A Unified Approach to MPC with Preprocessing using OT

Tore Kasper Frederiksen[1], Marcel Keller[2], Emmanuela Orsini[2], and Peter Scholl[2]

[1] Department of Computer Science, Aarhus University
[2] Department of Computer Science, University of Bristol
jot2re@cs.au.dk, {m.keller,emmanuela.orsini,peter.scholl}@bristol.ac.uk

Abstract. SPDZ, TinyOT and MiniMAC are a family of MPC protocols based on secret sharing with MACs, where a preprocessing stage produces multiplication triples in a finite field. This work describes new protocols for generating multiplication triples in fields of characteristic two using OT extensions. Before this work, TinyOT, which works on binary circuits, was the only protocol in this family using OT extensions. Previous SPDZ protocols for triples in large finite fields require somewhat homomorphic encryption, which leads to very inefficient runtimes in practice, while no dedicated preprocessing protocol for MiniMAC (which operates on vectors of small field elements) was previously known. Since actively secure OT extensions can be performed very efficiently using only symmetric primitives, it is highly desirable to base MPC protocols on these rather than expensive public key primitives. We analyze the practical efficiency of our protocols, showing that they should all perform favorably compared with previous works; we estimate our protocol for SPDZ triples in $\mathbb{F}_{2^{40}}$ will perform around 2 orders of magnitude faster than the best known previous protocol.

Keywords: MPC, SPDZ, TinyOT, MiniMAC, Preprocessing, OT extension

This article is based on an earlier article published by Springer-Verlag at Asiacrypt 2015, ©IACR 2015, DOI:TBD.

Table of Contents

1 Introduction

Secure multi-party computation (MPC) allows parties to perform computations on their private inputs, without revealing their inputs to each other. Recently, there has been much progress in the design of practical MPC protocols that can be efficiently implemented in the real world. These protocols are based on secret sharing over a finite field, and they provide security against an active, static adversary who can corrupt up to $n-1$ of n parties (dishonest majority).

In the *preprocessing model*, an MPC protocol is divided into two phases: a *preprocessing* (or *offline*) phase, which is independent of the parties' inputs and hence can be performed in advance, and an *online* phase. The preprocessing stage only generates random, correlated data, often in the form of secret shared multiplication triples [2]. The online phase then uses this correlated randomness to perform the actual computation; the reason for this separation is that the online phase can usually be much more efficient than the preprocessing, which results in a lower latency during execution than if the whole computation was done together. This paper builds on the so-called 'MPC with MACs' family of protocols, which use information-theoretic MACs to authenticate secret-shared data, efficiently providing active security in the online phase, starting with the work of Bendlin et al. [4]. We focus on the SPDZ [10], MiniMAC [11] and TinyOT [18] protocols, which we now describe.

The 'SPDZ' protocol of Damgård et al. [10,8] evaluates arithmetic circuits over a finite field of size at least 2^k, where k is a statistical security parameter. All values in the computation are represented using additive secret sharing and with an additive secret sharing of a MAC that is the product of the value and a secret key. The online phase can be essentially performed with only information theoretic techniques and thus is extremely efficient, with throughputs of almost 1 million multiplications per second as reported by Keller et al. [15]. The preprocessing of the triples uses somewhat homomorphic encryption (SHE) to create an initial set of triples, which may have errors due to the faulty distributed decryption procedure used. These are then paired up and a 'sacrificing' procedure is done: one triple is wasted to check the correctness of another. Using SHE requires either expensive zero knowledge proofs or cut-and-choose techniques to achieve active security, which are *much* slower than the online phase – producing a triple in $\mathbb{F}_p$ (for 64-bit prime p) takes around 0.03s [8], whilst $\mathbb{F}_{2^{40}}$ triples are even more costly due to the algebra of the homomorphic encryption scheme, taking roughly 0.27s [7].

TinyOT [18] is a two-party protocol for binary circuits based on OT extensions. It has similar efficiency to SPDZ in the online phase but has faster preprocessing, producing around 10000 $\mathbb{F}_2$ triples per second. Larraia et al. [16] extended TinyOT to the multi-party setting and adapted it to fit with the SPDZ online phase. The multi-party TinyOT protocol also checks correctness of triples using sacrificing, and two-party TinyOT uses a similar procedure called combining to remove possible leakage from a triple, but when working in small fields simple pairwise checks are not enough. Instead an expensive 'bucketing' method is used, which gives an overhead of around 3-8 times for each check, depending on the number of triples required and the statistical security parameter.

MiniMAC [11] is another protocol in the SPDZ family, which reduces the size of MACs in the online phase for the case of binary circuits (or arithmetic circuits over small fields). Using SPDZ or multi-party TinyOT requires the MAC on every secret shared value to be at least as big as the statistical security parameter, whereas MiniMAC can authenticate vectors of bits at once combining them into a codeword, allowing the MAC size to be constant. Damgård et al. [9] implemented the online phase of MiniMAC and found it to be faster than TinyOT for performing many operations in parallel, however no dedicated preprocessing protocol for MiniMAC has been published.

1.1 Our Contributions

In this paper we present new, improved protocols for the preprocessing stages of the 'MPC with MACs' family of protocols based on OT extensions, focusing on finite fields of characteristic two. Our main contribution is a new method of creating SPDZ triples in $\mathbb{F}_{2^k}$ using only symmetric primitives, so it is much more efficient than previous protocols using SHE. Our protocol is based on a novel correlated OT extension protocol that increases efficiency by allowing an adversary to introduce errors of a specific form, which may be

Finite field	Protocol	# Correlated OTs	# Random OTs	Time estimate, ms ($n = 2$)
$\mathbb{F}_2$	2-party TinyOT [18,5]	0	54	0.07
	n-party TinyOT [16,5]	$81n(n-1)$	$27n(n-1)$	0.24
	This work §5.2	$27n(n-1)$	$9n(n-1)$	0.08
$\mathbb{F}_{2^{40}}$	SPDZ [7]	N/A	N/A	272
	This work §5.1	$240n(n-1)$	$240n(n-1)$	1.13
$\mathbb{F}_{2^8}$ (MiniMAC)	**This work** §6	$1020n(n-1)$	$175n(n-1)$	2.63

Table 1. Number of OTs and estimates of time required to create a multiplication triple using our protocols and previous protocols, for n parties. See Section 7 for details.

of independent interest. Additionally, we revisit the multi-party TinyOT protocol by Larraia et al. from CRYPTO 2014 [16], and identify a crucial security flaw that results in a selective failure attack. A standard fix has an efficiency cost of at least 9x, which we show how to reduce to just 3x with a modified protocol. Finally, we give the first dedicated preprocessing protocol for MiniMAC, by building on the same correlated OT that lies at the heart of our SPDZ triple generation protocol.

Table 1 gives the main costs of our protocols in terms of the number of correlated and random OTs required, as well as an estimate of the total time per triple, based on OT extension implementation figures. We include the SPDZ protocol timings based on SHE to give a rough comparison with our new protocol for $\mathbb{F}_{2^{40}}$ triples. For a full explanation of the derivation of our time estimates, see Section 7. Our protocol for $\mathbb{F}_{2^{40}}$ triples has the biggest advantage over previous protocols, with an estimated 200x speed-up over the SPDZ implementation. For binary circuits, our multi-party protocol is comparable with the two-party TinyOT protocol and around 3x faster than the fixed protocol of Larraia et al. [16]. For MiniMAC, we give figures for the amortized cost of a single multiplication in $\mathbb{F}_{2^8}$. This seems to incur a slight cost penalty compared with using SPDZ triples and embedding the circuit in $\mathbb{F}_{2^{40}}$, however this is traded off by the more efficient online phase of MiniMAC when computing highly parallel circuits [9].

We now highlight our contributions in detail.

$\mathbb{F}_{2^k}$ Triples. We show how to use a new variant of correlated OT extension to create multiplication triples in the field $\mathbb{F}_{2^k}$, where k is at least the statistical security parameter. Note that this finite field allows much more efficient evaluation of AES in MPC than using binary circuits [7], and is also more efficient than $\mathbb{F}_p$ for computing ORAM functionalities for secure computation on RAM programs [14]. Previously, creating big field triples for the SPDZ protocol required using somewhat homomorphic encryption and therefore was very slow (particularly for the binary field case, due to limitations of the underlying SHE plaintext algebra [7]). It seems likely that our OT based protocol can improve the performance of SPDZ triples by 2 orders of magnitude, since OT extensions can be performed very efficiently using just symmetric primitives.

The naive approach to achieving this is to create k^2 triples in $\mathbb{F}_2$, and use these to evaluate the $\mathbb{F}_{2^k}$ multiplication circuit. Each of these $\mathbb{F}_2$ triples would need sacrificing and combining, in total requiring many more than k^2 OT extensions. Instead, our protocol in Section 5.1 creates a $\mathbb{F}_{2^k}$ triple using only $O(k)$ OTs. The key insight into our technique lies in the way we look at OT: instead of taking the traditional view of a sender and a receiver, we use a linear algebra approach with matrices, vectors and tensor products, which pinpoints the precise role of OT in secure computation. A correlated OT is a set of OTs where the sender's messages are all $(x, x + \Delta)$ for some fixed string Δ. We represent a set of k correlated OTs between two parties, with inputs $\mathbf{x}, \mathbf{y} \in \mathbb{F}_2^k$, as:

$$Q + T = \mathbf{x} \otimes \mathbf{y}$$

where $Q, T \in \mathbb{F}_2^{k \times k}$ are the respective outputs to each party. Thus, correlated OT gives precisely a secret sharing of the tensor product of two vectors. From the tensor product it is then straightforward to obtain a

$\mathbb{F}_{2^k}$ multiplication of the corresponding field elements by taking the appropriate linear combination of the components.

An actively secure protocol for correlated OT was presented by Nielsen et al. [18], with an overhead of ≈ 7.3 calls to the base OT protocol due to the need for consistency checks and privacy amplification, to avoid any leakage on the secret correlation. In our protocol, we choose to miss out the consistency check, allowing the party creating correlation to input different correlations to each OT. We show that if this party attempts to cheat then the error introduced will be amplified by the privacy amplification step so much that it can always be detected in the pairwise sacrificing check we later perform on the triples. Allowing these errors significantly complicates the analysis and security proofs, but reduces the overhead of the correlated OT protocol down to just 3 times that of a basic OT extension.

$\mathbb{F}_2$ Triples. The triple production protocol by Larraia et al. [16] has two main stages: first, unauthenticated shares of triples are created (using the aBit protocol by Nielsen et al. [18] as a black box) and secondly the shares are authenticated, again using aBit, and checked for correctness with a sacrificing procedure. The main problem with this approach is that given shares of an unauthenticated triple for $a, b, c \in \mathbb{F}_2$ where $c = a \cdot b$, the parties may not input their correct shares of this triple into the authentication step. A corrupt party can change their share such that $a + 1$ is authenticated instead of a; if $b = 0$ (with probability $1/2$) then $(a + 1) \cdot b = a \cdot b$, the sacrificing check still passes, and the corrupt party hence learns the value of b.[3]

To combat this problem, an additional *combining* procedure can be done: similarly to sacrificing, a batch of triples are randomly grouped together into buckets and combined, such that as long as one of them is secure, the resulting triple remains secure, as was done by Nielsen et al. [18]. However, combining only removes leakage on either a or b. To remove leakage on both a and b, combining must be done twice, which results in an overhead of at least 9x, depending on the batch size. Note that this fix is described in full in a recent preprint [5], which is a merged and extended version of the two TinyOT papers [18,16]

In Section 5.2 we modify the triple generation procedure so that combining only needs to be done once, reducing the overhead on top of the original (insecure) protocol to just 3x (for a large enough batch of triples). Our technique exploits the structure of the OT extension protocol to allow a triple to be created, whilst simultaneously authenticating one of the values a or b, preventing the selective failure attack on the other value. Combining still needs to be performed once to prevent leakage, however.

MiniMAC Triples. The MiniMAC protocol [11] uses multiplication triples of the form $C^*(\mathbf{c}) = C(\mathbf{a}) * C(\mathbf{b})$, where $\mathbf{a}, \mathbf{b} \in \mathbb{F}_{2^u}^k$ and C is a systematic, linear code over $\mathbb{F}_{2^u}$, for 'small' u (e.g. $\mathbb{F}_2$ or $\mathbb{F}_{2^8}$), $*$ denotes the component-wise vector product and C^* is the product code given by the span of all products of codewords in C. Based on the protocol for correlated OT used for the $\mathbb{F}_{2^k}$ multiplication triples, we present the first dedicated construction of MiniMAC multiplication triples. The major obstacles to overcome are that we must somehow guarantee that the triples produced form valid codewords. This must be ensured both during the triple generation stage and the authentication stage, otherwise another subtle selective failure attack can arise. To do this, we see $\mathbf{a}$ and $\mathbf{b}$ as vectors over $\mathbb{F}_2^{u \cdot k}$ and input these to the same secure correlated OT procedure as used for the $\mathbb{F}_{2^k}$ multiplication triples. From the resulting shared tensor product, we can compute shares of all of the required products in $C(\mathbf{a}) * C(\mathbf{b})$, due to the linearity of the code. For authentication we use the same correlated OT as used for authentication of the $\mathbb{F}_{2^k}$ triples. However, this only allows us to authenticate components in $\mathbb{F}_{2^u}$ one at a time, so we also add a "compression" step to combine individual authentications of each component in $C(\mathbf{x})$ into a single MAC. Finally, the construction is ended with a pairwise sacrificing step.

Furthermore, since the result of multiplication of two codewords results in an element in the Schur transform, we need some more preprocessed material, in order to move such an element back down to an "ordinary" codeword. This is done using an authenticated pair of equal elements; one being an ordinary codeword and one in the Schur transform of the code. We also construct these pairs by authenticating the

[3] We stress that this attack only applies to the multi-party protocol from CRYPTO 2014 [16], and not the original two-party protocol of Nielsen et al. [18].

k components in $\mathbb{F}_{2^u}$ and then, using the linearity of the code, computing authenticated shares of the entire codeword. Since this again results in a MAC for each component of the codeword we execute a compression step to combine the MAC's into a single MAC.

Efficient Authentication from Passively Secure OT. All of our protocols are unified by a common method of authenticating shared values using correlated OT extension. Instead of using an actively secure correlated OT extension protocol as was previously done [18,16], we use just a passively secure protocol, which is simply the passive OT extension of Ishai et al. [12], without the hashing at the end of the protocol (which removes the correlation).

This allows corrupt parties to introduce errors on MACs that depend on the secret MAC key, which could result in a few bits of the MAC key being leaked if the MAC check protocol still passes. Essentially, this means that corrupt parties can try to guess subsets of the field in which the MAC key shares lie, but if their guess is incorrect the protocol aborts. We model this ability in all the relevant functionalities, showing that the resulting protocols are actively secure, even when this leakage is present.

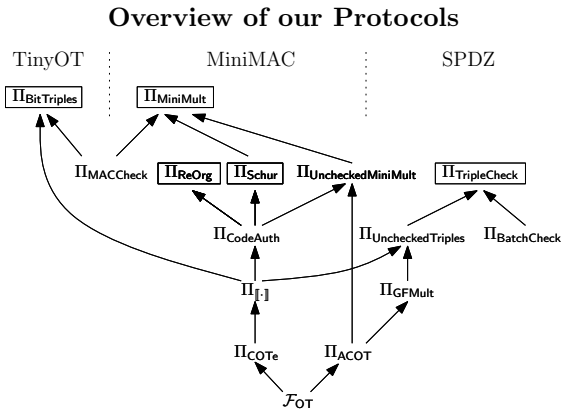

Fig. 1. Illustration of the relationship between our protocols. Protocols in boxes indicate final elements for use in online execution.

Security. The security of our protocols is proven in the standard UC framework of Canetti [6] (see Appendix A for details). We consider security against malicious, static adversaries, i.e. corruption may only take place before the protocols start, corrupting up to $n-1$ of n parties.

Setup Assumption. The security of our protocols is in the $\mathcal{F}_{\mathsf{OT}}$-hybrid model, *i.e.* all parties have access to an ideal 1-out-of-2 OT functionality. Moreover we assume *authenticated* communication between parties, in the form of a functionality $\mathcal{F}_{\mathsf{AT}}$ which, on input (m, i, j) from P_i, gives m to P_j and also leaks m to the adversary. Our security proof for $\mathbb{F}_2$ triples also uses the random oracle (RO) model [3] to model the hash function used in an OT extension protocol. This means that the parties and the adversaries have access to a uniformly random $H : \{0,1\}^* \to \{0,1\}^\kappa$, such that if it is queried on the same input twice, it returns the same output. We also use a standard coin flipping functionality, $\mathcal{F}_{\mathsf{Rand}}$, which can be efficiently implemented using hash-based commitments in the random oracle model as done previously [8].

Overview. The rest of this paper is organized as follows: In Section 2 we go through our general notation, variable naming and how we represent shared values. We continue in Section 3 with a description of the passively secure OT extensions we use as building block for our triple generation and authentication. We then go into more details on our authentication procedure in Section 4. This is followed by a description of how we generate TinyOT ($\mathbb{F}_2$) and SPDZ ($\mathbb{F}_{2^k}$) triples in Section 5 and MiniMAC triples in Section 6. We end with a complexity analysis in Section 7.

We illustrate the relationship between all of our protocols in Fig. 1. In the top we have the protocol producing final triples used in online execution and on the bottom the protocols for correlated OT extension and authentication.

2 Notation

We denote by κ the computational security parameter and s the statistical security parameter. We let $\mathsf{negl}(\kappa)$ denote some unspecified function $f(\kappa)$, such that $f = o(\kappa^{-c})$ for every fixed constant c, saying that such a

function is *negligible* in κ. We say that a probability is *overwhelming* in κ if it is $1 - \mathsf{negl}(\kappa)$. We denote by $a \xleftarrow{\$} A$ the random sampling of a from a distribution A, and by $[d]$ the set of integers $\{1, \dots d\}$.

We consider the sets $\{0,1\}$ and $\mathbb{F}_2^\kappa$ endowed with the structure of the fields $\mathbb{F}_2$ and $\mathbb{F}_{2^\kappa}$, respectively. We denote by $\mathbb{F}$ any finite field of characteristic two, and use roman lower case letters to denote elements in $\mathbb{F}$, and bold lower case letters for vectors. We will use the notation $\mathbf{v}[i]$ to denote the i-th entry of $\mathbf{v}$. Sometimes we will use $\mathbf{v}[i;j]$ to denote the range of bits from i to j when viewing $\mathbf{v}$ as a bit vector. Given matrix A, we denote its rows by subindices $\mathbf{a}_i$ and its columns by superindices $\mathbf{a}^j$. If we need to denote a particular entry we use the notation $A[i,j]$. We will use $\mathcal{O}$ to denote the matrix full of ones and $D_{\mathbf{x}}$ for some vector $\mathbf{x}$ to denote the square matrix whose diagonal is $\mathbf{x}$ and where every other positions is 0.

We use $\cdot$ to denote multiplication of elements in a finite field; note that in this case we often switch between elements in the field $\mathbb{F}_{2^\kappa}$, vectors in $\mathbb{F}_2^\kappa$ and vectors in $\mathbb{F}_{2^u}^{\kappa/u}$ (where $u|\kappa$), but when multiplication is involved we always imply multiplication over the field, or and entry-wise multiplication if the first operand is a scalar. If $\mathbf{a}, \mathbf{b}$ are vectors over $\mathbb{F}$ then $\mathbf{a} * \mathbf{b}$ denotes the component-wise product of the vectors, and $\mathbf{a} \otimes \mathbf{b}$ to denote the matrix containing the tensor (or outer) product of the two vectors.

We consider a systematic linear error correcting code C over finite field $\mathbb{F}_{2^u}$ of length m, dimension k and distance d. So if $\mathbf{a} \in \mathbb{F}_{2^u}^k$, we denote by $C(\mathbf{a}) \in \mathbb{F}_{2^u}^m$ the encoding of $\mathbf{a}$ in C, which contains $\mathbf{a}$ in its first k positions, due to the systematic property of the code. We let C^* denote the product code (or Schur transform) of C, which consists of the linear span of $C(\mathbf{a}) * C(\mathbf{b})$, for all vectors $\mathbf{a}, \mathbf{b} \in \mathbb{F}_{2^u}^k$. If C is a $[m,k,d]$ linear error correcting code then C^* is a $[m,k^*,d^*]$ linear error correcting code for which it holds that $k^* \geq k$ and $d^* \leq d$.

2.1 Authenticating Secret-shared Values

Let $\mathbb{F}$ be a finite field, we additively secret share bits and elements in $\mathbb{F}$ among a set of parties $\mathcal{P} = \{P_1, \dots, P_n\}$, and sometimes abuse notation identifying subsets $\mathcal{I} \subseteq \{1, \dots, n\}$ with the subset of parties indexed by $i \in \mathcal{I}$. We write $\langle a \rangle$ if a is additively secret shared amongst the set of parties, with party P_i holding a value $a^{(i)}$, such that $\sum_{i \in \mathcal{P}} a^{(i)} = a$. We adopt the convention that, if $a \in \mathbb{F}$ then the shares also lie in the same field, i.e. $a^{(i)} \in \mathbb{F}$.

Our main technique for authentication of secret shared values is similar to the one by Larraia et al. [16] and Damgård et al. [10], i.e. we authenticate a secret globally held by a system of parties, by placing an *information theoretic tag* (MAC) on the secret shared value. We will use a fixed global key $\Delta \in \mathbb{F}_{2^M}$, $M \geq \kappa$, which is additively secret shared amongst parties, and we represent an authenticated value $x \in \mathbb{F}$, where $\mathbb{F} = \mathbb{F}_{2^u}$ and $u|M$, as follows:

$$[\![x]\!] = (\langle x \rangle, \langle \mathbf{m}_x \rangle, \langle \Delta \rangle),$$

where $\mathbf{m}_x = x \cdot \Delta$ is the MAC authenticating x under Δ. We drop the dependence on x in $\mathbf{m}_x$ when it is clear from the context. In particular this notation indicates that each party P_i has a share $x^{(i)}$ of $x \in \mathbb{F}$, a share $\mathbf{m}^{(i)} \in \mathbb{F}_2^M$ of the MAC, and a uniform share $\Delta^{(i)}$ of Δ; hence a $[\![\cdot]\!]$-representation of x implies that x is both *authenticated* with the global key Δ and $\langle \cdot \rangle$-shared, i.e. its value is actually unknown to the parties. Looking ahead, we say that $[\![x]\!]$ is *partially open* if $\langle x \rangle$ is opened, i.e. the parties reveal x, but not the shares of the MAC value $\mathbf{m}$. It is straightforward to see that all the linear operations on $[\![\cdot]\!]$ can be performed locally on the $[\![\cdot]\!]$-sharings. We describe the ideal functionality for generating elements in the $[\![\cdot]\!]$-representation in Fig. 5.

In Section 6 we will see a generalization of this representation for codewords, i.e. we denote an authenticated codeword $C(\mathbf{x})$ by $[\![C(\mathbf{x})]\!]^* = (\langle C(\mathbf{x}) \rangle, \langle \mathbf{m} \rangle, \langle \Delta \rangle)$, where the $*$ is used to denote that the MAC will be "component-wise" on the codeword $C(\mathbf{x})$, i.e. that $\mathbf{m} = C(\mathbf{x}) * \Delta$.

