangle = \langle \mathbf{x} * \mathbf{y} \rangle$ and output this value.
5. **Output:** This stage is entered when the players have an unopened sharing $\langle \mathbf{z} \rangle$ which they want to output. The parties do the following:
   (a) Execute a MAC check on all opened vectors.
   (b) If the check passes, partially open $\mathbf{z}$.
   (c) Execute a MAC check on $\mathbf{z}$
   (d) If the check passes, output $\mathbf{z}$ to all parties.

Figure 16. Online phase of MiniMAC

field element σ. The simulator sends these values to the (internal copy of the) adversary who opens $(\boldsymbol{\epsilon}', \boldsymbol{\delta}', \sigma')$. If for some multiplication, the tuple $(\boldsymbol{\epsilon}', \boldsymbol{\delta}', \sigma')$ is different from the one sent by the simulator, the simulator will abort when simulating the first check in the output phase. If not, the simulator receives the output $\mathbf{z}$ from $\mathcal{F}_{\text{MPC}}$.Output and sends this to the adversary. If the adversary replies with a value $\mathbf{z}'$ which is different from $\mathbf{z}$ then the simulator aborts at the second check.

Proof (Proposition 1). The commands Initialize and ExtendField are as in [19] (the latter being called Extend there). The proof for our ExtendVector command is analogous to the one for the ExtendField except, as explained, because the ideal functionality restricts the choice by a corrupt P_A of the element that is secret shared. We briefly show the simulation of ExtendVector together with Initialize.

If P_B is corrupted, the simulator receives $(\alpha_0, \ldots, \alpha_{m-1})$ from the adversary, and simulates the initialization phase by sampling the seeds at random, and sending the corresponding one to the adversary. It simulates the ExtendVector phase by choosing $\mathbf{u}_i$, uniformly at random in the corresponding domain, computes q as an honest P_B would do and inputs this to the functionality. Indistinguishability holds by the pseudorandomness of F, as shown in [19].

If P_A is corrupted then the simulator receives the seeds from the adversary in the Initialize phase, and from there it computes all the $\mathbf{t}_b^i$ in the ExtendVector phase. Then when the adversary sends $\mathbf{u}_i$, the simulators extract $\mathbf{x}_i = \mathbf{u}_i - \mathbf{t}_0^i + \mathbf{t}_1^i$ and inputs $t = -\sum_{i=0}^{m-1} \phi(\mathbf{t}_0^i) \cdot X^i$ and $(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m)$ to $\mathcal{F}_{\text{COPEe}}$.

In this case all outputs are clearly computed as in the real world and indistinguishability follows.

Proof (Proposition 2). First of all note that our functionality, in contrary to $\Pi_{[[\cdot]]}$ from [19], has an Add command and a LinComb command. This is because we reserve the LinComb command for linear combinations which output $[\cdot]$-sharings, while Add outputs a $\langle\cdot\rangle$-sharing. In any case, the Add and LinComb command consist of local computations so it is trivial to argue their security. The Initialize command only invokes the Initialize command from the ideal functionality $\mathcal{F}_{\text{COPEe}}$, which is exactly the same as in [19]. Since the Open command lets the adversary choose what to open to there is not much to discuss here either.

Therefore, what we need to discuss is the Input and Check commands. The idea is that if the check in the input phase is passed and the adversary opens to incorrect values later on, then the probability to pass a check later on will be negligible. In comparison to [19], we have both values in $\mathbb{F}_{2^m}$ and vectors in $\mathbb{F}_2^k$, but we can still use the same arguments there, because the check in the Input phase and all further checks are in $\mathbb{F}_{2^m}$ and therefore the simulation and indistinguishability is following by the exact same arguments as in [19].

Proof (Proposition 4). First notice that at least one of the parties is honest and hence $r_j = \sum_{i=1}^{n} r_j^{(i)}$ is random because one of the terms is. Suppose that at the end of the Combine phase parties have created $(\langle \mathbf{s}_j \rangle, [r_j])$, where possibly $\mathbf{s}_j \neq \psi(r_j)$.

Let $\boldsymbol{\epsilon}_j = \mathbf{s}_j - \psi(r_j)$ for all j. By $\mathbb{F}_2$-linearity of ψ, $\mathbf{b}_i - \psi(b_i) = \sum_{j=1}^{t+s} a_{ij}\boldsymbol{\epsilon}_j$. Hence if all $\epsilon_j = 0$, the check passes for all i. While if there is some $\boldsymbol{\epsilon}_j \neq 0$, $j = 1, \ldots, t$, then for every i the probability that $\sum_{j=1}^{t+s} a_{ij}\boldsymbol{\epsilon}_j = 0$ is at most $1/2$.

Since the checks are independent we obtain that if some $\boldsymbol{\epsilon}_j \neq 0$, $j = 1, \ldots, t$ then the protocol will abort except with probability at most 2^{-s}. Note also that $b_i = r_{t+i} + \sum_{j=1}^{t} a_{ij} r_j$, so opening the b_i reveals no information about the output values $r_1, \ldots, r_t$.

D Triple creation

Denote by $a_{h,l}^{(j,i)}$ and $\mathbf{b}_{h,l}^{(j,i)}$ the actual values sent by a corrupt party to an honest party where it should have sent $a_{h,l}^{(j)}$ and $\mathbf{b}_l^{(j)}$ in the Construction step. We can fix some $a_{h,l}^{(j)}$ and $\mathbf{b}_l^{(j)}$ to be considered as the "input" for some specific instance (for example lowest index of honest party i, and lowest h for $\mathbf{b}_l^{(j)}$) and define the errors $\mathbf{e}_{\mathbf{b}_{h,l}}^{(j,i)} = \mathbf{b}_{h,l}^{(j,i)} - \mathbf{b}_l^{(j)}$ and $e_{a_{h,l}}^{(j,i)} = a_{h,l}^{(j,i)} - a_{h,l}^{(j)}$. Denoting the set of corrupt parties by A, we define $\mathbf{e}_{\mathbf{b}_{h,l}}^{(i)} = \sum_{j \in A} \mathbf{e}_{\mathbf{b}_{h,l}}^{(j,i)}$ and $e_{a_{h,l}}^{(i)} = \sum_{j \in A} e_{a_{h,l}}^{(j,i)}$. Summing up the shares $c_l^{(i)}$ we see that we end up with

$$c_l = \phi(\mathbf{a}_l) \cdot \phi(\mathbf{b}_l) + \sum_{i \notin A} \phi(\mathbf{b}_l^{(i)}) \cdot \sum_{h=0}^{m-1} e_{a_{h,l}}^{(i)} \cdot X^h + \sum_{i \notin A} \sum_{h=0}^{m-1} a_{h,l}^{(i)} \cdot \phi(\mathbf{e}_{\mathbf{b}_{h,l}}^{(i)}) \cdot X^h,$$

where the adversary controls $e_{a_{h,l}}^{(i)}$ and $\mathbf{e}_{\mathbf{b}_{h,l}}^{(i)}$. Denoting by $e_{a,l} = \sum_{i \notin A} \phi(\mathbf{b}_l^{(i)}) \cdot \sum_{h=0}^{m-1} e_{a_{h,l}}^{(i)} \cdot X^h$ and $e_{b,l} = \sum_{i \notin A} \sum_{h=0}^{m-1} a_{h,l}^{(i)} \cdot \phi(\mathbf{e}_{\mathbf{b}_{h,l}}^{(i)}) \cdot X^h$ we see that

$$c_l = \phi(\mathbf{a}_l) \cdot \phi(\mathbf{b}_l) + e_{a,l} + e_{b,l}$$

after the construction step. Additionally, the adversary can add an extra error by authenticating to another value. That is, the adversary can introduce an error $e_{auth,l}$ such that

$$c_l = \phi(\mathbf{a}_l) \cdot \phi(\mathbf{b}_l) + e_{a,l} + e_{b,l} + e_{auth,l}.$$

We call the triple malformed if $e_{a,l} + e_{b,l} + e_{auth,l} \neq 0$.

We now discuss that the bucketing technique in [15] guarantees that after the cut-and-choose and sacrifice steps, if the protocol does not abort then the surviving triples are not malformed with very large probability. This is based in the following lemma.