3 OT Extension Protocols

In this section we describe the OT extensions that we use as building blocks for our triple generation protocols. Two of these are standard – a 1-out-of-2 OT functionality and a passively secure correlated OT functionality

Functionality $\mathcal{F}_{\mathsf{COTe}}^{\kappa,\ell}$

The **Initialize** step is independent of inputs and only needs to be called once. After this, **Extend** can be called multiple times. The functionality is parametrized by the number ℓ of resulting OTs and by the bit length κ. Running with parties P_S, P_R and an ideal adversary denoted by $\mathcal{S}$, it operates as follows.

Initialize: Upon receiving $\Delta \in \mathbb{F}_2^\kappa$ from P_S, the functionality stores Δ.
Extend(R,S): Upon receiving $(P_R, (\mathbf{x}_1, \ldots, \mathbf{x}_\ell))$ from P_R, where $\mathbf{x}_h \in \mathbb{F}_2^\kappa$, it does the following:
- It samples $\mathbf{t}_h \in \mathbb{F}_2^\kappa$, $h = 1, \ldots, \ell$, for P_R. If P_R is corrupted then it waits for $\mathcal{S}$ to input $\mathbf{t}_h$.
- It computes $\mathbf{q}_h = \mathbf{t}_h + \mathbf{x}_h * \Delta$, $h = 1, \ldots, \ell$, and sends them to P_S. If P_S is corrupted, the functionality waits for $\mathcal{S}$ to input $\mathbf{q}_h$, and then it outputs to P_R values of $\mathbf{t}_h$ consistent with the adversarial inputs.

Fig. 2. IKNP extension functionality $\mathcal{F}_{\mathsf{COTe}}^{\kappa,\ell}$

– whilst the third protocol is our variant on passively secure correlated OT with privacy amplification, which may be of independent interest for other uses.

We denote by $\mathcal{F}_{\mathsf{OT}}$ the standard $\binom{2}{1}$ OT functionality, where the *sender* P_S inputs two messages $\mathbf{v}_0, \mathbf{v}_1 \in \mathbb{F}_2^\kappa$ and the receiver inputs a choice bit b, and at the end of the protocol the *receiver* P_R learns only the selected message $\mathbf{v}_b$. We use the notation $\mathcal{F}_{\mathsf{OT}}^{\kappa,\ell}$ to denote the functionality that provides ℓ $\binom{2}{1}$ OTs in $\mathbb{F}_2^\kappa$. (see Fig. 20 for a formal definition).Note that $\mathcal{F}_{\mathsf{OT}}^{\kappa,\ell}$ can be implemented very efficiently for any $\ell = \mathsf{poly}(\kappa)$ using just one call to $\mathcal{F}_{\mathsf{OT}}^{\kappa,\kappa}$ and symmetric primitives, for example with actively secure OT extensions [18,1,13].

A slightly different variant of $\mathcal{F}_{\mathsf{OT}}$ is correlated OT, which is a batch of OTs where the sender's messages are correlated, i.e. $\mathbf{v}_0^i + \mathbf{v}_1^i = \Delta$ for some constant Δ, for every pair of messages. We do not use an actively secure correlated OT protocol but a *passively* secure protocol, which is essentially the OT extension of Ishai et al. [12] without the hashing that removes correlation at the end of the protocol. We model this protocol with a functionality that accounts for the deviations an active adversary could make, introducing errors into the output, and call this *correlated OT with errors* (Fig. 2). The implementation of this is exactly the same as the first stage of the IKNP protocol, but for completeness we include the description in Appendix D. The security was proven e.g. by Nielsen [17], where it was referred to as the ABM box.

3.1 Amplified Correlated OT with Errors

Our main new OT extension protocol is a variant of correlated OT that we call *amplified correlated OT with errors*. To best illustrate our use of the protocol, we find it useful to use the concept of a tensor product to describe it. We observe that performing k correlated OTs on k-bit strings between two parties P_R and P_S gives a symmetric protocol: if the input strings of the two parties are $\mathbf{x}$ and $\mathbf{y}$ then the output is given by

$$Q + T = \mathbf{x} \otimes \mathbf{y}$$

where Q and T are the $k \times k$ matrices over $\mathbb{F}_2$ output to each respective party. Thus we view correlated OT as producing a secret sharing of the tensor product of two input vectors. The matrix $\mathbf{x} \otimes \mathbf{y}$ consists of every possible bit product between bits in $\mathbf{x}$ held by P_R and bits in $\mathbf{y}$ held by P_S. We will later use this to compute a secret sharing of the product in an extension field of $\mathbb{F}_2$.

The main difficulty in implementing this with active security is ensuring that a corrupt P_R inputs the same correlation into each OT: if they cheat in just one OT, for example, they can guess P_S's corresponding input bit, resulting in a selective failure attack in a wider protocol. The previous construction used in the TinyOT protocol [18] first employed a consistency check to ensure that P_R used the same correlation on most of the inputs. Since the consistency check cannot completely eliminate cheating, a *privacy amplification* step is then used, which multiplies all of the OTs by a random binary matrix to remove any potential leakage on the sender's input from the few, possibly incorrect OTs.

In our protocol, we choose to omit the consistency check, since the correctness of SPDZ multiplication triples is later checked in the sacrificing procedure. This means that an adversary is able to break the

Functionality $\mathcal{F}_{\mathsf{ACOT}}^{k,s}$

It runs between a sender P_S, a receiver P_R and an ideal adversary $\mathcal{S}$. The procedure can be called repeatedly. Let $\ell' = 2k + s$.

- Upon receiving $\mathbf{x} \in \mathbb{F}_2^k$ from P_R and $\mathbf{y} \in \mathbb{F}_2^k$ from P_S, the functionality does the following:

Honest parties

- The functionality samples a random matrix $Q \in \mathbb{F}_2^{k \times k}$. Then it computes $T = Q + \mathbf{x} \otimes \mathbf{y}$, and it outputs Q to P_S and T to P_R.

Corrupt parties

- If P_S is corrupted, the functionality waits for the adversary to input $Q \in \mathbb{F}_2^{k \times k}$ and one of the following:
  - If the adversary inputs $(\mathsf{MultError}, Y')$ for $Y' \in \mathbb{F}_2^{\ell' \times k}$ such that more than k rows of Y' are non-zero, it samples $\hat{M} \xleftarrow{\$} \mathbb{F}_2^{k \times \ell'}$ and $\hat{\mathbf{x}}' \xleftarrow{\$} \mathbb{F}_2^k$, sets $E = \hat{M} D_{\hat{\mathbf{x}}'} Y'$ and $\hat{\delta} = \hat{M}\hat{\mathbf{x}}' + \mathbf{x}$, and outputs $(\hat{M}, \hat{\delta})$ to $\mathcal{S}$.
  - The adversary inputs $(\mathsf{AddError}, E)$.

  The functionality then computes $T = Q + \mathbf{x} \otimes \mathbf{y} + E$ and sends T to P_R and Q to P_S.
- If P_R is corrupted, it waits for $\mathcal{S}$ to input $\mathbf{x}$. Then it samples $Q \in F_2^{k \times k}$ and computes $U = Q + \mathbf{x} \otimes \mathbf{y}$, outputs U to $\mathcal{S}$, waits for $\mathcal{S}$ to input T, and outputs T to P_R and Q to P_S.

Fig. 3. $\mathcal{F}_{\mathsf{ACOT}}^{k,s}$ – Amplified correlated OT

Protocol $\Pi_{\mathsf{ACOT}}^{k,s}$

Let $\mathbf{x} \in \mathbb{F}_2^k$ and $\mathbf{y} \in \mathbb{F}_2^k$ denote the inputs of P_R and P_S, respectively. Let $\ell' := 2k + s$.

1. Parties run $\mathcal{F}_{\mathsf{OT}}^{k,\ell'}$:
   (a) P_S samples $Q' \xleftarrow{\$} \mathbb{F}_2^{\ell' \times k}$, sets $Y = \mathcal{O} D_{\mathbf{y}}$ where $\mathcal{O} \in \mathbb{F}_2^{\ell' \times k}$ is the matrix full of ones and inputs $(Q', Q' + Y)$.
   (b) P_R samples and inputs $\mathbf{x}' \xleftarrow{\$} \mathbb{F}_2^{\ell'}$.
   (c) P_R receives $T' = Q' + D_{\mathbf{x}'} Y$.
2. Parties sample a random matrix $M \in \mathbb{F}_2^{k \times \ell'}$ using $\mathcal{F}_{\mathsf{Rand}}$ (Fig. 21).
3. P_R sends $\delta = M\mathbf{x}' + \mathbf{x}$ to P_S and outputs $T = MT'$.
4. P_S outputs $Q = MQ' + \delta \otimes \mathbf{y}$.

Fig. 4. Amplified correlated OT

correlation, but the output will be distorted in a way such that sacrificing will fail for all but one possible $\mathbf{x}$ input by P_R. Without amplification, the adversary could craft a situation where the latter check succeeds if, for example, first bit is zero, allowing the selective failure attack. On the other hand, if the success of the adversary depends on guessing k random bits, the probability of a privacy breach is 2^{-k}, which is negligible in k. In the functionality (Fig. 3),the amplification manifests itself in the fact that the environment does not learn $\mathbf{x}'$ which amplifies the error Y'.

The protocol $\Pi_{\mathsf{ACOT}}^{k,s}$ (Fig. 4) requires parties to create the initial correlated OTs on strings of length $\ell' = 2k + s$, where s is the statistical security parameter. The sender P_S is then allowed to input a $\ell' \times k$ matrix Y instead of a vector $\mathbf{y}$, whilst the receiver chooses a random string $\mathbf{x}' \in \mathbb{F}_2^{\ell'}$. $\mathcal{F}_{\mathsf{OT}}$ then produces a sharing of $D_{\mathbf{x}'} Y$, instead of $\mathbf{x}' \otimes \mathbf{y}$ in the honest case. For the privacy amplification, a random $k \times \ell'$ binary matrix M is chosen, and everything is multiplied by this to give outputs of length k as required. Finally, P_R sends $M\mathbf{x}' + \mathbf{x}$ to switch to their real input $\mathbf{x}$. Multiplying by M ensures that even if P_S learns a few bits of $\mathbf{x}'$, all of $\mathbf{x}$ remains secure as every bit of $\mathbf{x}'$ is combined into every bit of the output.

Lemma 1. *The protocol $\Pi_{\mathsf{ACOT}}^{k,s}$ (Fig. 4) implements the functionality $\mathcal{F}_{\mathsf{ACOT}}^{k,s}$ (Fig. 3)in the $\mathcal{F}_{\mathsf{OT}}^{k,\ell'}$-hybrid model with statistical security s.*

Proof. The proof essentially involves checking that $Q + T = \mathbf{x} \otimes \mathbf{y}$ for honest parties, that at most k deviations by P_S are canceled by M with overwhelming probability, and that more than k deviations cause

Functionality $\mathcal{F}^{\mathbb{F}}_{[\![\cdot]\!]}$

Let $\mathbb{F} = \mathbb{F}_{2^M}$, with $M \geq \kappa$. Let A be the indices of corrupt parties. Running with parties $P_1, \ldots, P_n$ and an ideal adversary $\mathcal{S}$, the functionality authenticates values in $\mathbb{F}_{2^u}$ for $u|M$.

Initialize: On input (Init) the functionality activates and waits for the adversary to input a set of shares $\{\Delta^{(j)}\}_{j\in A}$ in $\mathbb{F}$. It samples random $\{\Delta^{(i)}\}_{i\notin A}$ in $\mathbb{F}$ for the honest parties, defining $\Delta := \sum_{i\in[n]} \Delta^{(i)}$. If any $j \in A$ outputs Abort then the functionality aborts.

n-Share: On input $(\mathsf{Authenticate}, \mathbf{x}_1^{(i)}, \ldots, \mathbf{x}_\ell^{(i)})$ from the honest parties and the adversary where $\mathbf{x}_h^{(i)} \in \mathbb{F}_{2^u}$, the functionality proceeds as follows.

Honest parties: $\forall h \in [\ell]$, it computes $\mathbf{x}_h = \sum_{i\in\mathcal{P}} \mathbf{x}_h^{(i)}$ and $\mathbf{m}_h = \mathbf{x}_h \cdot \Delta$. [a] Then it creates a sharing $\langle \mathbf{m}_h \rangle = \{\mathbf{m}_h^{(1)}, \ldots, \mathbf{m}_h^{(n)}\}$ and outputs $\mathbf{m}_h^{(i)}$ to P_i for each $i \in \mathcal{P}, h \in [\ell]$.

Corrupted parties: The functionality waits for the adversary $\mathcal{S}$ to input the set A of corrupted parties. Then it proceeds as follows:

- $\forall h \in [\ell]$, the functionality waits for $\mathcal{S}$ to input shares $\{\mathbf{m}_h^{(j)}\}_{j\in A}$ and it generates $\langle \mathbf{m}_h \rangle$, with honest shares $\{\mathbf{m}_h^{(i)}\}_{i\notin A, h\in[\ell]}$, consistent with adversarial shares but otherwise random.
- If the adversary inputs $(\mathsf{Error}, \{e_{h,j}^{(k)}\}_{k\notin A, h\in[\ell], j\in[M]})$ with elements in $\mathbb{F}_{2^M}$, the functionality sets $\mathbf{m}_h^{(k)} = \mathbf{m}_h^{(k)} + \sum_{j=1}^{M} e_{h,j}^{(k)} \cdot \Delta_j^{(k)} \cdot X^{j-1}$ where $\Delta_j^{(k)}$ denotes the j-th bit of $\Delta^{(k)}$.
- For each $k \notin A$, the functionality outputs $\{\mathbf{m}_h^{(k)}\}$ to P_k.

Key queries: On input of a description of an affine subspace $S \subset (\mathbb{F}_2^M)^n$, return Success if $(\Delta^{(1)}, \ldots, \Delta^{(n)}) \in S$. Otherwise return Abort.

[a] If $u \neq M$ we view Δ as an element of $\mathbb{F}_{2^u}^{M/u}$ and perform the multiplication by $\mathbf{x}_h$ componentwise.

Fig. 5. Ideal Generation of $[\![\cdot]\!]$-representations

the desired entropy in the output. The two cases are modeled by two different possible adversarial inputs to the functionality. See Appendix F.1 for further details.

4 Authentication Protocol

In this section we describe our protocol to authenticate secret shared values over characteristic two finite fields, using correlated OT extension. The resulting MACs, and the relative MAC keys, are always elements of a finite field $\mathbb{F} := \mathbb{F}_{2^M}$, where $M \geq \kappa$ and κ is a computational security parameter, whilst the secret values may lie in $\mathbb{F}_{2^u}$ for any $u|M$. We then view the global MAC key as an element of $\mathbb{F}_{2^u}^{M/u}$ and the MAC multiplicative relation as componentwise multiplication in this ring. Our authentication method is similar to that by Larraia et al. [16] (with modifications to avoid the selective failure attack) but here we only use a passively secure correlated OT functionality ($\mathcal{F}_{\mathsf{COTe}}$), allowing an adversary to introduce errors in the MACs that depend on arbitrary bits of other parties' MAC key shares. When combined with the MAC check protocol by Damgård et al. [8] (shown in Fig. 16), this turns out to be sufficient for our purposes, avoiding the need for additional consistency checks in the OTs.

Our authentication protocol in Fig. 6 begins with an Initialize stage, which initializes a $\mathcal{F}_{\mathsf{COTe}}$ instance between every pair of parties (P_i, P_j), where P_j inputs their MAC key share $\Delta^{(j)}$. This introduces the subtle issue that a corrupt P_j may initialize $\mathcal{F}_{\mathsf{COTe}}$ with two different MAC shares for P_{i_1} and P_{i_2}, say $\Delta^{(j)}$ and $\hat{\Delta}^{(j)}$, which allows for the selective failure attack mentioned earlier – if P_{i_2} authenticates a bit b, the MAC check will still pass if $b = 0$, despite being authenticated under the wrong key. However, since $\mathcal{F}_{\mathsf{COTe}}$.Initialize is only called once, the MAC key shares are fixed for the entire protocol, so it is clear that P_j could not remain undetected if enough random values are authenticated and checked. To ensure this in our protocol in Fig. 6, we add a consistency check to the Initialize stage, where κ dummy values are authenticated, then opened and checked. If the check passes then every party's MAC key has been initialized correctly, except with probability $2^{-\kappa}$. Although in practice this overhead is not needed when authenticating $\ell \geq \kappa$ values,

modeling this would introduce additional errors into the functionality and make the analysis of the triple generation protocols more complex.

Now we present the protocol $\Pi_{[\![\cdot]\!]}$, realizing the ideal functionality of Fig. 5, in more detail. We describe the authentication procedure for bits first and then the extension to $\mathbb{F}_{2^u}$.

Suppose parties need to authenticate an additively secret shared random bit $x = x^{(1)} + \cdots + x^{(n)}$. Once the global key Δ is initialized, the parties call the subprotocol $\Pi_{[\cdot]}$ (Fig. 7) n times. Output of each of these calls is a value $\mathbf{u}^{(i)}$ for P_i and values $\mathbf{q}^{(j,i)}$ for each P_j, $j \neq i$, such that

$$\mathbf{u}^{(i)} + \mathbf{q}^{(j,i)} = \sum_{j \neq i} \mathbf{t}^{(i,j)} + x^{(i)} \cdot \Delta^{(i)} + \sum_{j \neq i} \mathbf{q}^{(j,i)} = x^{(i)} \cdot \Delta. \tag{1}$$

To create a complete authentication $[\![x]\!]$, each party sets $\mathbf{m}^{(i)} = \mathbf{u}^{(i)} + \sum_{j \neq i} \mathbf{q}^{(i,j)}$. Notice that if we add up all the MAC shares, we obtain:

$$\mathbf{m} = \sum_{i \in \mathcal{P}} \mathbf{m}^{(i)} = \sum_{i \in \mathcal{P}} \big(\mathbf{u}^{(i)} + \sum_{j \neq i} \mathbf{q}^{(i,j)}\big) = \sum_{i \in \mathcal{P}} \big(\mathbf{u}^{(i)} + \sum_{j \neq i} \mathbf{q}^{(j,i)}\big) = \sum_{i \in \mathcal{P}} x^{(i)} \cdot \Delta = x \cdot \Delta,$$

where the second equality holds for the symmetry of the notation $\mathbf{q}^{(i,j)}$ and the third follows from (1).

Finally, if P_i wants to authenticate a bit $x^{(i)}$, it is enough, from Equation (1), setting $\mathbf{m}^{(i)} = \mathbf{u}^{(i)}$ and $\mathbf{m}^{(j)} = \mathbf{q}^{(j,i)}$, $\forall j \neq i$. Clearly, from (1), we have $\sum_{i \in \mathcal{P}} \mathbf{m}^{(i)} = x^{(i)} \cdot \Delta$.

Consider now the case where parties need to authenticate elements in $\mathbb{F}_{2^u}$. We can represent any element $\mathbf{x} \in \mathbb{F}_{2^u}$ as a binary vector $(x_1, \ldots, x_u) \in \mathbb{F}_2^u$. In order to obtain a representation $[\![\mathbf{x}]\!]$ it is sufficient to repeat the previous procedure u times to get $[\![x_i]\!]$ and then compute $[\![\mathbf{x}]\!]$ as $\sum_{k=1}^{u} [\![x_k]\!] \cdot X^{k-1}$ (Fig. 6). Here we let X denote the variable in polynomial representation of $\mathbb{F}_{2^u}$ and $[\![x_k]\!]$ the k'th coefficient.

We now describe what happens to the MAC representation in presence of corrupted parties. As we have already pointed out before, a corrupt party could input different MAC key shares when initializing $\mathcal{F}_{\mathsf{COTe}}$ with different parties. Moreover a corrupt P_i could input vectors $\mathbf{x}_1^{(i)}, \ldots \mathbf{x}_\ell^{(i)}$ instead of bits to n-**Share**(i) (i.e. to $\mathcal{F}_{\mathsf{COTe}}$). This will produce an error in the authentication depending on the MAC key. Putting things together we obtain the following faulty representation:

$$\mathbf{m} = x \cdot \Delta + \sum_{k \notin A} x^{(k)} \cdot \delta^{(i)} + \sum_{k \notin A} \mathbf{e}^{(i,k)} * \Delta^{(k)}, \qquad \text{for some } i \in A$$

where A is the set of corrupt parties, $\delta^{(i)}$ is an offset vector known to the adversary which represents the possibility that corrupted parties input different MAC key shares, whilst $\mathbf{e}^{(i,k)}$ depends on the adversary inputting vectors and not just bits to $\mathcal{F}_{\mathsf{COTe}}$. More precisely, if P_i inputs a vector $\mathbf{x}^{(i)}$ to n-**Share**(i), we can rewrite it as $\mathbf{x}^{(i)} = x^{(i)} \cdot \mathbf{1} + \mathbf{e}^{(i,k)}$, where $\mathbf{e}^{(i,k)} \in \mathbb{F}_2^M$ is an error vector known to the adversary. While we prevent the first type of errors by adding a $\mathsf{MACCheck}$ step in the Initialize phase, we allow the second type of corruption. This faulty authentication suffices for our purposes due to the MAC checking procedure used later on.

Lemma 2. *In the $\mathcal{F}_{\mathsf{COTe}}^{\kappa,\ell}$-hybrid model, the protocol $\Pi_{[\![\cdot]\!]}$ (Fig. 6) implements $\mathcal{F}_{[\![\cdot]\!]}$ against any static adversary corrupting up to $n-1$ parties.*

Proof. See Appendix F.2.

Protocol $\Pi_{[\![\cdot]\!]}$

The protocol computes an authenticated sharing, $[\![x]\!]$, of an additively shared value $x \in \mathbb{F}_{2^u}$, where $u|M$ and the resulting MAC lies in $\mathbb{F}_{2^M}$.

Initialize: Each party P_i samples $\Delta^{(i)} \in \mathbb{F}_{2^M}$. Each pair of parties (P_i, P_j) (for $i \neq j$) calls $\mathcal{F}_{\mathsf{COTe}}^{M,\ell}$.Initialize where P_j inputs $\Delta^{(j)}$. Now check consistency of the MAC keys:

1. The parties run (as a subroutine) n-**Share** κ times, where each party P_i inputs random shares $x_1^{(i)}, \ldots, x_\kappa^{(i)} \in \mathbb{F}_2$, to obtain $[\![x_1]\!], \ldots, [\![x_\kappa]\!]$.
2. Each party P_i broadcasts their shares $x_1^{(i)}, \ldots, x_\kappa^{(i)}$ and computes $x_h = \sum_{i=1}^n x_h^{(i)}$ for $h = 1, \ldots, \kappa$.
3. The parties run Π_{MACCheck}, inputting $x_1, \ldots, x_\kappa$ and their shares of the corresponding MACs.
4. If Π_{MACCheck} fails, output Abort.

n-**Share:** Each party P_i inputs a share. We do one of the following, depending on the type of data being authenticated:

$\mathbb{F}_2$: P_i inputs $x_h^{(i)} \in \mathbb{F}_2$, $h \in [\ell]$.

1. For each $i = 1, \ldots, n$, run the subprotocol the $\Pi_{[\cdot]}$ with input $(i, (x_1^{(i)}, \ldots, x_\ell^{(i)}), \mathsf{Share})$ from P_i and (i, Share) from P_j for $j \neq i$, to obtain $[x^{(1)}]_\Delta^1, \ldots, [x^{(n)}]_\Delta^n$.
2. Each party P_i locally computes $[\![x]\!] = \sum_i [x^{(i)}]_\Delta^i$.

$\mathbb{F}_{2^u}$: P_i inputs $\mathbf{x}^{(i)} \in \mathbb{F}_{2^u}$.

1. For each $i = 1, \ldots, n$, run the subprotocol $\Pi_{[\cdot]}$ with input $(i, \mathbf{x}_1^{(i)}, \ldots, \mathbf{x}_u^{(i)}, \mathsf{Share})$ from P_i and (i, Share) from P_j for $j \neq i$, and then sum up the n MACs for each component $[\![\mathbf{x}_1]\!], \ldots, [\![\mathbf{x}_u]\!]$.
2. Compute $[\![\mathbf{x}]\!] = \sum_{j=1}^u [\![x_j]\!] \cdot X^{j-1}$.

Fig. 6. From $[\cdot]_\Delta^i$ to $[\![\cdot]\!]$

Subprotocol $\Pi_{[\cdot]}$

Share(i): On input $(i, (x_1^{(i)}, \ldots, x_\ell^{(i)}), \mathsf{Share})$ from P_i, and (i, Share) from all other parties, do:

1. For each $j \neq i$, P_i and P_j call $\mathcal{F}_{\mathsf{COTe}}^{M,\ell}$.Extend on inputs $\mathbf{x}^{(i)} = (x_1^{(i)}, \ldots, x_\ell^{(i)})$.
   P_i obtains $\{\mathbf{t}_h^{(i,j)}\}_{h \in [\ell]}$, while P_j obtains $\{\mathbf{q}_h^{(j,i)}\}_{h \in [\ell]}$ such that $\mathbf{q}_h^{(j,i)} = \mathbf{t}_h^{(i,j)} + x_h^{(i)} \cdot \Delta_h^{(j)}$.
2. P_i outputs $\mathbf{u}_h^{(i)} := x_h^{(i)} \cdot \Delta^{(i)} + \sum_{j \neq i} \mathbf{t}_h^{(i,j)}$ and P_j outputs $\mathbf{q}_h^{(j,i)}$ for $j \neq i$ and $h = 1, \ldots, \ell$. The system now has $[x_h^{(i)}]_\Delta^i$, $h = 1, \ldots, \ell$.

Fig. 7. Transforming two-party representations onto $[\cdot]_{\Delta,\mathcal{P}}^i$-representation

5 Triple Generation in $\mathbb{F}_2$ and $\mathbb{F}_{2^k}$

In this section we describe our protocols generating triples in finite fields. First we describe the protocols for multiplication triples in $\mathbb{F}_{2^\kappa}$ (Fig. 10 and 14), and then the protocol for bit triples (Fig. 15). Both approaches implement the functionality $\mathcal{F}_{\mathsf{Triples}}^{\mathbb{F}}$, given in Fig. 8. Note that the functionality allows an adversary to try and guess an affine subspace containing the parties' MAC key shares, which is required because of our faulty authentication procedure described in the previous section.

5.1 $\mathbb{F}_{2^k}$ Triples

In this section, we show how to generate $\mathbb{F}_{2^k}$ authenticated triples using $\mathcal{F}_{\mathsf{GFMult}}^{k,s}$ (Fig. 24) and $\mathcal{F}_{[\![\cdot]\!]}^{\mathbb{F}_{2^k}}$. We realize the functionality $\mathcal{F}_{\mathsf{GFMult}}^{k,s}$ with protocol $\Pi_{\mathsf{GFMult}}^{k,s}$ (Fig. 9). This protocol is a simple extension of $\mathcal{F}_{\mathsf{ACOT}}$ that converts the sharing of a tensor product matrix in $\mathbb{F}_2^{k \times k}$ to the sharing of a product in $\mathbb{F}_{2^k}$. Taking this modular approach simplifies the proof for triple generation, as we can deal with the complex errors from $\mathcal{F}_{\mathsf{ACOT}}$ separately. Our first triple generation protocol ($\Pi_{\mathsf{UncheckedTriples}}$) will not reveal any information about

Functionality $\mathcal{F}^{\mathbb{F}}_{\mathsf{Triples}}$

Let A be the indices of corrupt parties. Running with parties $P_1, \ldots, P_n$ and an adversary $\mathcal{S}$, the functionality operates as follows.

Initialize: On input (Init) the functionality activates and waits for $\mathcal{S}$ to input a set of shares $\{\Delta^{(j)}\}_{j\in A}$. It samples random $\{\Delta^{(i)}\}_{i\notin A}$ in $\mathbb{F}_2^\kappa$ for the honest parties, defining $\Delta := \sum_{i\in[n]} \Delta^{(i)}$. If any $j \in A$ outputs Abort then the functionality aborts.

Honest Parties: On input (Triples), the functionality outputs random $[\![x_h]\!]_\Delta, [\![y_h]\!]_\Delta, [\![z_h]\!]_\Delta$, such that $\langle z_h\rangle = \langle x_h\rangle \cdot \langle y_h\rangle$ and $z_h, y_h, x_h \in \mathbb{F}$.

Corrupted Parties: The functionality samples $x_h, y_h \xleftarrow{\$} \mathbb{F}$ and computes $z_h = x_h \cdot y_h$. To produce $[\![a]\!]_\Delta = (\langle a\rangle, \langle \mathbf{m}\rangle, \langle \Delta\rangle)$, where $a \in \{x_h, y_h, z_h\}_{h\in[\ell]}$ it does the following:

- It waits the adversary to input shares $\{a^{(i)}\}_{i\in A}$ and $\{\mathbf{m}^{(i)}\}_{i\in A}$.
- It waits for the adversary to input (ValueError, e) and (MacError, $\mathbf{e}$).
- It selects the shares of honest parties at random, but consistent with adversarial shares and with $a + e$ and $a \cdot \Delta + \mathbf{e}$, that is, such that $\sum_{i=1}^n a^{(i)} = a + e$ and $\sum_{i=1}^n \mathbf{m}^{(i)} = a \cdot \Delta + \mathbf{e}$.

Key queries: On input of a description of an affine subspace $S \subset (\mathbb{F}_2^\kappa)^n$, return Success if $(\Delta^{(1)}, \ldots, \Delta^{(n)}) \in S$. Otherwise return Abort.

Fig. 8. Ideal functionality for triples generation

Protocol $\Pi^{k,s}_{\mathsf{GFMult}}$

Let $\mathbf{x}$ and $\mathbf{y}$ denote the inputs of P_R and P_S respectively, in $\mathbb{F}_{2^k}$, and let s be a statistical security parameter. Furthermore, let $\mathbf{e} = (1, X, \ldots, X^{k-1})$ and $\ell' = 2k + s$.

1. The parties run $\mathcal{F}^{k,s}_{\mathsf{ACOT}}$:
   (a) P_R inputs $\mathbf{x}$ and P_S inputs $\mathbf{y}$.
   (b) P_R receives T and P_S receives Q such that $T + Q = \mathbf{x} \otimes \mathbf{y}$.
2. P_R outputs $\mathbf{t} = \mathbf{e}T\mathbf{e}^\top$ and P_S outputs $\mathbf{q} = \mathbf{e}Q\mathbf{e}^\top$.

Fig. 9. $\mathbb{F}_{2^k}$ multiplication

Protocol $\Pi_{\mathsf{UncheckedTriples}}$

Initialize: The parties initialize $\mathcal{F}^{\mathbb{F}_{2^k}}_{[\![\cdot]\!]}$, which outputs $\Delta^{(i)}$ to party i.
Triple generation:

1. Every party i samples random $\mathbf{a}^{(i)} \xleftarrow{\$} \mathbb{F}_{2^k}$ and $\mathbf{b}^{(i)} \xleftarrow{\$} \mathbb{F}_{2^k}$.
2. Every tuple of parties $(i, j) \in [n]^2, i \neq j$ call $\mathcal{F}^{k,s}_{\mathsf{GFMult}}$ with P_i inputting $\mathbf{a}^{(i)}$ and P_j inputting $\mathbf{b}^{(j)}$ to generate a random secret sharing $\mathbf{c}^{(i,j)}_{i,j} + \mathbf{c}^{(j,i)}_{i,j} = \mathbf{a}^{(i)} \cdot \mathbf{b}^{(j)}$.
3. Every party i computes $\mathbf{c}^{(i)} = \mathbf{a}^{(i)} \cdot \mathbf{b}^{(i)} + \sum_{j\neq i}(\mathbf{c}^{(i,j)}_{i,j} + \mathbf{c}^{(i,j)}_{j,i})$.
4. Party i calls $\mathcal{F}^{\mathbb{F}_{2^k}}_{[\![\cdot]\!]}$ with inputs $\mathbf{a}^{(i)}$, $\mathbf{b}^{(i)}$, and $\mathbf{c}^{(i)}$, and receives $\mathbf{m}^{(i)}_{\mathbf{a}}$, $\mathbf{m}^{(i)}_{\mathbf{b}}$, and $\mathbf{m}^{(i)}_{\mathbf{c}}$.

Fig. 10. Protocol for generation of unchecked $\mathbb{F}_{2^k}$ triples.

the values or the authentication key, but an active adversary can distort the output in various ways. We then present a protocol ($\Pi_{\mathsf{TripleCheck}}$) to check the generated triples from $\Pi_{\mathsf{UncheckedTriples}}$, similarly to the sacrificing step of the SPDZ protocol [8], to ensure that an adversary has not distorted them.

The protocol is somewhat similar to the one in the previous section. Instead of using $n(n-1)$ instances of $\mathcal{F}_{\mathsf{COTe}}$, it uses $n(n-1)$ instances of $\mathcal{F}^{k,s}_{\mathsf{GFMult}}$, which is necessary to compute a secret sharing of $\mathbf{x} \cdot \mathbf{y}$, where $\mathbf{x}$ and $\mathbf{y}$ are known to different parties.

$\mathcal{F}_{\mathsf{UncheckedTriples}}$

Let B denote the set of honest parties, and let $\hat{\imath}$ be the lowest index in B. Furthermore, let $B' = B \setminus \{\hat{\imath}\}$ and $A = [n] \setminus B$ the set of corrupted parties.

Initialize: Sample $\langle \Delta \rangle \xleftarrow{\$} \mathbb{F}_{2^k}$ and secret share it using shares for corrupted parties input by $\mathcal{S}$. Output the share $\Delta^{(i)}$ to party i.

Triple generation:

1. Sample $(\langle \mathbf{a} \rangle, \langle \mathbf{b} \rangle) \xleftarrow{\$} \mathbb{F}_{2^k}$ according to corrupted parties' shares from $\mathcal{S}$.
2. Wait for $\mathcal{S}$ to input $\{\mathbf{f}_{\mathbf{a}}^{(i)}, \mathbf{f}_{\mathbf{b}}^{(i)}\}_{i \in B'}$, $\{\varphi_{\mathbf{a},h}^{(i)}, \varphi_{\mathbf{b},h}^{(i)}, \varphi_{\mathbf{c},h}^{(i)}\}_{i \in B, h \in [k]}$, $\bar{\mathbf{f}}, \psi_{\mathbf{a}}, \psi_{\mathbf{b}}, \psi_{\mathbf{c}} \in \mathbb{F}_{2^k}$, $C \subset B \times A$, and $\{(Y^{(i,j)}, \bar{\mathbf{f}}^{(i,j)})\}_{(i,j) \in C}$.
3. For all $(i,j) \in C$, compute $(\bar{M}^{(i,j)}, \delta^{(i,j)}, \mathbf{f}^{(i,j)})$ as $\mathcal{F}_{\mathsf{GFMult}}$ would (using $\mathbf{a}^{(i)}$ for $\mathbf{x}$), and send $\{(\bar{M}^{(i,j)}, \delta^{(i,j)})\}_{(i,j) \in C}$ to $\mathcal{S}$.
4. Compute

$$\mathbf{f} = \sum_{(i,j) \in C} \mathbf{f}^{(i,j)} + \bar{\mathbf{f}}$$

$$\mathbf{c} = \mathbf{a} \cdot \mathbf{b} + \sum_{i \in B'} \mathbf{a}^{(i)} \cdot \mathbf{f}_{\mathbf{a}}^{(i)} + \sum_{i \in B'} \mathbf{b}^{(i)} \cdot \mathbf{f}_{\mathbf{b}}^{(i)} + \mathbf{f}$$

$$\mathbf{m}_{\mathbf{a}} = \mathbf{a} \cdot \Delta + \sum_{i \in B, h \in [k]} \Delta_h^{(i)} \cdot \varphi_{\mathbf{a},h}^{(i)} \cdot X^h + \psi_{\mathbf{a}}$$

$$\mathbf{m}_{\mathbf{b}} = \mathbf{b} \cdot \Delta + \sum_{i \in B, h \in [k]} \Delta_h^{(i)} \cdot \varphi_{\mathbf{b},h}^{(i)} \cdot X^h + \psi_{\mathbf{b}}$$

$$\mathbf{m}_{\mathbf{c}} = \mathbf{c} \cdot \Delta + \sum_{i \in B, h \in [k]} \Delta_h^{(i)} \cdot \varphi_{\mathbf{c},h}^{(i)} \cdot X^h + \psi_{\mathbf{c}}$$

   and secret share them using shares input by $\mathcal{S}$ for corrupted parties. $\Delta_h^{(i)}$ denotes the h-th bit of the share $\Delta^{(i)}$.
5. Output $(\mathbf{a}^{(i)}, \mathbf{b}^{(i)}, \mathbf{c}^{(i)}, \mathbf{m}_{\mathbf{a}}^{(i)}, \mathbf{m}_{\mathbf{b}}^{(i)}, \mathbf{m}_{\mathbf{c}}^{(i)})$ to party i.

Fig. 11. Unchecked $\mathbb{F}_{2^k}$ triple generation.

Lemma 3. *The protocol $\Pi_{\mathsf{UncheckedTriples}}$ shown in Fig. 10 implements the functionality $\mathcal{F}_{\mathsf{UncheckedTriples}}$ in the $(\mathcal{F}_{\mathsf{GFMult}}^{k,s}, \mathcal{F}_{[\![\cdot]\!]}^{\mathbb{F}_{2^k}})$-hybrid model with perfect security.*

Proof. The proof is straightforward using an appropriate simulator. See Appendix F.6 for further details.

The protocol $\Pi_{\mathsf{TripleCheck}}$ produces N triples using $2N$ unchecked triples similar to the sacrificing step of the SPDZ protocol. However, corrupted parties have more options to deviate here, which we counter by using more random coefficients for checking. Recall that, in the SPDZ protocol, parties input their random shares by broadcasting a homomorphic encryption thereof. Here, the parties have to input such a share by using an instance of $\mathcal{F}_{\mathsf{GFMult}}^{k,s}$ and $\mathcal{F}_{[\![\cdot]\!]}^{\mathbb{F}_{2^k}}$ with every other party, which opens up the possibility of using a different value in every instance. We will prove that, if the check passes, the parties have used consistent inputs to $\mathcal{F}_{\mathsf{GFMult}}^{k,s}$. On the other hand, $\mathcal{F}_{[\![\cdot]\!]}^{\mathbb{F}_{2^k}}$ provides less security guarantees. However, we will also prove that the more deviation there is with $\mathcal{F}_{[\![\cdot]\!]}^{\mathbb{F}_{2^k}}$, the more likely the check is to fail. This is modeled using the key query access of $\mathcal{F}_{\mathsf{Triples}}$. Note that, while this reveals some information about the MAC key Δ, this does not contradict the security of the resulting MPC protocol because Δ does not protect any private information. Furthermore, breaking correctness corresponds to guessing Δ, which will only succeed with probability negligible in k because incorrect guesses lead to an abort.

We use a supplemental functionality $\mathcal{F}_{\mathsf{BatchCheck}}$ (Fig. 12), which is used to check that a batch of shared values are equal to zero, and can be easily implemented using commitment and $\mathcal{F}_{\mathsf{Rand}}$ (see Fig. 13 and Appendix C). The first use of $\mathcal{F}_{\mathsf{BatchCheck}}$ corresponds to using the SPDZ MAC check protocol for $\mathbf{r}_j$ and

$\mathcal{F}_{\mathsf{BatchCheck}}$

Let A denote the set of corrupted parties, and $B = [n] \setminus A$ the set of honest ones.

Batch Checking:

1. Sample $\{\chi^{(j)}\}_{j\in[N]} \xleftarrow{\$} \mathbb{F}^N$.
2. Each party i inputs $\{\mathbf{z}_i^{(j)}\}_{j\in[N]}$.
3. For every party i, compute $\sigma_i = \sum_{i=1}^{N} \chi^{(j)} \cdot \mathbf{z}_i^{(j)}$.
4. Output $\{\chi^{(j)}\}_{j\in[N]}$ to the adversary.
5. Wait for the adversary to input ζ.
6. Output $\{\sigma_i\}_{i\in B}$ to the adversary.
7. If $\sum_{i\in B} \sigma_i = \zeta$, output OK, otherwise output $\perp$.

Fig. 12. Batch checking functionality.

Protocol $\Pi_{\mathsf{BatchCheck}}$

1. The parties invoke $\mathcal{F}_{\mathsf{Rand}}$ to receive $\{\chi^{(j)}\}_{j\in N} \in \mathbb{F}_{2^k}^N$.
2. Every party i computes $\sigma_i = \sum_{j=1}^{N} \chi^{(j)} \cdot \mathbf{z}_i^{(j)}$.
3. Every party i calls $\mathsf{Comm}(\sigma_i)$ on $\mathcal{F}_{\mathsf{Comm}}$, which broadcasts τ_i.
4. Every party i calls $\mathsf{Open}(\tau_i)$ on $\mathcal{F}_{\mathsf{Comm}}$, which broadcasts σ_j for all j.
5. If $\sigma_1 + \cdots + \sigma_n \neq 0$, the parties output $\perp$ and abort, otherwise they output OK.

Fig. 13. Batch Zero Checking Protocol

Protocol $\Pi_{\mathsf{TripleCheck}}$

Initialize: Each party receives $\Delta^{(i)}$ from $\mathcal{F}_{\mathsf{UncheckedTriples}}$.
Triple Generation:

1. Generate $2N$ $\{[\![\mathbf{a}_j]\!], [\![\mathbf{b}_j]\!], [\![\mathbf{c}_j]\!]\}_{j\in[2N]}$ unchecked triples using $\mathcal{F}_{\mathsf{UncheckedTriples}}$.
2. Sample $\mathbf{t}, \mathbf{t}', \mathbf{t}'' \xleftarrow{\$} \mathbb{F}_{2^k}$ using $\mathcal{F}_{\mathsf{Rand}}$.
3. For all $j \in [N]$, open $\mathbf{t} \cdot \langle \mathbf{b}_j \rangle + \mathbf{t}' \cdot \langle \mathbf{b}_{j+N} \rangle$ as $\mathbf{r}_j$ and $\mathbf{t}' \cdot \langle \mathbf{a}_j \rangle + \mathbf{t}'' \cdot \langle \mathbf{a}_{j+N} \rangle$ as $\mathbf{s}_j$.
4. Use $\mathcal{F}_{\mathsf{BatchCheck}}$ with $\{\mathbf{r}_j \cdot \langle \Delta \rangle + \mathbf{t} \cdot \langle \mathbf{m}_{\mathbf{b}_j} \rangle + \mathbf{t}' \cdot \langle \mathbf{m}_{\mathbf{b}_{j+N)}} \rangle\}_{j\in[N]}$ and $\{\mathbf{s}_j \cdot \langle \Delta \rangle + \mathbf{t}' \cdot \langle \mathbf{m}_{\mathbf{a}_j} \rangle + \mathbf{t}'' \cdot \langle \mathbf{m}_{\mathbf{a}_{j+N}} \rangle\}_{j\in[N]}$, and abort if it returns $\perp$.
5. Use $\mathcal{F}_{\mathsf{BatchCheck}}$ with $\{\mathbf{t} \cdot \langle \mathbf{m}_{\mathbf{c}_j} \rangle + \mathbf{t}'' \cdot \langle \mathbf{m}_{\mathbf{c}_{j+N}} \rangle + \mathbf{r}_j \cdot \langle \mathbf{m}_{\mathbf{a}_j} \rangle + \mathbf{s}_j \cdot \langle \mathbf{m}_{\mathbf{b}_{j+N}} \rangle\}_{j\in[N]}$, and abort if returns $\perp$.
6. Output $\{[\![\mathbf{a}_j]\!], [\![\mathbf{b}_j]\!], [\![\mathbf{c}_j]\!]\}_{j\in[N]}$.

Fig. 14. Triple checking protocol.

$\mathbf{s}_j$ for all $j \in [N]$, and the second use corresponds to the sacrificing step, which checks whether $\mathbf{t} \cdot \mathbf{c}_j + \mathbf{t}'' \cdot \mathbf{c}_{j+N} + \mathbf{r}_j \mathbf{a}_j + \mathbf{s}_j \cdot \mathbf{b}_{j+N} = 0$ for all $j \in [N]$.

Theorem 1. *The protocol $\Pi_{\mathsf{TripleCheck}}$, described in Fig. 14, implements $\mathcal{F}_{\mathsf{Triples}}$ in the $(\mathcal{F}_{\mathsf{UncheckedTriples}}, \mathcal{F}_{\mathsf{Rand}})$-hybrid model with statistical security $(k-4)$.*

Proof. The proof mainly consists of proving that, if $\mathbf{c}_j \neq \mathbf{a}_j \cdot \mathbf{b}_j$ or the MAC values are incorrect for some j, and the check passes, then the adversary can compute the offset of $\mathbf{c}_j$ or the MAC values. See Appendix F.7.

5.2 $\mathbb{F}_2$ Triples

This section shows how to produce a large number ℓ of random, authenticated bit triples using the correlated OT with errors functionality $\mathcal{F}_{\mathsf{COTe}}$ from Section 3. We describe the main steps of the protocol in Fig. 15. The main difference with respect to the protocol by Larraia et al. [16] is that here we use the outputs of $\mathcal{F}_{\mathsf{COTe}}$

Protocol $\Pi_{\mathsf{BitTriples}}$

The goal of the protocol is to generate ℓ $\mathbb{F}_2$ triples $\langle x_h\rangle$, $\langle y_h\rangle$, $\langle z_h\rangle$, $h = 1, \ldots, \ell$, such that $z_h = x_h \cdot y_h$, together with $[\![x_h]\!], [\![y_h]\!], [\![z_h]\!]$. The protocol is parametrized by the number ℓ of authenticated triples, and it assumes access to a random oracle $H : \{0,1\}^* \to \{0,1\}$.