Lemma 1 ([15],[18]). *Let $N = \tau_1 + \tau_1 \cdot \tau_2^2 \cdot T$ be the number of constructed triples where the statistical security parameter satisfies $s < \log_2\left(\frac{N!}{\tau_2^2 \cdot T \cdot \tau_1! \cdot (\tau_1 \cdot \tau_2^2 \cdot T)!}\right)$. If τ_1 random triples are opened and all are correct, splitting the remaining $\tau_1 \cdot \tau_2^2 \cdot T$ into buckets of size τ_1 will ensure that except with probability 2^{-s} either all buckets consist of correct triples or there will be at least one bucket with both correct and malformed triples.*

The lemma states that if the cut-and-choose step passes, we will be in one of the two situations described with large probability. Notice that in the first case the sacrifice step will pass and we end up with $\tau_2^2 \cdot T$ correct triples. In the second case there will be some bucket where the protocol aborts in the sacrifice step. To see that the sacrifice step aborts notice that if there is a pair of triples where one is malformed and the other is not, then there exists an index l such that either the first or the l-th triple, but not both, is malformed. Then when opening

$$\sigma_l = c_l - c_1 + \phi(\mathbf{a}_1) \cdot \phi(\mathbf{b}_1) - \phi(\mathbf{a}_l) \cdot \phi(\mathbf{b}_l),$$

this will not open towards 0.

However, after the cut-and-choose and sacrifice phases passes, the adversary may now have information about some of the triplets. This is because of the following selective failure attacks.

Denote by $e_{a,l}^{(i)} = \sum_{h=0}^{m-1} e_{a_{h,l}}^{(i)} \cdot X^h$ and notice that $e_{a,l} = \sum_{i \notin A} e_{a,l}^{(i)} \cdot \phi(\mathbf{b}_l^{(i)})$. Assume that the adversary has chosen $e_{a_{h,l}}^{(i)} \neq 0$ for some h's and a single i. This implies that $e_{a,l}^{(i)}$ is nonzero, and then $e_{a,l}$ is only zero if $\mathbf{b}_l^{(i)} = 0$, which happens with probability 2^{-k}. This means that even though the adversary has introduced some error, the check will pass with relatively high probability (recall that in our case we consider 2^{-k} too large, as we want the success probability to be at most 2^{-m}), and in that case the adversary has obtained some information about $\mathbf{b}_l^{(i)}$. The argument generalizes to the case where the adversary chooses $e_{a_{h,l}}^{(i)} \neq 0$ for several i's.

Analogously, assume that the adversary has chosen $\mathbf{e}_{\mathbf{b}_{h,l}}^{(i)} \neq 0$ for a single h and i. Then $e_{b,l} = 0$ if and only if $a_{h,l}^{(i)} = 0$ for the same h and i, which happens with probability $\frac{1}{2}$. Note that the probability decreases if $\mathbf{e}_{\mathbf{b}_{h,l}}^{(i)} \neq 0$ for several h's and that the argument generalizes to the case where the adversary chooses $e_{\mathbf{b}_{h,l}}^{(i)} \neq 0$ for several i's.

In this way, the adversary can be lucky to introduce some errors which cancel out and cause the triples to be correct, and this fact will give the adversary information about some parts of $\mathbf{a}$ and $\mathbf{b}$, when the protocol does not abort when opening values in the sacrifice step.

To make sure that this leakage is cleaned, we execute a combine step in order to re-establish the randomness.

We call the l-th triple leaky if the adversary has introduced some errors, i.e. if $e_{a_{h,l}}^{(j,i)}$ or $\mathbf{e}_{\mathbf{b}_{h,l}}^{(j,i)}$ is nonzero, but the resulting triple is correct (not malformed). With very high probability, at most s triples will be leaky if the sacrifice phase has succeeded.

In order to remove the leakage we apply the Combine steps. For this we need to ensure that after the sacrifice step there is at least one non-leaky triple in each bucket. This is ensured by the following lemma.

Lemma 2 ([15]). *Inputting at least* $\sqrt[\tau_2-1]{\frac{(s\cdot e)\tau_2\cdot 2^s}{\tau_2}}$ *triples to the combine step where at most s of them are leaky in the component being combined on, we have that every bucket of τ_2 triples contains at least one non-leaky triple (in the component) with overwhelming probability in s.*

Notice that if a bucket contains at least one non-leaky triple in the component being combined on the outputted triple cannot be leaky in that component and hence $\mathbf{a}$ and $\mathbf{b}$ are random in $\mathbb{F}_2^k$ after the combine steps.