Initialize:

1. Each party P_i samples a random MAC key share $\Delta^{(i)}$, a second value $\tilde{\Delta}^{(i)} \in \mathbb{F}_2^\kappa$ and sets $\hat{\Delta}^{(i)} = (\tilde{\Delta}^{(i)} \| \Delta^{(i)}) \in \mathbb{F}_2^{2\kappa}$.
2. Each pair of parties (P_i, P_j) (for $i \neq j$) calls $\mathcal{F}_{\mathsf{COTe}}.\mathsf{Initialize}$, where P_j inputs $\hat{\Delta}^{(j)}$, and $\mathcal{F}_{[\![\cdot]\!]}.\mathsf{Init}$, where P_j inputs $\Delta^{(j)}$.
3. Parties check consistency of the $\mathcal{F}_{\mathsf{COTe}}$ inputs $\hat{\Delta} = \hat{\Delta}^{(1)} + \cdots + \hat{\Delta}^{(n)}$ as in the Initialize step of $\Pi_{[\![\cdot]\!]}$, using κ random values. If Π_{MACCheck} fails, output Abort.

COTe.Extend: Each P_i, $i \in \mathcal{P}$, runs $\mathcal{F}_{\mathsf{COTe}}.\mathsf{Extend}$ with P_j, $\forall j \neq i$: P_i inputs $\mathbf{x}^{(i)} = (x_1^{(i)}, \ldots, x_\ell^{(i)}) \in \mathbb{F}_2^\ell$, and then it receives $\{\hat{\mathbf{t}}_h^{(i,j)}\}_{h\in[\ell]}$ and P_j receives $\hat{\mathbf{q}}_h^{(j,i)} = \hat{\mathbf{t}}_h^{(i,j)} + x_h^{(i)} \cdot \hat{\Delta}^{(j)}, h \in [\ell]$.

Triple generation: Each party P_i uses only the first κ components of its shares. We denote them by $\tilde{\mathbf{q}}_h^{(i,j)}$, $\tilde{\Delta}^{(i)}$ and $\tilde{\mathbf{t}}_h^{(i,j)}$.

1. Each party P_i generates ℓ random $y_h^{(i)} \in \mathbb{F}_2$.
2. For each $i \in \mathcal{P}$ do:
   (a) Using a random oracle $H : \{0,1\}^* \to \{0,1\}$, break the correlation from the previous step. P_i locally computes $H(\tilde{\mathbf{t}}_h^{(i,j)}) = w_h^{(i,j)}$, and P_j locally computes $H(\tilde{\mathbf{q}}_h^{(j,i)}) = v_{0,h}^{(j,i)}, H(\tilde{\mathbf{q}}_h^{(j,i)} + \tilde{\Delta}^{(j)}) = v_{1,h}^{(j,i)}$, $\forall j \neq i, \forall h \in [\ell]$.
   (b) Parties need to create new correlations corresponding to y_h:
   - Each P_j, $j \neq i$, sends a vector $\mathbf{s}^{(j,i)} \in \mathbb{F}_2^\ell$ to P_i such that each component is $s_h^{(j,i)} = v_{0,h}^{(j,i)} + v_{1,h}^{(j,i)} + y_h^{(j)}$.
   - $\forall j \neq i$, P_i computes $n_h^{(i,j)} = w_h^{(i,j)} + x_h^{(i)} \cdot s_h^{(j,i)} = v_{0,h}^{(j,i)} + x_h^{(i)} \cdot y_h^{(j)}$.
3. Each P_i computes
$$z_h^{(i)} = \sum_{j\neq i} n_h^{(i,j)} + x_h^{(i)} \cdot y_h^{(i)} + \sum_{j\neq i} v_{0,h}^{(i,j)}.$$

Authentication:
1. Authenticate x_h by summing up the last κ components of the outputs from the COTe step to obtain $[\![x_h]\!]$, for $h = 1, \ldots, \ell$.
2. Call $\mathcal{F}_{[\![\cdot]\!]}^{\mathbb{F}_2^\kappa}$ with input $\mathsf{Authenticate}$ to authenticate $y_h^{(j)}, z_h^{(j)}$ for $j = 1, \ldots, n$ and $h = 1, \ldots, \ell$, obtaining $[\![y_h]\!], [\![z_h]\!]$

Check triples: This step performs sacrificing and combining, to check that the triples are correctly generated and to prevent any leakage on x_h in case y_h was authenticated incorrectly. The parties call the subprotocol $\Pi_{\mathsf{CheckTriples}}$ in Fig. 23.

Fig. 15. $\mathbb{F}_2$-triples generation

to *simultaneously* generate triples, $\langle z_h\rangle = \langle x_h\rangle \cdot \langle y_h\rangle$, and authenticate the random bits x_h, for $h = 1, \ldots, \ell$, under the fixed global key Δ, giving $[\![x_h]\!] = (\langle x_h\rangle, \langle \mathbf{m}_h\rangle, \langle \Delta\rangle)$. To do this, we need to double the length of the correlation used in $\mathcal{F}_{\mathsf{COTe}}$, so that half of the output is used to authenticate x_h, and the other half is hashed to produce shares of the random triple.[4]

The shares $\langle y_h\rangle, \langle z_h\rangle$ are then authenticated with additional calls to $\mathcal{F}_{\mathsf{COTe}}$ to obtain $[\![y_h]\!], [\![z_h]\!]$. We then use a random bucketing technique to combine the x_h values in several triples, removing any potential leakage due to incorrect authentication of y_h (avoiding the selective failure attack present in the previous protocol [16]) and then sacrifice to check for correctness (as in the previous protocol).

[4] If the correlation length is not doubled, and the same output is used both for authentication *and* as input to the hash function, we cannot prove UC security as the values and MACs of a triple are no longer independent.

The **Initialize** stage consists of initializing the functionality $\mathcal{F}_{\mathsf{COTe}}^{2\kappa,\ell}$ with $\hat{\Delta} \in \mathbb{F}_2^{2\kappa}$. Note that $\hat{\Delta}$ is the concatenation of a random $\tilde{\Delta} \in \mathbb{F}_2^{\kappa}$ and the MAC key Δ. We add a consistency check to ensure that each party initialize $\hat{\Delta}$ correctly, as we did in $\Pi_{[\![\cdot]\!]}$.

Then, in **COTe.Extend**, each party P_i runs a $\mathsf{COTe}^{2\kappa,\ell}$ with all other parties on input $\mathbf{x}^{(i)} = (x_1^{(i)}, \ldots, x_\ell^{(i)}) \in \mathbb{F}_2^{\ell}$. For each $i \in \mathcal{P}$, we obtain $\hat{\mathbf{q}}_h^{(j,i)} = \hat{\mathbf{t}}_h^{(i,j)} + x_h^{(i)} \cdot \hat{\Delta}^{(j)}$, $h \in [\ell]$, where

$$\hat{\mathbf{q}}_h^{(j,i)} = (\tilde{\mathbf{q}}_h^{(j,i)} \| \mathbf{q}_h^{(j,i)}) \in \mathbb{F}_2^{2\kappa} \quad \text{and} \quad \hat{\mathbf{t}}_h^{(j,i)} = (\tilde{\mathbf{t}}_h^{(j,i)} \| \mathbf{t}_h^{(j,i)}) \in \mathbb{F}_2^{2\kappa}.$$

Note that we allow corrupt parties to input vectors $\mathbf{x}_h^{(i)}$ instead of bits.

Parties use the first κ components of their shares during the **Triple Generation** phase. More precisely, each party P_i samples ℓ random bits $y_h^{(i)}$ and then uses the first κ components of the output of $\mathsf{COTe}^{2\kappa,\ell}$ to generate shares $z_h^{(i)}$. The idea (as previously [16]) is that of using OT-relations to produce multiplicative triples. In step 2, in order to generate ℓ random and independent triples, we need to break the correlation generated by COTe. For this purpose we use a hash function H, but after that, as we need to "bootstrap" to an n-parties representation, we must create new correlations for each $h \in [\ell]$. P_i sums all the values $n_h^{(i,j)}, j \neq i$, and $x_h^{(i)} \cdot y_h^{(i)}$ to get $u_h^{(i,j)} = \sum_{j \neq i} n_h^{(j,i)} + x_h^{(i)} \cdot y_h^{(i)}$. Notice that adding up the share $u_h^{(i,j)}$ held by P_i and all the shares of other parties, after step 2 we have:

$$u_h^{(i,j)} + \sum_{j \neq i} v_{0,h}^{(j,i)} = x_h^{(i)} \cdot y_h.$$

Repeating this procedure for each $i \in \mathcal{P}$ and adding up, we get $z_h = x_h \cdot y_h$.

Once the multiplication triples are generated the parties **Authenticate** z_h and y_h using $\mathcal{F}_{[\![\cdot]\!]}$, while to authenticate x_h they use the remaining κ components of the outputs of the $\mathsf{COTe.Extend}$ step.

Checking Triples. In the last step we want to check that the authenticated triples are correctly generated. For this we use the subprotocol $\Pi_{\mathsf{CheckTriples}}$ in Appendix F.3. This generalizes the bucket-based cut-and-choose technique by Larraia et al. [16], optimizing the parameters and abstracting away the key properties that are needed from the data being checked. This means that the procedure can easily be adapted for other purposes.

The bucket-cut-and-choose step ensures that the generated triples are correct. Privacy on x is then guaranteed by the combine step, whereas privacy on y follows from the use of the original COTe for both creating triples and authenticating x. Note also that if a corrupt party inputs an inconsistent bit $x_h^{(i)}$ in $n_h^{(i,k)}$, for some $k \notin A$ in step 2.b, then the resulting triples $z_h = x_h \cdot y_h + s_h^{(k,i)} \cdot y_h$ will pass the checks if and only if $s_h^{(k,i)} = 0$, revealing nothing about y_h.

We conclude by stating the main result of this section.

Theorem 2. *For every static adversary $\mathcal{A}$ corrupting up to $n-1$ parties, the protocol $\Pi_{\mathsf{BitTriples}}$ κ-securely implements $\mathcal{F}_{\mathsf{Triples}}$ (Fig. 8) in the $(\mathcal{F}_{\mathsf{COTe}}^{\kappa,\ell}, \mathcal{F}_{[\![\cdot]\!]})$-hybrid model.*

Proof. Correctness easily follows from the above discussion. For more details see Appendix F.4.

6 Triple Generation for MiniMACs

In this section we describe how to construct the preprocessing data needed for the online execution of the MiniMAC protocol [11,9]. The complete protocols and security proofs are in Appendix G. Here we briefly outline the protocols and give some intuition of security.

6.1 Raw Material

The raw material used for MiniMAC is very similar to the raw material in both TinyOT and SPDZ. In particular this includes random multiplication triples. These are used in the same manner as $\mathbb{F}_2$ and $\mathbb{F}_{2^k}$ triples to allow for multiplication during an online phase. However, remember that we work on elements which are codewords of some systematic linear error correcting code, C. Thus an authenticated element is defined as $[\![C(\mathbf{x})]\!]^* = \{\langle C(\mathbf{x})\rangle, \langle \mathbf{m}\rangle, \langle \Delta\rangle\}$ where $\mathbf{m} = C(\mathbf{x}) * \Delta$ with $C(\mathbf{x})$, $\mathbf{m}$ and Δ elements of $\mathbb{F}_{2^u}^m$ and $\mathbf{x} \in \mathbb{F}_{2^u}^k$. Similarly a triple is a set of three authenticated elements, $\{[\![C(\mathbf{a})]\!]^*, [\![C(\mathbf{b})]\!]^*, [\![C^*(\mathbf{c})]\!]^*\}$ under the constraint that $C^*(\mathbf{c}) = C(\mathbf{a}) * C(\mathbf{b})$, where $*$ denotes component-wise multiplication. We notice that the multiplication of two codewords results in an element in the Schur transform. Since we might often be doing multiplication involving the result of another multiplication, that thus lives in C^*, we need some way of bringing elements from C^* back down to C. To do this we need another piece of raw material: the Schur pair. Such a pair is simply two authenticated elements of the same message, one in the codespace and one in the Schur transform. That is, the pair $\{[\![C(\mathbf{r})]\!]^*, [\![C^*(\mathbf{s})]\!]^*\}$ with $\mathbf{r} = \mathbf{s}$. After doing a multiplication using a preprocessed random triple in the online phase, we use the $[\![C^*(\mathbf{s})]\!]^*$ element to onetime pad the result, which can then be partially opened. This opened value is re-encoded using C and then added to $[\![C(\mathbf{r})]\!]^*$. This gives a shared codeword element in C, that is the correct output of the multiplication.

Finally, to avoid being restricted to just parallel computation within each codeword vector, we also need a way to reorganize these components within a codeword. To do so we need to construct "reorganization pairs". Like the Schur pairs, these will simply be two elements with a certain relation on the values they authenticate. Specifically, one will encode a random element and the other a linear function applied to the random element encoded by the first. Thus the pair will be $\{[\![C(\mathbf{r})]\!]^*, [\![C(f(\mathbf{r}))]\!]^*\}$ for some linear function $f : \mathbb{F}_{2^u}^k \to \mathbb{F}_{2^u}^k$. We use these by subtracting $[\![C(\mathbf{r})]\!]^*$ from the shared element we will be working on. We then partially open and decode the result. This is then re-encoded and added to $[\![C(f(\mathbf{r}))]\!]^*$, resulting in the linear computation defined by $f(\cdot)$ on each of the components.

6.2 Authentication

For the MiniMAC protocol to be secure, we need a way of ensuring that authenticated vectors always form valid codewords. We do this based on the functionality $\mathcal{F}_{\mathsf{CodeAuth}}$ in two steps, first a 'BigMAC' authentication, which is then compressed to give a 'MiniMAC' authentication. The steps are described by the **BigMAC**, respectively **Compress** phases in Fig. 31 and their implementations in Fig. 30 in Appendix G.2. For the BigMAC authentication, we simply use the $\mathcal{F}_{[\![\cdot]\!]}$ functionality to authenticate each component of $\mathbf{x}$ (living in $\mathbb{F}_{2^u}$) separately under the whole of $\Delta \in \mathbb{F}_{2^u}^m$. Because every component of $\mathbf{x}$ is then under the same MAC key, we can compute MACs for the rest of the codeword $C(\mathbf{x})$ by simply linearly combining the MACs on $\mathbf{x}$, due to the linearity of C. We use the notation $[\![C(\mathbf{x})]\!] = \left\{\langle C(\mathbf{x})\rangle, \{\langle \mathbf{m}_{\mathbf{x}_i}\rangle\}_{i\in[m]}, \langle \Delta\rangle\right\}$ to denote the BigMAC share. To go from BigMAC to MiniMAC authentication, we just extract the relevant $\mathbb{F}_{2^u}$ element from each MAC. We then use $[\![C(\mathbf{x})]\!] = \{\langle C(\mathbf{x})\rangle, \langle \mathbf{m}_{\mathbf{x}}\rangle, \langle \Delta\rangle\}$ to denote a MiniMAC element, where $\mathbf{m}_{\mathbf{x}}$ is made up of one component of each of the m BigMACs.

6.3 Multiplication Triples

To generate a raw, unauthenticated MiniMAC triple, we need to be able to create vectors of shares $\langle C(\mathbf{a})\rangle$, $\langle C(\mathbf{b})\rangle$, $\langle C^*(\mathbf{c})\rangle$ where $C^*(\mathbf{c}) = C(\mathbf{a}) * C(\mathbf{b})$ and $\mathbf{a}, \mathbf{b} \in \mathbb{F}_{2^u}^k$. These can then be authenticated using the $\mathcal{F}_{\mathsf{CodeAuth}}$ functionality described above.

Since the authentication procedure only allows shares of valid codewords to be authenticated, it might be tempting to directly use the SPDZ triple generation protocol from Section 5.1 in $\mathbb{F}_{2^u}$ for each component of the codewords $C(\mathbf{a})$ and $C(\mathbf{b})$. In this case, it is possible that parties do not input valid codewords, but this would be detected in the authentication stage. However, it turns out this approach is vulnerable to a subtle selective failure attack – a party could input to the triple protocol a share for $C(\mathbf{a})$ that differs from a codeword in just one component, and then change their share to the correct codeword before submitting

it for authentication. If the corresponding component of $C(\mathbf{b})$ is zero then this would go undetected, leaking that fact to the adversary.

To counter this, we must ensure that shares output by the triple generation procedure are guaranteed to be codewords. To do this, we only generate shares of the $\mathbb{F}_{2^u}^k$ vectors $\mathbf{a}$ and $\mathbf{b}$ – since C is a linear $[m, k, d]$ code, the shares for the parity components of $C(\mathbf{a})$ and $C(\mathbf{b})$ can be computed locally. For the product $C^*(\mathbf{c})$, we need to ensure that the first $k^* \geq k$ components can be computed, since C^* is a $[m, k^*, d^*]$ code. Note that the first k components are just $(\mathbf{a}_1, \ldots, \mathbf{a}_k) * (\mathbf{b}_1, \ldots, \mathbf{b}_k)$, which could be computed similarly to the SPDZ triples. However, for the next $k^* - k$ components, we also need the cross terms $\mathbf{a}_i \cdot \mathbf{b}_j$, for every $i, j \in [k]$. To ensure that these are computed correctly, we input vectors containing all the bits of $\mathbf{a}, \mathbf{b}$ to $\mathcal{F}_{\mathsf{ACOT}}$, which outputs the tensor product $\mathbf{a} \otimes \mathbf{b}$, from which all the required codeword shares can be computed locally. Similarly to the BigMAC authentication technique, this results in an overhead of $O(k \cdot u) = O(\kappa \log \kappa)$ for every multiplication triple when using Reed-Solomon codes. We express the above operations in the subprotocol CodeOT in Fig. 32 in Appendix G.3.

Taking our departure in the above description we generate the multiplication triples in two steps: First unauthenticated multiplication triples are generated by using the CodeOT subprotocol, which calls $\mathcal{F}_{\mathsf{ACOT}}$ and takes the diagonal of the resulting shared matrices. The codewords of these diagonals are then used as inputs to $\mathcal{F}_{\mathsf{CodeAuth}}$, which authenticates them. This is described by protocol $\Pi_{\mathsf{UncheckedMiniTriples}}$ in Fig. 33. Then a random pairwise sacrificing is done to ensure that it was in fact shares of multiplication being authenticated. This is done using protocol $\Pi_{\mathsf{MiniTriples}}$ in Fig. 36. One minor issue that arises during this stage is that we also need to use a Schur pair to perform the sacrifice, to change one of the multiplication triple outputs back down to the code C, before it is multiplied by a challenge codeword and checked.

Security intuition. Since the CodeOT procedure is guaranteed to produces shares of valid codewords, and the authentication procedure can only be used to authenticate valid codewords, if an adversary changes their share before authenticating it, they must change it in at least d positions, where d is the minimum distance of the code. For the pairwise sacrifice check to pass, the adversary then has to essentially guess d components of the random challenge codeword to win, which only happens with probability $2^{-u \cdot d}$.

6.4 Schur and Reorganization Pairs

The protocols Π_{Schur} (Fig. 38) and Π_{Reorg} (Fig. 40) describe how to create the Schur and reorganization pairs. We now give a brief intuition of how these work.

Schur Pairs. We require random authenticated codewords $[\![C(\mathbf{r})]\!]^*, [\![C^*(\mathbf{s})]\!]^*$ such that the first k components of $\mathbf{r}$ and $\mathbf{s}$ are equal. Note that since $C \subset C^*$, it might be tempting to use the same codeword (in C) for both elements. However, this will be insecure – during the online phase, parties reveal elements of the form $[\![C^*(\mathbf{x} * \mathbf{y})]\!]^* - [\![C^*(\mathbf{s})]\!]^*$. If $C^*(\mathbf{s})$ is actually in the code C then it is uniquely determined by its first k components, which means $C^*(\mathbf{x} * \mathbf{y})$ will not be masked properly and could leak information on $\mathbf{x}, \mathbf{y}$.

Instead, we have parties authenticate a random codeword in C^* that is zero in the first k positions, reveal the MACs at these positions to check that this was honestly generated, and then add this to $[\![C(\mathbf{r})]\!]^*$ to obtain $[\![C^*(\mathbf{s})]\!]^*$. This results in a pair where the parties' shares are identical in the first k positions, however we prove in Section G.4 that this does not introduce any security issues for the online phase.

Reorganizing Pairs. To produce the pairs $[\![C(\mathbf{r})]\!]^*, [\![C(f(\mathbf{r}))]\!]^*$, we take advantage of the fact that during BigMAC authentication, every component of a codeword vector has the same MAC key. This means linear functions can be applied across the components, which makes creating the required data very straightforward. Note that with MiniMAC shares, this would not be possible, since you cannot add two elements with different MAC keys.

7 Complexity Analysis

We now turn to analyzing the complexity of our triple generation protocols, in terms of the required number of correlated and random OTs (on κ-bit strings) and the number of parties n.

Two-party TinyOT. The appendix of TinyOT [18] states that 54 aBits are required to compute an AND gate, when using a bucket size of 4. An aBit is essentially a passive correlated OT combined with a consistency check and some hashes, so we choose to model this as roughly the cost of an actively secure random OT.

Multi-party TinyOT. Note that although the original protocol of Larraia et al. [16] and the fixed protocol of Burra et al. [5] construct secret-shared OT quadruples, these are locally equivalent to multiplication triples, which turn out to be simpler to produce as one less authentication is required. Producing a triple requires one random OT per pair of parties, and the 3 correlated OTs per pair of parties to authenticate the 3 components of each triple. Combining twice, and sacrificing gives an additional overhead of B^3, where B is the bucket size. When creating a batch of at least 1 million triples with statistical security parameter 40, the proofs in Appendix F.3 show that we can use bucket size 3, giving $81n(n-1)$ calls to $\mathcal{F}_{\mathsf{COTe}}$ and $27n(n-1)$ to $\mathcal{F}_{\mathsf{OT}}$.

Authentication. To authenticate a single bit, the $\Pi_{[\![\cdot]\!]}$ protocol requires $n(n-1)$ calls to $\mathcal{F}_{\mathsf{COTe}}$. For full field elements in $\mathbb{F}_{2^k}$ this is simply performed k times, taking $kn(n-1)$ calls.

$\mathbb{F}_2$ Triples. The protocol starts with $n(n-1)$ calls to $\mathcal{F}_{\mathsf{COTe}}$ to create the initial triple and authenticate x; however, these are on strings of length 2κ rather than κ and also require a call to H, so we choose to count this as $n(n-1)$ calls to both $\mathcal{F}_{\mathsf{OT}}$ and $\mathcal{F}_{\mathsf{COTe}}$ to give a conservative estimate. Next, y and z are authenticated using $\mathcal{F}_{[\![\cdot]\!]}$, needing a further $2n(n-1) \times \mathcal{F}_{\mathsf{COTe}}$.

We need to sacrifice once and combine once, and if we again use buckets of size 3 this gives a total overhead of 9x. So the total cost of an $\mathbb{F}_2$ triple with our protocol is $27n(n-1)$ $\mathcal{F}_{\mathsf{COTe}}$ calls and $9n(n-1)$ $\mathcal{F}_{\mathsf{OT}}$ calls.

$\mathbb{F}_{2^k}$ Triples. We start with $n(n-1)$ calls to $\mathcal{F}_{\mathsf{ACOT}}^{k,s}$, each of which requires $3k$ $\mathcal{F}_{\mathsf{OT}}$ calls, assuming that k is equal to the statistical security parameter. We then need to authenticate the resulting triple (three field elements) for a cost of $3kn(n-1)$ calls to $\mathcal{F}_{\mathsf{COTe}}$. The sacrificing step in the checked triple protocol wastes one triple to check one, so doubling these numbers gives $6kn(n-1)$ for each of $\mathcal{F}_{\mathsf{OT}}$ and $\mathcal{F}_{\mathsf{COTe}}$.

MiniMAC Triples. Each MiniMAC triple also requires one Schur pair for the sacrificing step and one Schur pair for the online phase multiplication protocol.

Codeword Authentication. Authenticating a codeword with Π_{CodeAuth} takes k calls to $\mathcal{F}_{[\![\cdot]\!]}$ on u-bit field elements, giving $kun(n-1)$ COTe's on a $u \cdot m$-bit MAC key. Since COTe is usually performed with a κ-bit MAC key and scales linearly, we choose to scale by $u \cdot m/\kappa$ and model this as $ku^2mn(n-1)/\kappa$ calls to $\mathcal{F}_{\mathsf{COTe}}$.

Schur and Reorganization Pairs. These both just perform 1 call to $\mathcal{F}_{\mathsf{CodeAuth}}$, so have the same cost as above.

Multiplication Triples. Creating an unchecked triple first uses $n(n-1)$ calls to CodeOT on $k \cdot u$-bit strings, each of which calls $\mathcal{F}_{\mathsf{ACOT}}$, for a total of $(2ku+s)n(n-1)$ $\mathcal{F}_{\mathsf{OT}}$'s. The resulting shares are then authenticated with 3 calls to $\mathcal{F}_{\mathsf{CodeAuth}}$. Pairwise sacrificing doubles all of these costs, to give $2kun(n-1)(2ku+s)/\kappa$ $\mathcal{F}_{\mathsf{OT}}$'s and 6 calls to $\mathcal{F}_{\mathsf{CodeAuth}}$, which becomes $8ku^2mn(n-1)/\kappa$ $\mathcal{F}_{\mathsf{COTe}}$'s when adding on the requirement for two Schur pairs.

Parameters. [9] implemented the online phase using Reed-Solomon codes over $\mathbb{F}_{2^8}$, with $(m,k) = (256,120)$ and $(255,85)$, for a 128-bit statistical security level. The choice $(255,85)$ allowed for efficient FFT encoding, resulting in a much faster implementation, so we choose to follow this and use $u = 8, k = 85$. This means the cost of a single (vector) multiplication triple is $86700n(n-1)$ calls to $\mathcal{F}_{\mathsf{COTe}}$ and $14875(n-1)$ calls to $\mathcal{F}_{\mathsf{OT}}$. Scaling this down by k, the amortized cost of a single $\mathbb{F}_{2^u}$ multiplication becomes $1020(n-1)$ and $175(n-1)$

calls. Note that this is around twice the cost of $\mathbb{F}_{2^{40}}$ triples, which were used to embed the AES circuit by Damgård et al. [7], so it seems that although the MiniMAC online phase was reported by Damgård et al. [9] to be more efficient than other protocols for certain applications, there is some extra cost when it comes to the preprocessing using our protocol.

7.1 Estimating Runtimes

To provide rough estimates of the runtimes for generating triples, we use the OT extension implementation of Asharov et al. [1] to provide estimates for $\mathcal{F}_{\mathsf{COTe}}$ and $\mathcal{F}_{\mathsf{OT}}$. For $\mathcal{F}_{\mathsf{COTe}}$, we simply use the time required for a passively secure extended OT ($1.07\mu s$), and for $\mathcal{F}_{\mathsf{OT}}$ the time for an actively secure extended OT ($1.29\mu s$) (both running over a LAN). Note that these estimates will be too high, since $\mathcal{F}_{\mathsf{COTe}}$ does not require hashing, unlike a passively secure random OT. However, there will be additional overheads due to communication etc, so the figures given in Table 1 are only supposed to be a rough guide.

8 Acknowledgements

We would like to thank Nigel Smart, Rasmus Zakarias and the anonymous reviewers, whose comments helped to improve the paper. The first author has been supported by the Danish National Research Foundation and The National Science Foundation of China (under the grant 61361136003) for the Sino-Danish Center for the Theory of Interactive Computation and from the Center for Research in Foundations of Electronic Markets (CFEM), supported by the Danish Strategic Research Council. Furthermore, partially supported by Danish Council for Independent Research via DFF Starting Grant 10-081612 and the European Research Commission Starting Grant 279447. The second, third and fourth authors have been supported in part by EPSRC via grant EP/I03126X.

References

1. G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer extensions with security for malicious adversaries. In *Advances in Cryptology – EUROCRYPT 2015*, pages 673–701, 2015.
2. D. Beaver. Efficient multiparty protocols using circuit randomization. *Advances in Cryptology - CRYPTO 1991*, 1992.
3. M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993.*, pages 62–73, 1993.
4. R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-homomorphic encryption and multiparty computation. *Advances in Cryptology – EUROCRYPT 2011*, pages 169–188, 2011.
5. S. S. Burra, E. Larraia, J. B. Nielsen, P. S. Nordholt, C. Orlandi, E. Orsini, P. Scholl, and N. P. Smart. High performance multi-party computation for binary circuits based on oblivious transfer. Cryptology ePrint Archive, Report 2015/472, 2015. http://eprint.iacr.org/.
6. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 136–145, 2001.
7. I. Damgård, M. Keller, E. Larraia, C. Miles, and N. P. Smart. Implementing AES via an actively/covertly secure dishonest-majority MPC protocol. In I. Visconti and R. D. Prisco, editors, *SCN*, volume 7485 of *Lecture Notes in Computer Science*, pages 241–263. Springer, 2012.
8. I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In J. Crampton, S. Jajodia, and K. Mayes, editors, *ESORICS*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.
9. I. Damgård, R. Lauritsen, and T. Toft. An empirical study and some improvements of the minimac protocol for secure computation. In *Security and Cryptography for Networks - 9th International Conference, SCN 2014, Amalfi, Italy, September 3-5, 2014. Proceedings*, pages 398–415, 2014.

10. I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In R. Safavi-Naini and R. Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.
11. I. Damgård and S. Zakarias. Constant-overhead secure computation of boolean circuits using preprocessing. In *TCC*, pages 621–641, 2013.
12. Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, pages 145–161, 2003.
13. M. Keller, E. Orsini, and P. Scholl. Actively secure OT extension with optimal overhead. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, pages 724–741, 2015.
14. M. Keller and P. Scholl. Efficient, oblivious data structures for MPC. In *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*, pages 506–525, 2014.
15. M. Keller, P. Scholl, and N. P. Smart. An architecture for practical actively secure MPC with dishonest majority. In *ACM Conference on Computer and Communications Security*, pages 549–560, 2013.
16. E. Larraia, E. Orsini, and N. P. Smart. Dishonest majority multi-party computation for binary circuits. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, pages 495–512, 2014.
17. J. B. Nielsen. Extending oblivious transfers efficiently - how to get robustness almost for free. *IACR Cryptology ePrint Archive*, 2007:215, 2007.
18. J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A new approach to practical active-secure two-party computation. In *Advances in Cryptology–CRYPTO 2012*, pages 681–700. Springer, 2012.
19. K. Pietrzak. Subspace LWE. In *Theory of Cryptography - 9th Theory of Cryptography Conference, TCC 2012, Taormina, Sicily, Italy, March 19-21, 2012. Proceedings*, pages 548–563, 2012.

A UC Security

We work in the standard Universal Composability (UC) framework of Canetti [6]. The UC framework introduces a PPT environment $\mathcal{Z}$ that is invoked on the security parameter κ and an auxiliary input $z \in \{0,1\}^*$ and oversees the execution of a protocol in one of the two worlds. The "ideal" world execution involves dummy parties $\pi_1, \ldots, \pi_n$, an ideal adversary $\mathcal{S}$ who may corrupt some of the dummy parties, and a functionality $\mathcal{F}$. The "real" world execution involves the PPT parties $P_1, \ldots, P_n$, possibly corrupted by a real world adversary $\mathcal{A}$, and interacting with each other by means of a protocol Π realizing an ideal functionality $\mathcal{F}$. The environment $\mathcal{Z}$ chooses the input of the parties and may interact with the ideal/real adversary during the execution. At the end of the execution, it has to decide upon and output whether a real or an ideal world execution has taken place.

We let $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\kappa, z)$ denote the random variable describing the output of the environment $\mathcal{Z}$ after interacting with the ideal execution with adversary $\mathcal{S}$, the functionality $\mathcal{F}$, on the security parameter κ and z. Let $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ denote the ensemble $\{\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\kappa, z)\}_{\kappa\in\mathbb{N},z\in\{0,1\}t^*}$. Similarly let $\mathsf{REAL}_{\mathcal{P},\mathcal{A},\mathcal{Z}}(1^\kappa, z)$ denote the random variable describing the output of the environment $\mathcal{Z}$ after interacting in a real execution of a protocol Π with adversary $\mathcal{A}$, the parties $\mathcal{P}$, on the security parameter κ and z. Let $\mathsf{REAL}_{\mathcal{P},\mathcal{A},\mathcal{Z}}$ denote the ensemble $\{\mathsf{REAL}_{\mathcal{P},\mathcal{A},\mathcal{Z}}(\kappa, z)\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$.

Also, the UC framework considers the $\mathcal{G}$-hybrid world, where the computation proceeds as in the "real" world with the additional assumption that the parties have access to an auxiliary ideal functionality $\mathcal{G}$. In this model, honest parties do not communicate with the ideal functionality directly, but instead the adversary delivers all the messages to and from the ideal functionality. We consider the communication channels to be ideally authenticated, so that the adversary may read but not modify these messages. Unlike messages exchanged between parties, which can be read by the adversary, the messages exchanged between parties and the ideal functionality are divided into a *public header* and a *private header*. The public header can be read by the adversary and contains non-sensitive information (such as session identifiers, type of message, sender and receiver). On the other hand, the private header cannot be read by the adversary and contains information such as the parties private inputs. We denote the ensemble of environment outputs that represents the execution of a protocol Π in a $\mathcal{G}$-hybrid model as $\mathsf{HYB}^{\mathcal{G}}_{\Pi,\mathcal{A},\mathcal{Z}}$ (defined analogously to $\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}}$). UC security is then formally defined as:

Definition 1. *For $n \in \mathbb{N}$, let $\mathcal{F}$ be an n-ary functionality and let Π be an n-party protocol. We say that Π securely realizes $\mathcal{F}$ in the $\mathcal{G}$-hybrid model if for every environment $\mathcal{Z}$, for every PPT real world adversary $\mathcal{A}$, there exists a PPT ideal world adversary $\mathcal{S}$, corrupting the same parties, such that*

$$\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}} \approx \mathsf{HYB}^{\mathcal{G}}_{\Pi,\mathcal{A},\mathcal{Z}}.$$

The crucial aspect of universal composability framework is the composition theorem. It works as follows: denote by $\pi \circ \mathcal{G}$ a protocol π that during its execution makes calls to an ideal functionality $\mathcal{G}$. The composition proof shows that if $\pi_f \circ \mathcal{G}$ implements $\mathcal{F}$ and if π_g securely implements $\mathcal{G}$, then $\pi_f \circ \pi_g$ securely implements $\mathcal{F}$. This provides modularity in construction of protocols and simplifies proofs dramatically. It is also shown that proving security against a dummy adversary, *i.e.* one that acts as a communication channel, is sufficient for proving general security.

B Information Theoretic Tags for Dishonest Majority

In this section we recall the $\mathsf{MACCheck}$ protocol by Damgård et al. [8]. The procedure utilizes an ideal functionality $\mathcal{F}_{\mathsf{Comm}}$ for commitments given in Figure 17. An implementation of $\mathcal{F}_{\mathsf{Comm}}$ in the random oracle model can be found in the Appendix of Damgård et al.

Theorem 3 (Damgård et al. [8]). *The protocol $\mathsf{MACCheck}$ is correct, i.e. it accepts if all the public values b_i, and the corresponding MACs are correctly computed. Moreover, it is sound, i.e. it rejects except with probability $2^{-\kappa+1}$ in case at least one value, or MAC, is not correctly computed.*

Protocol Π_{MACCheck}

Usage: The parties have a set of $[\![a_h]\!]$, sharings and public values b_h, for $h = 1, \ldots, t$, and they wish to check that $a_h = b_h$, i.e. they want to check whether the public values are consistent with the shared MACs held by the parties.

As input the system has sharings $\left(\langle \Delta \rangle, \{b_h, \langle a_h \rangle, \langle \mathbf{m}_{a_h} \rangle\}_{h=1}^{t}\right)$. If the MAC values are correct then we have that $\mathbf{m}_{a_h} = b_h \cdot \Delta$, for all h.

$\mathsf{MACCheck}(\{b_1, \ldots, b_t\})$**:**

1. Every party P_i samples a seed $s^{(i)}$ and asks $\mathcal{F}_{\mathsf{Comm}}$ to broadcast $\tau^{(i)} = \mathsf{Comm}(s^{(i)})$.
2. Every party P_i calls $\mathcal{F}_{\mathsf{Comm}}$ with $\mathsf{Open}(\tau^{(i)})$ and all parties obtain $s^{(j)}$ for all $j \in \mathcal{P}$.
3. Set $s = s^{(1)} + \cdots + s^{(n)}$.
4. Parties sample a random element $\mathbf{r} = \mathsf{PRF}(\mathbb{F}_{2^\kappa}^t)_s \in \mathbb{F}_{2^\kappa}^t$; note all parties obtain the same vector as they have agreed on the seed s.
5. Each party computes the public value $b = \sum_{h=1}^{t} r_h \cdot b_h$.
6. The parties locally compute the sharings $\langle \mathbf{m}_a \rangle = r_1 \cdot \langle \mathbf{m}_{a_1} \rangle + \cdots + r_t \cdot \langle \mathbf{m}_{a_t} \rangle$ and $\langle \sigma \rangle = \langle \mathbf{m}_a \rangle - b \cdot \langle \Delta \rangle$.
7. Party i asks $\mathcal{F}_{\mathsf{Comm}}$ to broadcast his share $\bar{\tau}^{(i)} = \mathsf{Comm}(\sigma^{(i)})$.
8. Every party calls $\mathcal{F}_{\mathsf{Comm}}$ with $\mathsf{Open}(\bar{\tau}^{(i)})$, and all parties obtain $\sigma^{(j)}$ for all $j \in \mathcal{P}$.
9. If $\sigma^{(1)} + \cdots + \sigma^{(n)} \neq 0$, the parties output $\varnothing$ and abort, otherwise they accept all b_h as valid authenticated bits.

Fig. 16. Protocol Π_{MACCheck} - For checking MACs on partially opened values

The Functionality $\mathcal{F}_{\mathsf{Comm}}$

Commit: On input $(\mathsf{Comm}, v, i, \tau_v)$ by P_i or the adversary on his behalf (if P_i is corrupt), where v is either in a specific domain or $\perp$, it stores (v, i, τ_v) on a list and outputs (i, τ_v) to all parties and adversary.

Open: On input $(\mathsf{Open}, i, \tau_v)$ by P_i or the adversary on his behalf (if P_i is corrupt), the ideal functionality outputs (v, i, τ_v) to all parties and adversary. If $(\mathsf{NoOpen}, i, \tau_v)$ is given by the adversary, and P_i is corrupt, the functionality outputs $(\perp, i, \tau_v)$ to all parties.

Fig. 17. Ideal Commitments

1. Receive $\{\chi^{(j)}\}_{j \in [N]}$ from $\mathcal{F}_{\mathsf{BatchCheck}}$ and pass this to the corrupted parties.
2. Emulating $\mathcal{F}_{\mathsf{Comm}}$, receive σ_i from every corrupted party i.
3. Emulating $\mathcal{F}_{\mathsf{Comm}}$, sample τ_i and send it to all corrupted parties for every honest party i.
4. Compute $\zeta = \sum_{i \in A} \sigma_i$ and send it to $\mathcal{F}_{\mathsf{BatchCheck}}$.
5. Receive $\{\sigma_i\}_{i \in B}$ from $\mathcal{F}_{\mathsf{BatchCheck}}$.
6. For every honest party i, send σ_i it to all corrupted parties.

Fig. 18. Simulator for batch checking protocol

C Batch Checking

We use a simple protocol based on $\mathcal{F}_{\mathsf{Comm}}$ and $\mathcal{F}_{\mathsf{Rand}}$ for checking that a batch of additively shared values are equal to zero, by opening a linear combination of them. This is similar to what was used in the MAC checking procedure of SPDZ [10,8], but in our protocol we also use it for checking correctness of a batch of $\mathbb{F}_{2^k}$ triples. Because of the nature of UC, the $\mathcal{F}_{\mathsf{BatchCheck}}$ functionality closely matches the protocol, only performing the check on a random linear combination of the inputs. Later on, we use Lemmas 6 and 7 to argue that this implies the original inputs were zero, with high probability.

Lemma 4. *The protocol $\Pi_{\mathsf{BatchCheck}}$ in Figure 16 securely implements $\mathcal{F}_{\mathsf{BatchCheck}}$ in the $(\mathcal{F}_{\mathsf{Rand}}, \mathcal{F}_{\mathsf{Comm}})$-hybrid model.*

Proof. We use the simulator in Figure 18. The use of $\mathcal{F}_{\mathsf{Comm}}$ ensures that the corrupted parties cannot chose $\{\sigma_i\}_{i \in A}$ dependent on $\{\sigma_i\}_{i \in B}$.

Protocol $\Pi^{\kappa,\ell}_{\mathsf{COTe}}$

Initialize: This is independent of inputs and only needs to be done once.

1. P_R samples κ pairs of κ-bit seeds, $\{(\mathbf{k}_0^j, \mathbf{k}_1^j)\}_{j=1}^{\kappa}$.
2. P_S samples a random κ-bit string Δ.
3. The parties call $\mathcal{F}_{\mathsf{OT}}^{\kappa,\kappa}$ with inputs $\{\mathbf{k}_0^j, \mathbf{k}_1^j\}_{j\in[\kappa]}$ and $\{\Delta[j]\}_{j\in[\kappa]}$.
4. P_R receives $\mathbf{k}^j_{\Delta_j}$ for $j = 1, \ldots, \kappa$.

Extend(P_R, P_S): On input $x_1, \ldots, x_\ell$ from P_R, do:

1. Expand $\mathbf{k}_0^j$ and $\mathbf{k}_1^j$ using a pseudo random generator (PRG) [a], letting
$$\mathbf{t}_0^j = \mathsf{PRG}(\mathbf{k}_0^j) \in \mathbb{F}_2^\ell \quad \text{and} \quad \mathbf{t}_1^j = \mathsf{PRG}(\mathbf{k}_1^j) \in \mathbb{F}_2^\ell, \quad j = 1, \ldots, \kappa.$$
so P_R knows $(\mathbf{t}_0^j, \mathbf{t}_1^j)$ and P_S knows $\mathbf{t}^j_{\Delta_j}$ for $j = 1, \ldots, \kappa$.
2. P_R computes
$$\mathbf{u}^j = \mathbf{t}_0^j + \mathbf{t}_1^j + \mathbf{x} \in \mathbb{F}_2^\ell, \quad j = 1, \ldots, \kappa,$$
where $\mathbf{x} = (x_1, \ldots, x_\ell) \in \mathbb{F}_2^\ell$ and sends them to P_S. Here we are creating the keys correlation that permits to extend OTs, inverting the role of sender and receiver.
3. P_S computes
$$\mathbf{q}^j = \Delta_j \cdot \mathbf{u}^j + \mathbf{t}^j_{\Delta_j} \in \mathbb{F}_2^\ell.$$
Notice that $\mathbf{q}^j = \mathbf{t}_0^j + \Delta_j \cdot \mathbf{x}$, for $j = 1, \ldots, \kappa$.
4. Let $\mathbf{q}_h$ denote the h-th row of the $\ell \times \kappa$ bit matrix $Q = [\mathbf{q}^1 | \ldots | \mathbf{q}^\kappa]$, and similarly let $\mathbf{t}_h$ be the h-th row of $[\mathbf{t}_0^1 | \ldots | \mathbf{t}_0^\kappa]$. Note that
$$\mathbf{q}_h = \mathbf{t}_h + x_h \cdot \Delta, \quad h = 1, \ldots, \ell.$$
5. P_R outputs $\mathbf{t}_h$, P_S outputs $\mathbf{q}_h$.

[a] If Extend is being iterated, set $\mathbf{t}_0^j, \mathbf{t}_1^j$ instead to be the next ℓ bits output from the PRG, to create fresh randomness.

Fig. 19. Protocol for correlated OT with errors between P_R and P_S.

Functionality $\mathcal{F}^{\kappa,\ell}_{\mathsf{OT}}$

$\mathcal{F}$ running with P_R and P_S and an adversary $\mathcal{S}$ proceeds as follows:

- The functionality waits for input $(\mathbf{v}_{0,h}, \mathbf{v}_{1,h}) \in \mathbb{F}_2^\kappa \times \mathbb{F}_2^\kappa$, $h \in [\ell]$, from P_S and $x_1, \ldots, x_\ell$, with $x_h \in \mathbb{F}_2$, from P_R.
- It outputs $\mathbf{v}_{x_h,h}$, $h \in [\ell]$, to P_R.

Fig. 20. The OT functionality

D IKNP extension and other OT functionalities

The protocol in Fig. 19 is the passively secure IKNP OT extension, using the standard OT functionality given in Fig. 20. Note that the receiver can cheat in Step 2 by using different values of $\mathbf{x}$; this is modeled in the $\mathcal{F}_{\mathsf{COTe}}$ functionality (Fig. 2) by allowing P_R to input vectors $\mathbf{x}_1, \ldots, \mathbf{x}_\ell$ instead of bits. This means that the protocol can be shown (and was proven by e.g. Nielsen [17]) to securely implement the $\mathcal{F}_{\mathsf{COTe}}$ functionality with *active security*, in a $\mathcal{F}_{\mathsf{OT}}^{\kappa,\kappa}$-hybrid model.

E Other functionalities

Functionality $\mathcal{F}_{\mathsf{Rand}}^{\mathbb{F}}$

Random sample: Upon receiving $(rand; u)$ from all parties, it samples a uniform $r \in \mathbb{F}$ and outputs $(rand, r)$ to all parties.

Fig. 21. Functionality $\mathcal{F}_{\mathsf{Rand}}^{\mathbb{F}}$

F Security proofs

For some of our proofs we require the following simple, technical lemma on the rank of a random matrix over $\mathbb{F}_2$.

Lemma 5. *Let A be a random $(k+m) \times k$ matrix over $\mathbb{F}_2$, where $m > 0$. Then A has rank k except with probability less than 2^{-m}.*

Proof. See [19][Lemma 1].

We will also use the following lemmas on sums with random coefficients.

Lemma 6 (Principle of Deferred Decisions). *Let $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i\in[N]}, \mathbf{z}$ be random variables in $\mathbb{F}_{2^k}$ for some $k, N \in \mathbb{N}$. Furthermore, let $\mathbf{x}_i$ be uniformly distributed and independent of all other variables for all $i \in [N]$. Then, if there exists $j \in N$ such that $\mathbf{y}_j \neq 0$,*

$$\Pr\left[\mathbf{z} = \sum_{i=1}^{N} \mathbf{x}_i \cdot \mathbf{y}_i\right] = 2^{-k}.$$

Proof.

$$\mathbf{z} = \sum_{i=1}^{N} \mathbf{x}_i \cdot \mathbf{y}_i$$

is equivalent to

$$\mathbf{x}_j = \mathbf{y}_j^{-1} \cdot \Big(\mathbf{z} + \sum_{i\neq j} \mathbf{x}_i \cdot \mathbf{y}_i\Big).$$

The claim follows because the left hand side is uniformly distributed and independent of the right hand side by definition.

Lemma 7. *Let $\{\mathbf{y}_{i,j,l}\}_{i\in[N_1],j\in[N_2],l\in[0,N_3]}, \mathbf{z}$ be random variables in $\mathbb{F}_{2^k}$ for some $k, N_1, N_2, N_3 \in \mathbb{N}$. Furthermore, let $\{\mathbf{x}_i\}_{i\in[N_1]}, \{\mathbf{w}_j\}_{j\in[N_2]}, \{\mathbf{v}_l\}_{l\in[N_3]}$ be uniformly distributed random variables, of which $\{\mathbf{x}_i\}_{i\in[N_1]}$ are independent of all other variables as well as mutually. Then, if there exists $(i', j', l') \in [N_1] \times [N_2] \times [0, N_3]$ such that $\mathbf{y}_{i',j',l'} \neq 0$,*

$$\Pr\left[\mathbf{z} = \sum_{i=1}^{N_1} \mathbf{x}_i \cdot \sum_{j=1}^{N_2} \mathbf{w}_j \cdot \Big(\mathbf{y}_{i,j,0} + \sum_{l=1}^{N_3} \mathbf{v}_l \cdot \mathbf{y}_{i,j,l}\Big)\right] \leq 3 \cdot 2^{-k}.$$

Proof. If $l' = 0$ and $\mathbf{w}_{j'} \neq 0$,

$$\mathbf{z} = \sum_{i=1}^{N_1} \mathbf{x}_i \cdot \sum_{j=1}^{N_2} \mathbf{w}_j \cdot \Big(\mathbf{y}_{i,j,0} + \sum_{l=1}^{N_3} \mathbf{v}_l \cdot \mathbf{y}_{i,j,l}\Big)$$

is equivalent to

$$\mathbf{x}_{i'} = \mathbf{w}_{j'}^{-1} \cdot \mathbf{y}_{i',j',0}^{-1} \cdot \Big(\mathbf{z} + \sum_{i\neq i'} \mathbf{x}_i \cdot \sum_{j=1}^{N_2} \mathbf{w}_j \cdot \Big(\mathbf{y}_{i,j,0} + \sum_{l=1}^{N_3} \mathbf{v}_l \cdot \mathbf{y}_{i,j,l}\Big) +$$

$$\sum_{i\neq i'} \mathbf{x}_i \cdot \sum_{j\neq j'} \mathbf{w}_j \cdot \left(\mathbf{y}_{i,j,0} + \sum_{l=1}^{N_3} \mathbf{v}_l \cdot \mathbf{y}_{i,j,l}\right)\Bigg) .$$

Because the left hand side is uniformly random and independent of the right hand side, equality holds with probability 2^{-k}. If $l' \neq 0$, $\mathbf{w}_{j'} \neq 0$, and $\mathbf{v}_{l'} \neq 0$, the equation is equivalent to

$$\begin{aligned}\mathbf{x}_{i'} = \mathbf{w}_{j'}^{-1} \cdot \mathbf{v}_{l'}^{-1} \cdot \mathbf{y}_{i',j',l'}^{-1} \cdot \Bigg(\mathbf{z} + \sum_{i\neq i'} \mathbf{x}_i \cdot \sum_{j=1}^{N_2} \mathbf{w}_j \cdot \left(\mathbf{y}_{i,j,0} + \sum_{l=1}^{N_3} \mathbf{v}_l \cdot \mathbf{y}_{i,j,l}\right) \\ + \sum_{i\neq i'} \mathbf{x}_i \cdot \sum_{j\neq j'} \mathbf{w}_j \cdot \left(\mathbf{y}_{i,j,0} + \sum_{l=1}^{N_3} \mathbf{v}_l \cdot \mathbf{y}_{i,j,l}\right) + \sum_{i\neq i'} \mathbf{x}_i \cdot \sum_{j\neq j'} \mathbf{w}_j \cdot \sum_{l\neq l'} \mathbf{v}_l \cdot \mathbf{y}_{i,j,l}\Bigg)\Bigg).\end{aligned}$$

Again, the equality holds with probability. Finally, the probability that $\mathbf{w}_{j'} = 0$ and $\mathbf{v}_{l'} = 0$ is 2^{-k}, respectively. The claim follows using the union bound.

Lemma 8. *For $N, \ell \in \mathbb{N}$, let $f_i : \mathbb{F}_2^\ell \to \mathbb{F}_{2^k}$ be a $\mathbb{F}_2$-linear map for all $i \in [N]$, $\{\mathbf{x}_i\}_{i\in[N]}$ be independent and uniformly distributed in $\mathbb{F}_{2^k}$, and $f = \sum_{i=1}^N \mathbf{x}_i \cdot f_i$. Furthermore, let $\mathbf{y}$ be independent and uniformly distributed in $\mathbb{F}^\ell$, and $\mathbf{z}$ a random variable independent of $\mathbf{y}$ (but not necessarily $\{\mathbf{x}_i\}_{i\in[N]}$). Then,*

$$\Pr\left[\mathbf{z} = f(\mathbf{y}) \wedge \exists i \in [N], \mathbf{y}' \in \mathbb{F}_2^\ell : (f(\mathbf{y}) = f(\mathbf{y}') \wedge f_i(\mathbf{y}) \neq f_i(\mathbf{y}'))\right] \leq 2^{-k+1}.$$

In other words, the probability that $f(\mathbf{y}) = \mathbf{z}$ and $f(\mathbf{y})$ does not uniquely determine $f_i(\mathbf{y})$ for all $i \in [N]$ is at most 2^{-k+1}.

Proof. Let $\{\mathbf{v}_j\}_{j\in[h]}$ be a basis of $V = (\bigcap_{i=1}^N \ker f_i)^\perp \subset \mathbb{F}_2^\ell$. Then, $f_i(\mathbf{y})$ can be expressed as $\sum_{j=1}^h a_j f_i(\mathbf{v}_j)$ for all $i \in [N]$. Let F_h denote the above event, which can be stated as follows:

$$F_h = \left(\mathbf{z} = f(\mathbf{y}) \wedge \exists i \in [N], \mathbf{y}' \in \mathbb{F}_2^\ell : (f(\mathbf{y}') = 0 \wedge f_i(\mathbf{y}') \neq 0)\right)$$

for h being the dimension of V. The equivalence is straightforward by replacing $\mathbf{y}'$ with $\mathbf{y} + \mathbf{y}'$. We will prove the statement by induction over h.

For $h = 1$, $\mathbf{y} = a_1\mathbf{v}_1$ for random a_1. $f_i(\mathbf{y}) \neq f_i(\mathbf{y}')$ can only hold for $\mathbf{y}' = (1 - a_1)\mathbf{v}_1$. Furthermore, $f(\mathbf{y}) = f(\mathbf{y}')$ is equivalent to $0 = f(\mathbf{y} - \mathbf{y}') = f(\mathbf{v}_1) = \sum_{i=1}^N f_i(\mathbf{v}_1)$. Since $\mathbf{v}_1 \in V$, there exists $i \in [N]$ such that $f_i(\mathbf{v}_1) \neq 0$. From Lemma 6, it follows that

$$\Pr\left[\exists i \in [N], \mathbf{y}' \in \mathbb{F}_2^\ell : (f(\mathbf{y}) = f(\mathbf{y}') \wedge f_i(\mathbf{y}) \neq f_i(\mathbf{y}'))\right] = 2^{-k}.$$

This implies that $\Pr[F_1] \leq 2^{-k+1}$.

Let E_j^h denote the event that $\{f(\mathbf{v}_g)\}_{g\in[h]}$ has $\mathbb{F}_2$-dimension j, that is, there exists $G \subset [h]$ of size $j + 1$ such that $\sum_{g\in[h]} f(\mathbf{v}_g) = 0$, but not such G of size j exists. The $\mathbb{F}_2$-dimension of a set of size h clearly is in $[0, h]$, and hence,

$$\Pr[F_h] = \Pr\left[F_h \cap \bigcup_{j=0}^h E_j^h\right] = \sum_{j=0}^h \Pr[F_h | E_j^h] \cdot \Pr[E_j^h].$$

In the event E_h^h, $0 = f(\mathbf{y}') = \sum_{g=1}^h a_g f(\mathbf{v}_g)$ for some $a_1, \ldots, a_h \in \mathbb{F}_2$ implies that $a_g = 0$ for all $g \in [h]$ and thus $f_i(\mathbf{y}') = f_i(\sum_{g=1}^h a_g\mathbf{v}_g) = 0$. Therefore, $\Pr[F_h|E_h^h] = 0$. On the other hand, for $j \neq h$, there exists $a_1, \ldots, a_h \in \mathbb{F}_2$ with $a_g = 1$ for some $g \in [h]$ and $0 = \sum_{g=1}^h a_h f(\mathbf{v}_g) = f(\sum_{g=1}^h a_h\mathbf{v}_g)$. Since $\sum_{g=1}^h a_g\mathbf{v}_g \in (\bigcap_{i=1}^N \ker f_i)^\top$, there exists $i \in [N]$ such that $f_i(\sum_{g=1}^h a_h\mathbf{v}_g) \neq 0$. It follows that

$$\Pr[F_h|E_j^h] = \Pr[\mathbf{z} = f(\mathbf{y})|E_j^h] = 2^{-j}$$

for $j \neq h$. The equality follows from the fact that $f(\mathbf{y})$ is uniformly random in a space of $\mathbb{F}_2$-dimension j in the event E_j^h and the fact that $\mathbf{y}$ is independent of $\mathbf{z}$. Hence,

$$\Pr[F_h] = \sum_{j=0}^{h-1} 2^{-j} \cdot \Pr[E_j^h] \tag{2}$$

Furthermore, if $\{f(\mathbf{v}_g)\}_{g\in[h]}$ has $\mathbb{F}_2$-dimension $j > 1$, $\{f(\mathbf{v}_g)\}_{g\in[h-1]}$ clearly has $\mathbb{F}_2$-dimension j or $j-1$. It follows that

$$\begin{aligned}
\Pr[F_h] &= Pr[E_0^h] + \sum_{j=1}^{h-1} 2^{-j} \cdot \big(\Pr[E_j^h|E_j^{h-1}] \cdot \Pr[E_j^{h-1}] + \Pr[E_j^h|E_{j-1}^{h-1}] \cdot \Pr[E_{j-1}^{h-1}]\big). \\
&= \Pr[E_0^h] + \sum_{j=1}^{h-1} 2^{-j} \cdot \Pr[E_j^h|E_j^{h-1}] \cdot \Pr[E_j^{h-1}] + \\
&\quad \sum_{j=0}^{h-2} 2^{-j-1} \cdot \Pr[E_{j+1}^h|E_j^{h-1}] \cdot \Pr[E_j^{h-1}].
\end{aligned}$$

Similarly,

$$\Pr[E_{j+1}^h|E_j^{h-1}] = (1 - \Pr[E_j^h|E_j^{h-1}]).$$

Therefore,

$$\begin{aligned}
\Pr[F_h] &= \Pr[E_0^h] + \sum_{j=1}^{h-1} 2^{-j} \cdot \Pr[E_j^h|E_j^{h-1}] \cdot \Pr[E_j^{h-1}] + \\
&\qquad \sum_{j=0}^{h-2} 2^{-j-1} \cdot (1 - \Pr[E_j^h|E_j^{h-1}]) \cdot \Pr[E_j^{h-1}] \\
&= \Pr[E_0^h] + 2^{-h+1} \cdot \Pr[E_{h-1}^h|E_{h-1}^{h-1}] \cdot \Pr[E_{h-1}^{h-1}] + \\
&\qquad \sum_{j=1}^{h-2} 2^{-j-1} \Pr[E_j^h|E_j^{h-1}] \cdot \Pr[E_j^{h-1}] \\
&\qquad - 2^{-1} \cdot \Pr[E_0^h|E_0^{h-1}] + \sum_{j=0}^{h-2} 2^{-j-1} \cdot \Pr[E_j^{h-1}] \\
&\leq \Pr[E_0^h|E_0^{h-1}] \cdot \Pr[E_0^{h-1}] + 2^{-k} \cdot \Pr[E_{h-1}^{h-1}] + \sum_{j=1}^{h-2} 2^{-k-1} \cdot \Pr[E_j^{h-1}] \\
&\qquad - 2^{-1} \cdot \Pr[E_0^h|E_0^{h-1}] + \sum_{j=1}^{h-2} 2^{-j-1} \Pr[E_j^{h-1}] \\
&\leq \Pr[E_0^h|E_0^{h-1}] \cdot (\Pr[E_0^{h-1}] - 2^{-1}) + 2^{-k-1} \cdot \Pr[E_{h-1}^{h-1}] + 2^{-k-1} + \\
&\qquad \sum_{j=1}^{h-2} 2^{-j-1} \Pr[E_j^{h-1}] \\
&\leq 2^{-k} + \sum_{j=1}^{h-2} 2^{-j-1} \Pr[E_j^{h-1}].
\end{aligned}$$

In the first inequality, we have used that $\Pr[E_0^h] = \Pr[E_0^h \wedge E_0^{h-1}]$ and that $\Pr[E_j^h | E_j^{h-1}] \leq 2^{-k+j}$ for all $0 \leq j \leq h-1$. The latter can be seen as follows: $\{f(\mathbf{v}_g)\}_{g\in[h]}$ having $\mathbb{F}_2$-dimension j implies that $f(\mathbf{v}_h) + \sum_{g\in G} f(\mathbf{v}_g) = 0$ for any set $G \subset [h-1]$ of size j. This is equivalent to $\sum_{i=1}^N \mathbf{x}_i \cdot f_i(\mathbf{v}_h + \sum_{g\in G} \mathbf{v}_g) = 0$. By Lemma 6, this happens with probability 2^{-k}. Given that $\{f(\mathbf{v}_g)\}_{g\in[h-1]}$ has $\mathbb{F}_2$-dimension j, there are 2^j possibilities of for $\sum_{g\in G} f(\mathbf{v}_g)$ over all choices of $G \subset [h-1]$. Summing up gives the desired inequality. In the second inequality, we have used that $\sum_{j=1}^{h-1} \Pr[E_j^{h-1}] \leq 1$, and in the third inequality, we have used that $\Pr[E_0^{h-1}] \leq 2^{-k}$ for $h > 1$.

Finally, consider that

$$\Pr[F_{h-1}] = \sum_{j=1}^{h-2} 2^{-j} \cdot \Pr[E_j^{h-1}]$$

like in (2). We conclude that

$$\begin{aligned}
\Pr[F_h] &\leq 2^{-k} + \sum_{j=1}^{h-2} 2^{-j-1} \cdot \Pr[E_j^{h-1}] \\
&= 2^{-k} + 2^{-1} \cdot \Pr[F_{h-1}] \\
&\leq 2^{-k} + 2^{-1} \cdot 2^{-k+1} \\
&= 2^{-k+1},
\end{aligned}$$

which completes the induction.

Corollary 1. *For $N_1, N_2, \ell \in \mathbb{N}$, let $f_{i,j} : \mathbb{F}_2^\ell \to \mathbb{F}_{2^k}$ be a $\mathbb{F}_2$-linear map for all $(i,j) \in [N_1] \times [N_2]$, $\{\mathbf{x}_i\}_{i\in[N]}$ be independent and uniformly distributed in $\mathbb{F}_{2^k}$, $\mathbf{w}_j$ be either non-zero or uniformly distributed in $\mathbb{F}_{2^k}$ for all $j \in [N_2]$, and $f = \sum_{i=1}^N \mathbf{x}_i \cdot \mathbf{w}_j \cdot f_{i,j}$. Furthermore, let $\mathbf{y}$ be independent and uniformly distributed in $\mathbb{F}^\ell$, and $\mathbf{z}$ a random variable independent of $\mathbf{y}$ (but not necessarily of $\{\mathbf{x}_i\}_{i\in[N_1]}$ or $\{\mathbf{w}_j\}_{j\in[N_2]}$). Then,*

$$\Pr\left[\mathbf{z} = f(\mathbf{y}) \wedge \exists i \in [N_1], j \in [N_2], \mathbf{y}' \in \mathbb{F}_2^\ell : (f(\mathbf{y}) = f(\mathbf{y}') \wedge f_{i,j}(\mathbf{y}) \neq f_{i,j}(\mathbf{y}'))\right] \leq 2^{-k+1}.$$

In other words, the probability that $f(\mathbf{y}) = \mathbf{z}$ and $f(\mathbf{y})$ does not uniquely determine $f_{i,j}(\mathbf{y})$ for all $(i,j) \in [N_1] \times [N_2]$ is at most 2^{-k+1}.

Proof. Similarly to the previous lemma, we establish that $\Pr[F_1] \leq 2^{-k+1}$ using a simplified version of Lemma 7. The rest of the proof is identical.

F.1 Amplified Correlated OT – Lemma 1

In the simulator in Figure 22, $\mathcal{I}$ describes P_R's bits that the adversary tries to tamper with in the OT. If $\mathcal{I}$ is small enough, the amplification rules out any leakage with overwhelming probability in k. On the other hand, if $|\mathcal{I}|$ is larger than k, the amplified result contains so many errors that the adversary has only negligible chance of opening all of them correctly.

Proof. It is easy to see that if both parties follow the protocol,

$$Q' + T' = D_{\mathbf{x}'} Y = D_{\mathbf{x}'} \mathcal{O} D_{\mathbf{y}} = \mathbf{x}' \otimes \mathbf{y}.$$

Hence,

$$\begin{aligned}
Q + T &= M(Q' + T') + \delta \otimes \mathbf{y} \\
&= M(\mathbf{x}' \otimes \mathbf{y}) + \delta \otimes \mathbf{y} \\
&= M\mathbf{x}' \otimes \mathbf{y} + \delta \otimes \mathbf{y}
\end{aligned}$$

1. Get Y and Q' from P_S by emulating $\mathcal{F}_{\mathsf{OT}}^{k,\ell'}$ for $\ell' = 2k + s$.
2. Define $\mathbf{y}$ as the majority of rows in Y.
3. Define $Y' = Y - \mathcal{O}D_{\mathbf{y}}$ where is $\mathcal{O} \in \mathbb{F}_2^{\ell' \times k}$ is the matrix full of ones.
4. Define $\mathcal{I}$ as the set of indices where the i-th row of Y' is non-zero.
   - If $|\mathcal{I}| > k$:
     (a) Input $(\mathsf{MultError}, Y')$ to $\mathcal{F}_{\mathsf{ACOT}}^{k,s}$ and receive $(\hat{M}, \hat{\delta})$.
     (b) Compute $Q = \hat{M}Q' + \hat{\delta} \otimes \mathbf{x}$.
   - Otherwise:
     (a) Sample $\tilde{M} \xleftarrow{\$} \mathbb{F}_2^{k \times \ell'}$, $\tilde{\delta} \xleftarrow{\$} \mathbb{F}_2^k$, and $\tilde{\mathbf{x}}' \xleftarrow{\$} \mathbb{F}_2^{\ell'}$.
     (b) Compute $E = \tilde{M}D_{\tilde{\mathbf{x}}'}Y'$ and $Q = \tilde{M}Q' + \tilde{\delta} \otimes \mathbf{y}$.
     (c) Send $(\mathsf{AddError}, E)$ to $\mathcal{F}_{\mathsf{ACOT}}^{k,s}$.

Fig. 22. Simulator for amplified leaky correlated OT (corrupted P_S)

$$\begin{aligned} &= (M\mathbf{x}' + M\mathbf{x}' + \mathbf{x}) \otimes \mathbf{y} \\ &= \mathbf{x} \otimes \mathbf{y}. \end{aligned}$$

The case of P_R being corrupted is straightforward. P_R can only deviate when sending δ, so we define $\mathbf{x} = \delta + M\mathbf{x}'$ and input it to $\mathcal{F}_{\mathsf{ACOT}}^{k,s}$. For a corrupted P_S, we use the simulator in Figure 22. In the real world, we have that

$$\begin{aligned} T + Q &= M(T' + Q') + \delta \otimes \mathbf{y} \\ &= MD_{\mathbf{x}'}Y + \delta \otimes \mathbf{y} \\ &= M(D_{\mathbf{x}'}(\mathcal{O}D_{\mathbf{y}} + Y')) + \delta \otimes \mathbf{y} \\ &= \mathbf{x} \otimes \mathbf{y} + MD_{\mathbf{x}'}Y'. \end{aligned}$$

We have applied that $MD_{\mathbf{x}'}\mathcal{O}D_{\mathbf{y}} + \delta \otimes \mathbf{y} = \mathbf{x} \otimes \mathbf{y}$. Hence, we have to prove that E added by $\mathcal{F}_{\mathsf{ACOT}}^{k,s}$ to the output is statistically close to $MD_{\mathbf{x}'}Y'$ added in the real protocol.

- If $|\mathcal{I}| \leq k$, $E = \tilde{M}D_{\tilde{\mathbf{x}}'}Y'$. While this looks similar to the real world, note that $\delta = M\mathbf{x}'$ in the real world, while $\tilde{\delta}$ is uniformly random. Define $\mathbf{x}'_{\mathcal{I}}$ as $\mathbf{x}'$ with all indices not in $\mathcal{I}$ set to zero and $\mathbf{x}'_{\bar{\mathcal{I}}} = \mathbf{x}' - \mathbf{x}'_{\mathcal{I}}$. Furthermore, define $M|_{\bar{\mathcal{I}}}$ as M restricted to column indices in $\mathcal{I}$. Clearly, $M\mathbf{x}' = M\mathbf{x}'_{\mathcal{I}} + M\mathbf{x}'_{\bar{\mathcal{I}}}$ and $D_{\mathbf{x}'}Y' = D_{\mathbf{x}'_{\mathcal{I}}}Y'$ by definition. Then, for uniformly random $\mathbf{x}'_{\bar{\mathcal{I}}}$, $M\mathbf{x}'_{\bar{\mathcal{I}}}$ is uniformly distributed if $M|_{\bar{\mathcal{I}}}$ has full rank. By Lemma 5, this happens with probability at least $1 - 2^{-s}$ because $M|_{\bar{\mathcal{I}}}$ has at least $k + s$ columns. If $M\mathbf{x}'_{\bar{\mathcal{I}}}$ is uniformly random (even given M), however, $M\mathbf{x}'$ is independent of $D_{\mathbf{x}'}Y'$ and $MD_{\mathbf{x}'}Y'$.
  It is easy to see that (M, δ) has the same distribution as $(\tilde{M}, \tilde{\delta})$. M and $\tilde{M}$ are sampled uniformly at random and so is $\tilde{\delta}$. Since $\delta = M\mathbf{x}'$ for uniformly random $\mathbf{x}'$, δ is uniformly distributed if M has full rank, which trivially holds if $M|_{\bar{\mathcal{I}}}$ does so. Hence, the statistical distance between $(MD_{\mathbf{x}'}Y', M, \delta)$ and $(\tilde{M}D_{\tilde{\mathbf{x}}'}Y', \tilde{M}, \tilde{\delta})$ is at most 2^{-s}.
- Otherwise, $(\hat{M}D_{\hat{\mathbf{x}}'}Y', \hat{M}, \hat{\delta})$ and $(MD_{\mathbf{x}'}Y', M, \delta)$ are computed in the exact same way and thus perfectly indistinguishable.

We conclude that the statistical distance between simulation and real execution is at most 2^{-s}. □

F.2 Authentication – Lemma 2

Proof. We prove the lemma for authenticating elements of $\mathbb{F}_2$ over the extension field $\mathbb{F}_{2^\kappa}$, i.e. $M = \kappa$. The generalization to higher order fields and authenticating extension field elements is straightforward. Let $\mathcal{S}$ be a simulator that has access to $\mathcal{F}_{[\![\cdot]\!]}$, we show that no environment $\mathcal{Z}$ can distinguish between an interaction with $\mathcal{S}$ and $\mathcal{F}_{[\![\cdot]\!]}$ and an interaction with the real adversary $\mathcal{A}$ and real parties. The simulator invokes an internal copy of $\mathcal{A}$ and sets dummy parties $\pi_i, i \in \mathcal{P}$. Let A be the set of corrupt parties, it proceeds as follows:

1. Simulating the *Initialize phase*: $\mathcal{S}$ samples random shares $\{\Delta^{(k)}\}_{k\notin A}$ for honest π_k and receives $\{\Delta^{(j)}\}_{j\in A}$ internally from $\mathcal{A}$. Then it runs an internal copy of $\mathcal{F}_{\mathsf{COTe}}$.Extend initializing the inputs of the honest (dummy) parties at random, with the inputs of the corrupt parties specified by $\mathcal{A}$ as before. After this it performs the local computation as specified in the protocol. If it receives Abort, then it forwards Abort to the functionality, and it halts. Otherwise inputs (Authenticate) to the functionality, together with the set A of corrupt parties and all the extracted shares of P_i, $i \in A$.
2. Simulating the *n-Share phase*: $\mathcal{S}$ runs an internal copy of $\mathcal{F}_{\mathsf{COTe}}$.Extend as specified before. If P_i inputs vectors instead of bits, the simulator sends $(\mathsf{Error}, \mathbf{e}_h^{(k)})$ to $\mathcal{F}_{[\![\cdot]\!]}$ and sets the flag $\mathsf{Error_k}$ to true. It outputs what corrupts π_i outputs and it halts.

Now we argue indistinguishability.

If during the internal execution of the protocol an abort occurred, then an abort occurs in both the ideal and the real world, and the simulation in this case is perfect.

For the sake of simplicity, we separately consider the two cases $(\neg\mathsf{Abort})\wedge(\neg\mathsf{Error_k})$ and $(\neg\mathsf{Abort})\wedge(\mathsf{Error_k})$. In the first situation, during the Initialize phase, we could have a faulty representation $\mathbf{m}_h = x_h \cdot \Delta + \sum_{k\notin A} x_h^{(k)} \cdot \delta^{(i)}$. We must examine the probability that the Π_{MACCheck} passes when a corrupt party P_i has submitted different $\Delta^{(i,j_1)}, \Delta^{(i,j_2)}$ for some $j_1 \neq j_2$. Assuming $\delta^{(i)} \neq 0$ for at least one i, then for every h, the adversary must adjust their shares $\mathbf{m}_h^{(i)}$ so that $\sum_{i=1}^n \mathbf{m}_h^{(i)} = x_h \cdot \Delta$, for the MAC check to pass. This is easily seen to be equivalent to guessing $\sum_{i\notin A} x_h^{(i)}$ for $h = 1, \ldots, \kappa$, which happens with probability $2^{-\kappa}$, since $x_h^{(i)}$ are uniformly random bits. So if the MAC check passes, we are guaranteed that corrupt P_i used the same MAC key $\Delta^{(i)}$ for each $\mathcal{F}_{\mathsf{COTe}}$ instance, except with negligible probability.
We need to examine the n-Share phase. Since $\mathsf{Error_k}$ is not true, then parties run $\mathcal{F}_{\mathsf{COTe}}$ inputting bits, and indistinguishability of the outputs follows from correctness and privacy of $\mathcal{F}_{\mathsf{COTe}}$.

Let us consider the second case. As before, since there is no interaction among parties, we only need to show outputs indistinguishability. In the real execution we could have that some shares $\{\mathbf{m}_h^{(k)}\}$ are shifted by the value $\mathbf{e}_h^{(i,k)} * \Delta^{(k)}$, that is exactly what the simulator tells to output to the functionality. Note that the flag $\mathsf{Error_k}$ can be set to true also during the Initialize phase, when parties run n-Share on dummy input $x_1, \ldots, x_\kappa$. Now imagine that $\mathbf{m}_h = x_h \cdot \Delta + \sum_{k\notin A} \mathbf{e}_h^{(i,k)} * \Delta^{(k)}$, let $S_h^{(i,k)}$ be the set of indices where $\mathbf{e}_h^{(i,k)}$ is 1 and set $S = \cup_{i\in A} S_h^{(i,k)}$. In the ideal world, the simulator can query the functionality with a description of an affine space $S \subset \mathbb{F}_2^\kappa$. In both the executions the adversary may guess $|S|$ bit of $\Delta^{(k)}$ with the same probability $2^{-|S|}$.

□

F.3 Generalized Bucket Sacrificing for $\mathbb{F}_2$ Triples

In this section we generalize the bucket-based sacrificing step by Larraia et al. [16], abstracting away the key properties that are required. This means the procedure could be reused for other purposes, such as checking different kinds of triples or preprocessing data. We also obtain a tighter proof than Larraia et al., allowing the bucket size to be just 3 when checking batches of 1 million triples, whereas previously this required bucket size 4.

Definition 2. *We say that a data type is (R, p)-checkable if:*

- *Values of the data type consist of a fixed-length list of secret shared and MACed values.*
- *There is a predicate R that is efficiently computable if the data is revealed.*
- *There is an algorithm* CheckR *and associated value p that takes as input two items a, b and outputs either* Good *or* Bad*, such that:*
  - *If $R(a) = 1$ and $R(b) = 1$ then* $\mathsf{CheckR}(a, b) = \mathsf{Good}$.
  - *If $R(a) = 0$ and $R(b) = 0$ then* $\Pr[\mathsf{CheckR}(a, b) = \mathsf{Good}] \leq p$.
  - *If $R(a) = 0, R(b) = 1$ or $R(a) = 1, R(b) = 0$ then* $\mathsf{CheckR}(a, b) = \mathsf{Bad}$

Subprotocol $\Pi_{\mathsf{CheckTriples}}$

The protocol is parametrized by the number ℓ of triples generated, the size T and T' of buckets, and a positive integer c which is used to control how much cut-and-choose we perform.

Input: Let $N = T(T'\ell) + c$ be the number of inputs denoted by $\{\mathsf{aT}_i\}_{i\in[N]}$, where $\mathsf{aT}_i = \{[\![\mathbf{x}_i]\!], [\![\mathbf{y}_i]\!], [\![\mathbf{z}_i]\!]\}$

Phase-I *Cut-and-choose*:

1. Using $\mathcal{F}_{\mathsf{Rand}}$, the parties sample a random vector $\mathbf{v} \in \mathbb{F}_2^N$ with c of non-zero entries.
2. Let $\mathcal{J}$ be the set of indices j such that $v_j \neq 0$ and $\forall j \in \mathcal{J}$. The parties partially open $\{\mathsf{aT}_j\}_{j\in\mathcal{J}}$ and $\forall j \in \mathcal{J}$, verify the predicate $R(\mathsf{aT}_j)$. If $R = 0$ happens, then Abort.

Phase-II *Bucket-Sacrifice*: After Phase I we are left with $T(T'\ell)$ aTs. Set $t = T'\ell$.

1. Permute each of the unopened items according to a random permutation π on Tt indices, again using $\mathcal{F}_{\mathsf{Rand}}$. Then renumber the permuted unopened sets of values such that $j = 1, \ldots, Tt$ and, for $i = 1, \ldots, t$ create the i'th bucket as $\{\mathsf{aT}_j\}_{j=iT-T+1}^{iT}$.
2. Parties compute a $\mathsf{BucketHead}(i)$ for each $i = 1, \ldots, t$, i.e. return the first (lexicographically) element in the i'th bucket.
3. For $i = 1, \ldots, t$, check the correctness of $\mathsf{BucketHead}(i)$ against every other item in the bucket. That is, for $j = iT - T + 2, \ldots, iT$:
   - Run $\mathsf{CheckR}(\mathsf{BucketHead}(i), \mathsf{aT}_j)$
   - If CheckR outputs Bad, then Abort.

Phase-III *Combine*: After Phase I and II, there are $t = T'\ell$ authenticated triples.

1. Repeat Step 1. of Phase II to permute the remaining aT_i and split them into ℓ buckets of size T'.
2. Recursively combine the triples in each bucket as follows:
   a) Consider the first two elements in the bucket, say aT_i and aT_j
   b) Parties open $\langle \mathbf{y}_i + \mathbf{y}_j \rangle$
   c) Parties set: $[\![\mathbf{x}_k]\!] = [\![\mathbf{x}_i + \mathbf{x}_j]\!], [\![\mathbf{y}_k]\!] = [\![\mathbf{y}_i]\!]$ and $[\![\mathbf{z}_k]\!] = [\![\mathbf{z}_i + \mathbf{z}_j]\!] + (\mathbf{y}_i + \mathbf{y}_j)[\![\mathbf{x}_j]\!]$.
   d) Then repeat Steps b) and c) with aT_k and the next element in the bucket.

Phase-IV *Mac Check*:

1. The parties execute the protocol Π_{MACCheck} on all partially opened values from Phase I, II and III.
2. If Π_{MACCheck} does not abort then output the resulting combined triples bucket aT_k for $k = 1, \ldots, \ell$ as correct authenticated triples.

Fig. 23. Checking triples

- *If* $\mathsf{CheckR}(a, b) = \mathsf{Good}$ *and* b *is discarded then no information about the secret shared data* a *is revealed.*

Here we want to verify that an authenticated triple, $\mathsf{aT} = \{[\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{z}]\!]\}$, where $\mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathbb{F}$, verifies the multiplicative relation. This means that R is true, *i.e.* $R(\mathsf{aT}) = 1$, if and only if the opened values $\mathbf{x}, \mathbf{y}, \mathbf{z}$ satisfy $\mathbf{x} \cdot \mathbf{y} = \mathbf{z}$. Note that if p is negligible in the statistical security parameter then we can check the relation R by simple pairwise sacrificing, that here we denote by $\mathsf{CheckR}(\cdot, \cdot)$, as with the main methods of Damård et al. [10,8]. If p is non-negligible, we do the bucket-based sacrifice given in Figure 23.

Lemma 9. *Let* $t = \ell T'$*. For an* (R, p)*-checkable data type, the sacrifice procedure, e.g. (Phase* I *and* II *in* $\Pi_{\mathsf{CheckTriples}}$*), with* $c = 3\log_2 t$ *and* $T \geq \frac{s}{\log_2 t - \log_2 p} + 1$ *outputs* t *values satisfying* R *with error probability* 2^{-s}*.*

Proof. Correctness in the semi-honest model is straightforward. For active security, first note that by the properties of the CheckR procedure, the only way an incorrect tuple evades detection is if it is not opened in Phase I and then if *all* of the tuples in its bucket in Phase II are incorrect, in which case the test passes with probability p^{T-1}.

Let $m > 0$ be the number of initial tuples that do not satisfy R. If $m > T \cdot t$ then the check in Phase I will never succeed, and similarly if $m \bmod T \neq 0$ then the checks in Phase II will never succeed. Furthermore, note that for the check to pass m must be a multiple of k, otherwise there will always be at least one bucket

with both correct and incorrect triples. So let $m = k \cdot T$ for some $k \in \{1, \ldots, t\}$. As in Larraia et al. [16], let E_1 denote the event that the cut-and-choose step (Phase I) passes, and E_2 the event that the sacrificing step (phase II) also passes. The cut-and-choose step passes only if none of the c triples chosen to be checked are incorrect, and since m out of all the N triples are incorrect, we have:

$$\Pr[E_1] = \frac{N-m}{N} \cdot \frac{N-m-1}{N-1} \cdots \frac{N-m-c+1}{N-c+1}.$$

It is easy to see that:

$$\Pr[E_2] = \Pr[E_1] \cdot p^{k(T-1)} \cdot \binom{t}{k} \cdot \binom{tT}{kT}^{-1}.$$

Note that the quantities $\Pr[E_1]$ and $p^{k(T-1)}$ are less than one and strictly decrease as k increases. Let $\rho(k) := \binom{t}{k} \cdot \binom{tT}{kT}^{-1}$, the right-hand term in the above expression, and notice that this is symmetric around $k = t/2$, i.e. $\rho(k) = \rho(t-k)$, and is minimized at $k = t/2$. Therefore for $k \in \{1, \ldots, t\}$, $\Pr[E_2]$ is maximized either at $k = 1$ (when $\rho(k) = \rho(1) = \rho(t-1)$) or $k = t$ (when $\rho(k) = \rho(0)$). We now examine the choices for c that lead to $\Pr[E_2]$ being maximal at $k = 1$. If this is the case, we have:

$$\begin{aligned}
&\Pr[E_2]|_{k=1} \geq \Pr[E_2]|_{k=t} \\
\Leftrightarrow &\frac{N-T}{N} \cdots \frac{N-T-c+1}{N-c+1} \cdot p^{T-1} \cdot t \cdot \binom{Tt}{T}^{-1} \geq \frac{N-tT}{N} \cdots \frac{N-tT-c+1}{N-c+1} \cdot p^{t(T-1)} \\
\Leftrightarrow &(N-T) \cdots (N-T-c+1) \cdot t \cdot \frac{T!(T(t-1))!}{(Tt)!} \geq (N-tT) \cdots (N-tT-c+1) \cdot p^{(t-1)(T-1)} \\
\Leftrightarrow &(N-T) \cdots (N-c+1) \cdot t \cdot T! \geq c! \cdot p^{(t-1)(T-1)}
\end{aligned}$$

Recall that $p \leq 1$, and notice that if $c \geq T$ and $N - c + 1 \geq T + 1$ then the above equation is always true, so we now impose these constraints on c and continue to inspect $\Pr[E_2]$, assuming $k = 1$.

$$\begin{aligned}
\Pr[E_2] &= \frac{N-T}{N} \cdots \frac{N-T-c+1}{N-c+1} \cdot t \cdot \binom{Tt}{T}^{-1} \cdot p^{T-1} \\
&\leq t \cdot \binom{Tt}{T}^{-1} \cdot p^{T-1} \\
&\leq t^{1-T} \cdot p^{T-1} \\
&= t^{1-T} \cdot p^{T-1} \\
&= \left(\frac{t}{p}\right)^{1-T} \\
&= 2^{-(T-1)(\log_2 t - \log_2 p)}
\end{aligned}$$

so choosing $T \geq \frac{s}{\log_2 t - \log_2 p} + 1$ ensures that $\Pr[T_2] \leq 2^{-s}$. Since the last inequality above is independent of the cut-and-choose parameter c, it suffices to set $c = T$, meaning the amount of cut-and-choose performed is essentially negligible. □

The following Lemma shows that the combining step in the triple checking protocol suffices to remove leakage from our $\mathbb{F}_2$ triples.

Lemma 10. *Phase* III *in* $\Pi_{\mathsf{CheckTriples}}$ *outputs leaky triples with negligible probability* $\leq 2^{-s}$ *if* $T' > \frac{s-1}{\log_2(\ell)} + 1$.

# triples	s	T, T'
1024	40	5
16384	40	4
1048576	40	3
1024	64	8
16384	64	6
1048576	64	5

Table 2. Bucket sizes for $\mathbb{F}_2$ triple checking and combining with statistical security parameter s.

Proof. Straightforward from Theorem 8 by Nielsen et al. [18]. □

For the case of bit triples, the bucket checking procedure has detection probability $p = 1$ when a bucket contains both good and bad triples. This means we have the following requirements:

F.4 Bit Triples – Theorem 2

Let $\mathcal{A}$ be a real world adversary corrupting up to $n-1$ parties, we describe an ideal world adversary $\mathcal{S}$ for $\mathcal{A}$. $\mathcal{S}$ has access to $\mathcal{F}_{\mathsf{Triples}}$, and we show that no environment $\mathcal{Z}$ can distinguish between its interaction with $\mathcal{S}$ and $\mathcal{F}_{\mathsf{Triples}}$ and its interaction with the real adversary $\mathcal{A}$ and real parties. The ideal world adversary $\mathcal{S}$ internally runs a copy of $\mathcal{A}$ setting dummy parties $\pi_1, \ldots, \pi_n$, and then it simulates for them a real execution of $\Pi_{\mathsf{BitTriples}}$. The communication of $\mathcal{Z}$ with the adversary $\mathcal{A}$ is handled as follows: every input value received by the simulator from $\mathcal{Z}$ is written on $\mathcal{A}$'s input tape. Likewise, every output value written by $\mathcal{A}$ on its output tape is copied to the simulator's output tape (to be read by the environment $\mathcal{Z}$).

$\mathcal{S}$ simulates an internal copy of $\Pi_{\mathsf{BitTriples}}$ as follows.

1. Simulating the *Initialize phase*: It samples random shares $\{\hat{\Delta}^{(k)}\}_{k\notin A}$ for honest π_k and receives $\{\hat{\Delta}^{(j)}\}_{j\in A}$ internally from $\mathcal{A}$. Then it emulates $\mathcal{F}_{\mathsf{COTe}}$ and $\mathcal{F}_{[\![\cdot]\!]}.\mathsf{Init}$ to check the consistency of $\hat{\Delta}$ and Δ, with the input of the honest (dummy) parties sampled at random, and with the input of corrupt parties specified by $\mathcal{A}$. Then it proceeds as in the protocol performing local computations. If it receives Abort, then it forwards Abort to the functionality, and it halts. Otherwise, it inputs $\mathsf{Triples}$, together with the set A of corrupt parties and all the extracted shares for $P_i, i \in A$.
2. Simulating the COTe *phase*: The simulator runs internal copies of $\mathcal{F}_{\mathsf{COTe}}^{2\kappa,\ell}$. It sends $\{\hat{\mathbf{t}}_h^{(j,k)}, \hat{\mathbf{q}}_h^{(j,k)}\}_{j\in A}$ to $\mathcal{A}$. If $\mathcal{A}$ inputs vectors instead of bits to some π_k, $k \notin A$, then it sets the flag $\mathsf{MacError}$ to true and computes $\mathbf{e} = \sum_{k\notin A} e_h^{(i,k)} \cdot \Delta^{(k)}$, for some $i \in A$. It sends $(\mathsf{MacError}, \mathbf{e})$ to the functionality.
3. Simulating the *Triple generation phase*: For $j \in A$, $\mathcal{S}$ receives $s_h^{(j,i)}$ from $\mathcal{A}$ and sets $y_h^{(j)} = s_h^{(j,i)} + v_{0,h}^{(j,i)} + v_{1,h}^{(j,i)}$. $\mathcal{S}$ internally sends random $\{s_h^{(k,i)}\}_{k\notin A}$ to $\mathcal{A}$ acting as $\mathcal{F}_{\mathsf{AT}}$. If $\mathcal{A}$ gives any inconsistent $y_h^{(j)}$ for $j \in A$, or an inconsistent $n_h^{(i,j)}$, $i \in A$, $\mathcal{S}$ sets flag $\mathsf{badTriples}$ to true.
4. Simulating the *Authentication phase*: If during the pairwise calls of $\mathcal{F}_{\mathsf{COTe}}$, $\mathcal{A}$ misbehaves in such a way that dummy parties $\{\pi_1, \ldots, \pi_n\}$ hold incorrect authenticated bits, then the simulator sets the flag $\mathsf{badAuth}$ to true; if $\mathcal{A}$ inputs vectors instead of bits set the flag $\mathsf{MacError}$ to true and compute the subset $S_h^{(k)} \subseteq \{1, \ldots, \kappa\}$ and $\mathbf{e} = \sum_{k\notin A} e_h^{(i,k)} \cdot \Delta^{(k)}$, for some $i \in A$. Hence it sends $(\mathsf{MacError}, \mathbf{e})$ to the functionality. Also, if $\mathcal{A}$ authenticates something different to what it was generated in the Triple generation step, then $\mathcal{S}$ sets flag $\mathsf{badTriples}$ to true.
5. Simulating the *Check triples phase*: In the **MacCheck** step the simulator uses the dummy input $\{\Delta^{(k)}\}_{k\notin A}$ to run Π_{MACCheck} with $\mathcal{A}$.

Now the simulator checks what happened during the internal execution of the protocol.

If during this execution an abort occurred, the simulator sends Abort to the functionality; both the real and the ideal process abort and the simulation is perfect.

Otherwise, it sends $\{\Delta^{(j)}\}_{j\in A}$ and Triples to the functionality together with all the extracted corrupted shares. Notice that the simulator only sends the last κ components $\Delta^{(j)}$ of $\hat{\Delta^{(j)}}$ to the functionality.
Now, if the flag badAuth is true and no Abort occurred, the simulator acknowledges itself to the environment. In this case $\mathcal{Z}$ always distinguishes between the two executions of the protocol: we have corrected authenticated triples in the ideal world and bad authenticated values in the real execution. By Theorem 3 we know that this happens with negligible probability $2^{-\kappa+1}$.

Consider the case badTriples true and again no Abort occurred. This happens either if some $y_h^{(j)}$ are inconsistent, for $j \in A$, or if corrupted parties authenticate something different to what it was generated in the triple generation phase. In both cases, in a real execution of the protocol, this will produce a faulty representation of type $z_h = x_h \cdot y_h + x_h^{(k)}$, for some $k \in A$. However if no abort occurred, from Lemma 9 and Lemma 10, we know that the authenticated triples that the protocol outputs are correct (i.e. satisfy the multiplicative relation), and that the privacy on x_h is guaranteed with overwhelming probability. On the other hand, if the adversary produces inconsistent $n_h^{(i,j)}$, for some $i \in A$, then the triple will pass CheckR if and only if $s_h^{(j,i)} = 0$ in both executions, and the privacy is again guaranteed as $s_h^{(j,i)}$ is already part of the public transcript. Both the executions output random authenticated triples and hence they are indistinguishable.

Consider now the case when MacError is true. This means that, for some $j \in A$ and $k \notin A$, we have $\mathbf{q}_h^{(k,j)} = \mathbf{t}_h^{(j,k)} + x_h \cdot \Delta + \mathbf{e}_h^{(j,k)} * \Delta^{(k)}$. This faulty representation could happen both before the triples generation step and during the authentication. First we argue indistinguishability of the transcripts: the values $s_h^{k,j}$ are identically distributed in both the executions, as the value $v_0^{(k,j)} + v_1^{(k,j)}$, where $v_0^{(k,j)} = H(\mathbf{q}_h^{(k,j)} + \mathbf{e}_h^{(j,k)} * \Delta^{(k)})$ and $v_1^{(k,j)} = H(\mathbf{q}_h^{(k,j)} + \mathbf{e}_h^{(j,k)} * \Delta^{(k)} + \Delta^{(k)})$, is uniformly random and independent of $w_h^{(i,j)}$, and these values perfectly mask the value $y_h^{(j)}$. If the protocol outputs the triple, we know that it is correct and that privacy on x_h is guaranteed, except with negligible probability. However we could have a leaky MAC representation: the simulator can query the functionality with a description of an affine subspace $S \subset (\mathbb{F}_2^\kappa)^n$, and in both the executions, and the adversary might guess $c = |S_h^{(k)}|$ bits of the global key with probability 2^{-c} .

Finally, if no corruptions occur, we show that $\mathcal{Z}$ does not distinguish between the real and the ideal process. In both processes $\mathcal{Z}$ can see the masks $s^{(j,i)}$ leaked by $\mathcal{F}_{\mathsf{AT}}$: they look perfectly random and independent of $\mathcal{Z}$'s view, as the values $H(\mathbf{q}_h^{(j,i)})$ and $H(\mathbf{q}_h^{(j,i)} + \Delta^{(j)})$ (with H modeled as a random oracle) used to pair $y^{(j)}$ are uniformly random, even if we allow $\mathcal{Z}$ to adaptively make additional calls to H. All the partial openings in both the simulated and the real run of the protocol reveal uniform values. More precisely, all opened values are a combination of output data and sacrificed data, with the latter that is not part of the final output, and therefore by no means $\mathcal{Z}$ can reconstruct the set of opened values using its view. Finally note that we are using the same call to $\mathcal{F}_{\mathsf{COTe}}$ "twice", namely to generate triples and to authenticate x_h. However the z_h's share are obtained by using only the first κ components of the ℓ outputs of $\mathcal{F}_{\mathsf{COTe}}^{2\kappa,\ell}$, and these values, that are hidden from $\mathcal{Z}$'s view, are then randomized again through random oracle queries to break the correlation. To authenticate x_h we then use the second block of κ components, which are hence independent of that used in the previous step. The resulting z_h'share are not bind to the values produced in the authentication phase, all the outputs are identically distributed, and all the ideal transcripts are consistent with what $\mathcal{F}_{\mathsf{Triples}}$ outputs. □

F.5 $\mathbb{F}_{2^k}$ Multiplication

In this section we introduce a new functionality $\mathcal{F}_{\mathsf{GFMult}}$, that will be used in the next two sections to prove the security of $\Pi_{\mathsf{TripleCheck}}$. Notice that Π_{GFMult} essentially consists of a call to $\mathcal{F}_{\mathsf{ACOT}}$ plus an additional step at the end.

Notation. For $\ell', k \in \mathbb{N}$, let M be a matrix in $F_2^{k\times\ell'}$, Y' be a matrix in $\mathbb{F}_2^{\ell'\times k}$, and $\mathcal{I} \subset [\ell']$. For $\{\mathbf{m}_i\}_{i\in[\ell']}$ denoting the columns of M, let $M_{\mathcal{I}} \in \mathbb{F}_2^{k\times|\mathcal{I}|}$ and $M_{\overline{\mathcal{I}}} \in \mathbb{F}_2^{k\times(\ell'-|\mathcal{I}|)}$ be the matrices consisting of the columns

Functionality $\mathcal{F}^{k,s}_{\mathsf{GFMult}}$

It involves two parties P_S and P_R, and an ideal adversary $\mathcal{S}$. The procedure can be called repeatedly. Let $\ell' = 2k + s$.

- Upon receiving $\mathbf{x} \in \mathbb{F}_{2^k}$ from P_R and $\mathbf{y} \in \mathbb{F}_{2^k}$ from P_S, the functionality does the following:

Honest parties

- The functionality samples a random element $\mathbf{q} \in \mathbb{F}_{2^k}$. Then it computes $\mathbf{t} = \mathbf{q} + \mathbf{x} \cdot \mathbf{y}$, and it outputs $\mathbf{q}$ to P_S and $\mathbf{t}$ to P_R.

Corrupt parties

- If P_S is corrupted:
  1. The functionality waits for $\mathcal{S}$ to input $\mathbf{q} \in \mathbb{F}_{2^k}$ and one of the following:
     - If $\mathcal{S}$ inputs $(\mathsf{RandomError}, Y, \bar{\mathbf{f}})$ such that $Y \in \mathbb{F}_2^{\ell' \times k}$ and $|\mathcal{I}| > k$ for $\mathcal{I} = \{i \in [\ell'] \mid \mathbf{y}_i \neq 0\}$, the functionality samples $\bar{M} = M_{\mathcal{I}} \in \mathbb{F}_2^{k \times |\mathcal{I}|}$ such that $\hat{M} = M \odot_{\mathcal{I}} Y$ has rank k and $\hat{\mathbf{x}} \xleftarrow{\$} \mathbb{F}_2^{|\mathcal{I}|}$, sets $\mathbf{f} = \hat{M}\hat{\mathbf{x}} + \bar{\mathbf{f}}$ and $\delta = M_{\mathcal{I}}\hat{\mathbf{x}} + \mathbf{x}$, and sends $(\bar{M}, \delta)$ to $\mathcal{S}$.
     - $\mathcal{S}$ inputs $(\mathsf{AddError}, \mathbf{f})$.
  2. It computes $\mathbf{t} = \mathbf{q} + \mathbf{x} \cdot \mathbf{y} + \mathbf{f}$, and sends $\mathbf{t}$ to P_R, $\mathbf{q}$ to P_S.
- If P_R is corrupted, the functionality waits for $\mathcal{S}$ to input $\mathbf{x}$. Then it samples $\mathbf{q} \in \mathbb{F}_{2^k}$ and computes $\mathbf{u} = \mathbf{q} + \mathbf{x} \cdot \mathbf{y}$, outputs $\mathbf{u}$ to $\mathcal{S}$, waits for $\mathcal{S}$ to input $\mathbf{t}$, and outputs $\mathbf{t}$ to P_R and $\mathbf{q}$ to P_S.

Fig. 24. $\mathcal{F}^{k,s}_{\mathsf{GFMult}}$ – Galois field multiplication

1. Emulating $\mathcal{F}^{k,s}_{\mathsf{ACOT}}$, get $\mathbf{y}$ from P_S.
2. Similarly, wait for $\mathcal{Z}$ to input $Q \in \mathbb{F}_2^{k \times k}$ and $(\mathsf{MultError}, Y')$ for $Y' \in \mathbb{F}_2^{\ell' \times k}$.
3. Let $\{\mathbf{y}'_i\}_{i \in \mathcal{I}}$ denote the rows of Y' and define $\mathcal{I} = \{i \in [\ell'] \mid \mathbf{y}'_i \neq 0\}$.
4. Sample $M \xleftarrow{\$} \mathbb{F}_2^{k \times \ell'}$.
   - If $M_{\overline{\mathcal{I}}}$ has rank k, sample $\delta \xleftarrow{\$} \mathbb{F}_{2^k}$ and $\mathbf{f} \xleftarrow{\$} \mathrm{im}(M \odot_{\mathcal{I}} Y')$, and input $(\mathsf{AddError}, \mathbf{f})$ to $\mathcal{F}^{k,s}_{\mathsf{GFMult}}$.
   - Otherwise, if $M \odot_{\mathcal{I}} Y'$ has rank k, sample $\mathbf{x}'_{\overline{\mathcal{I}}} \xleftarrow{\$} \mathbb{F}_2^{|\overline{\mathcal{I}}|}$, input $(\mathsf{RandomError}, Y', M_{\overline{\mathcal{I}}}\mathbf{x}'_{\overline{\mathcal{I}}})$ to $\mathcal{F}^{k,s}_{\mathsf{GFMult}}$, receive $(\hat{M}, \delta)$ from $\mathcal{F}^{k,s}_{\mathsf{GFMult}}$, and replace $M_{\mathcal{I}}$ by $\hat{M}$.
   - Otherwise, abort.
5. Send (M, δ) to $\mathcal{Z}$.
6. Compute $\mathbf{q} = \mathbf{e}Q\mathbf{e}^\top$ and send $\mathbf{q}$ and Y' to $\mathcal{F}^{k}_{\mathsf{GFMult}}$.

Fig. 25. Simulator for $\mathbb{F}_{2^k}$ multiplication (corrupted P_S)

of $\{\mathbf{m}_i\}_{i \in \mathcal{I}}$ and $\{\mathbf{m}_i\}_{i \notin \mathcal{I}}$, respectively. Furthermore, $(M \odot_{\mathcal{I}} Y') \in \mathbb{F}_2^{k \times |\mathcal{I}|}$ be the matrix consisting of the columns $\{\mathbf{y}'_i \cdot \mathbf{m}_i\}_{i \in \mathcal{I}}$.

Lemma 11. *The protocol Π^k_{GFMult}, described in Figure 9, implements $\mathcal{F}^{\kappa,s}_{\mathsf{GFMult}}$ (Fig. 24) in the $\mathcal{F}^{k,s}_{\mathsf{ACOT}}$-hybrid model with statistical security s.*

Proof. If both parties follow the protocol,

$$\mathbf{t} + \mathbf{q} = \mathbf{e}(T + Q)\mathbf{e}^\top = \mathbf{e}(\mathbf{x} \otimes \mathbf{y})\mathbf{e}^\top = (\mathbf{e}\mathbf{x}) \otimes (\mathbf{e}\mathbf{y}) = \mathbf{x} \cdot \mathbf{y}.$$

Note that, for $\mathbf{x} \in \mathbb{F}_2^k$, $\mathbf{e}\mathbf{x}$ denotes the representation of $\mathbf{x}$ in $\mathbb{F}_{2^k}$, and hence, the tensor product collapses to the field product in $\mathbb{F}_{2^k}$ in the last equality.

The case of P_R being corrupt is trivial. For corrupt P_S we use the simulator in Figure 25. On input $(\mathsf{MultError}, Y')$ from $\mathcal{Z}$ to $\mathcal{F}^{k,s}_{\mathsf{ACOT}}$ in the real world, $\mathcal{F}^{k,s}_{\mathsf{ACOT}}$ will sample $M \in \mathbb{F}_2^{k \times \ell'}$ and $\mathbf{x}' \in \mathbb{F}_2^{\ell'}$, compute $\delta = M\mathbf{x}' + \mathbf{x}$, and output (M, δ) to $\mathcal{Z}$. It is easy to see that

$$\mathbf{t} + \mathbf{q} = \mathbf{x} \cdot \mathbf{y} + \mathbf{e}MD_{\mathbf{x}'}Y'\mathbf{e}^\top$$

for $\mathbf{t}$ and $\mathbf{q}$ output by P_R and P_S in Π_{GFMult}. If $\mathbf{m}_i \in \mathbb{F}_{2^k}$ and $\mathbf{y}'_i \in \mathbb{F}_{2^k}$ denote the columns of M and rows of Y', respectively, it holds that

$$\begin{aligned}\mathbf{e}M &= (\mathbf{m}_1, \ldots, \mathbf{m}_{\ell'}) \quad \text{and}\\ Y'\mathbf{e}^\top &= (\mathbf{y}'_1, \ldots, \mathbf{y}'_{\ell'})^\top.\end{aligned}$$

It follows that

$$\begin{aligned}\mathbf{e}MD_{\mathbf{x}'}Y'\mathbf{e}^\top &= \sum_{i=1}^{\ell'} x'_i \cdot \mathbf{m}_i \cdot \mathbf{y}'_i\\ &= \sum_{i\in\mathcal{I}} x'_i \cdot \mathbf{m}_i \cdot \mathbf{y}'_i\\ &= (M \odot_{\mathcal{I}} Y')\mathbf{x}'_{\mathcal{I}}\end{aligned}$$

for $\mathbf{x}'_{\mathcal{I}}$ consisting of the elements of $\mathbf{x}'$ with index in $\mathcal{I}$.

By definition, $M_{\overline{\mathcal{I}}}$ has the same distribution in both worlds. The same holds for $M_{\mathcal{I}}$ even though it is generated twice if $M \odot_{\mathcal{I}} Y'$ has rank k. The second generation is executed to match this condition. We now consider the same distinction for real-world M as in the simulator.

- If $M_{\overline{\mathcal{I}}}$ has rank k, $M_{\overline{\mathcal{I}}}\mathbf{x}'_{\overline{\mathcal{I}}}$ is distributed uniformly in $\mathbb{F}_{2^k}$ for uniformly random $\mathbf{x}'_{\overline{\mathcal{I}}}$ and thus is
$$\delta = M\mathbf{x}' + \mathbf{x} = M_{\mathcal{I}}\mathbf{x}'_{\mathcal{I}} + M_{\overline{\mathcal{I}}}\mathbf{x}'_{\overline{\mathcal{I}}} + \mathbf{x}.$$
$\mathbf{f} \xleftarrow{\$} \mathrm{im}(M \odot_{\mathcal{I}} Y')$ and $\mathbf{e}MD_{\mathbf{x}'}Y'\mathbf{e}^\top = (M \odot_{\mathcal{I}} Y')\mathbf{x}'_{\mathcal{I}}$ are trivially distributed identically because $\mathbf{x}'_{\mathcal{I}}$ and $\mathbf{x}'_{\overline{\mathcal{I}}}$ are independent.
- In the case of $M \odot_{\mathcal{I}} Y'$ having rank k, $\hat{\mathbf{x}}$ is distributed identically to $\mathbf{x}'_{\mathcal{I}}$, and hence $\mathbf{f} = (M \odot_{\mathcal{I}} Y')\hat{vx}$ is so to $(M \odot_{\mathcal{I}} Y')\mathbf{x}_{\mathcal{I}}$ and so is $M_{\mathcal{I}}\hat{\mathbf{x}} + M_{\overline{\mathcal{I}}}\mathbf{x}'_{\overline{\mathcal{I}}} + \mathbf{x}$ to $M\mathbf{x}' + \mathbf{x}$.
- Neither $M_{\overline{\mathcal{I}}}$ nor $M \odot_{\mathcal{I}} Y'$ having rank k happens with probability at most 2^{-s}. According to Lemma 5, it happens with probability $2^{-(\ell'-|\mathcal{I}|-k)}$ for $M_{\overline{\mathcal{I}}}$ and $2^{-(|\mathcal{I}|-k)}$ for either $M_{\mathcal{I}}$ and $M \odot_{\mathcal{I}} Y'$. The latter holds because $\mathbf{m}_i$ is sampled uniformly and $\mathbf{y}_i \neq 0$ for all $i \in \mathcal{I}$, and therefore, $\mathbf{m}_i \cdot \mathbf{y}_i$ is distributed uniformly for all $i \in \mathcal{I}$. Since $M_{\overline{\mathcal{I}}}$ and $(M_{\mathcal{I}}, M \odot_{\mathcal{I}} Y')$ are distributed independently, the probability of neither $M_{\overline{\mathcal{I}}}$ nor both $M_{\mathcal{I}}$ and $M \odot_{\mathcal{I}} Y'$ having full rank is $2^{-(\ell'-|\mathcal{I}|-k)} \cdot 2^{-(|\mathcal{I}|-k)} = 2^{-(\ell'-2k)} = 2^{-s}$ using the union bound.

We conclude that the statistical distance between real and ideal execution is 2^{-s}.

Lemma 12. *Let $M \xleftarrow{\$} \mathbb{F}_2^{k\times\ell'}$ consisting of columns $\{\mathbf{m}_i\}_{i=1}^{\ell'}$, $Y' \in \mathbb{F}_2^{\ell'\times k}$ consisting of rows $\{\mathbf{y}_i\}_{i=1}^{\ell'}$, $\mathcal{I} = \{i \in [\ell'] \mid \mathbf{y}'_i \neq 0\}$ and $M \odot_{\mathcal{I}} Y'$ as above. Furthermore, let f and f' denote the linear map defined by $M_{\mathcal{I}}$ and $(M \odot_{\mathcal{I}} Y')$, respectively. Then, if not all non-zero rows of Y' are the same, the probability that $\ker M_{\mathcal{I}} \subset \ker(M \odot_{\mathcal{I}} Y')$ is at most 2^{-k}.*

Proof. For all $i \in \mathcal{I}$, let $Y_i \in \mathbb{F}_2^{k\times k}$ denote the matrix induced by the multiplication with $\mathbf{y}'_i$ in $\mathbb{F}_{2^k}$. $\ker M_{\mathcal{I}} \subset \ker(M \odot_{\mathcal{I}} Y')$ is equivalent to $(M \odot_{\mathcal{I}} Y') + NM_{\mathcal{I}} = 0$ for some matrix $N \in \mathbb{F}_2^{k\times k}$, which means that, for all $\mathbf{x} \in \mathbb{F}_2^{|\mathcal{I}|}$,

$$\sum_{i\in\mathcal{I}} x_i \cdot (Y_i + N)\mathbf{m}_i = 0,$$

which is equivalent to

$$(Y_i + N)\mathbf{m}_i = 0$$

for all $i \in \mathcal{I}$. By assumption, there exist $j, j' \in \mathcal{I}$ such that $Y_j \neq Y_{j'}$. $Y_j + Y_{j'}$ is invertible because it represents the multiplication with $\mathbf{y}'_j + \mathbf{y}'_{j'}$ and thus has rank k. Let $Y_j + N$ have rank k'. It follows that $Y_{j'} + N$ has rank $k - k'$ because $(Y_j + N) + (Y_{j'} + N) = Y_j + Y'_j$, which has rank k. We conclude that the probability that both $(Y_j + N)\mathbf{m}_j = 0$ and $(Y_{j'} + N)\mathbf{m}_{j'} = 0$ is $2^{-k'}$ and $2^{-k+k'}$, respectively, and hence, the probability of both events is 2^{-k}.

Initialize:

1. Emulating $\mathcal{F}_{[\![\cdot]\!]}$, receive $\{\Delta^{(i)}\}_{i\in A}$ from $\mathcal{Z}$.
2. Input $\{\Delta^{(i)}\}_{i\in A}$ to $\mathcal{F}_{\mathsf{UncheckedTriples}}$.

Triple generation:

1. Emulate all instances of $\mathcal{F}^{k,s}_{\mathsf{GFMult}}$ between honest and corrupted parties.
   - For every corrupt party $i \in A$ and every honest party $j \in B$, define $\mathbf{a}^{(i,j)}$ and $\mathbf{b}^{(i,j)}$ to be the inputs of party P_i to the $\mathcal{F}^{k,s}_{\mathsf{GFMult}}$ instances with party P_j.
   - For every $i \in A$, define $\mathbf{a}^{(i)} = \mathbf{a}^{(i,i)}$ and $\mathbf{b}^{(i)} = \mathbf{b}^{(i,i)}$.
   - For every $j \in B'$, define $\mathbf{f}^{(j)}_{\mathbf{a}} = \sum_{i\in A}(\mathbf{a}^{(i,j)} - \mathbf{a}^{(i)})$ and $\mathbf{f}^{(j)}_{\mathbf{b}} = \sum_{i\in A}(\mathbf{b}^{(i,j)} - \mathbf{b}^{(i)})$.
   - If the adversary inputs $(\mathsf{RandomError}, Y^{(i,j)}, \bar{\mathbf{f}}^{(i,j)})$ to any instance of $\mathcal{F}^{k,s}_{\mathsf{GFMult}}$, define C as the set of all such instances.
   - For all $(i,j) \in B \times A \setminus C$, $\mathcal{A}$ inputs $(\mathsf{AddError}, \bar{\mathbf{f}}^{(i,j)})$ to the respective $\mathcal{F}^{k,s}_{\mathsf{GFMult}}$ instance.
   - Compute $\bar{\mathbf{f}} = \sum_{(i,j)\in B\times A\setminus C} \mathbf{f}^{(i,j)}$.
   - For all $i \in B$ and $j \in A$, in the instance with party i as P_S and party j as P_R, sample $U^{(i,j)} \xleftarrow{\$} \mathbb{F}_2^{k\times k}$ and output it to $\mathcal{Z}$. Wait for $\mathcal{A}$ to input $T^{(i,j)} \in \mathbb{F}_2^{k\times k}$. Set $\bar{\mathbf{f}} \leftarrow \bar{\mathbf{f}} + \sum_{i\in B, j\in A}(\mathbf{t}^{(i,j)} + \mathbf{u}^{(i,j)})$.
2. Emulate $\mathcal{F}^{\mathbb{F}_{2^k}}_{[\![\cdot]\!]}$.
   - If $\mathcal{Z}$ inputs $(\mathsf{Error}, \{e^{(i)}_{1,h}, e^{(i)}_{2,h}, e^{(i)}_{3,h}\}_{k\in B, h\in[k]})$, define $\varphi^{(i)}_{\mathbf{a},h} = e^{(i)}_{1,h}$, $\varphi^{(i)}_{\mathbf{b},h} = e^{(i)}_{2,h}$, and $\varphi^{(i)}_{\mathbf{c},h} = e^{(i)}_{3,h}$ for all $i \in B, h \in [k]$.
3. Wait for $\mathcal{Z}$ to input $\{\mathbf{m}^{(i)}_{\mathbf{a}}, \mathbf{m}^{(i)}_{\mathbf{b}}, \mathbf{m}^{(i)}_{\mathbf{c}}\}_{i\in A}$.
4. Add the cumulative errors in the outputs of corrupted parties to $\bar{\mathbf{f}}, \psi_{\mathbf{a}}, \psi_{\mathbf{b}}, \psi_{\mathbf{c}}$.
5. Input $\{\mathbf{f}^{(i)}_{\mathbf{a}}, \mathbf{f}^{(i)}_{\mathbf{b}}\}_{i\in B'}$, $\{\varphi^{(i)}_{\mathbf{a},h}, \varphi^{(i)}_{\mathbf{b},h}, \varphi^{(i)}_{\mathbf{c},h}\}_{i\in B, h\in[k]}$, $\bar{\mathbf{f}}, \psi_{\mathbf{a}}, \psi_{\mathbf{b}}, \psi_{\mathbf{c}}$ to $\mathcal{F}_{\mathsf{UncheckedTriples}}$, C and $\{(Y^{(i,j)}, \bar{\mathbf{f}}^{(i,j)})\}_{(i,j)\in C}$ to the functionality.
6. Receive $\{(\bar{M}^{(i,j)}, \delta^{(i,j)})\}_{(i,j)\in C}$ and forward $(\bar{M}^{(i,j)}, \delta^{(i,j)})$ to $\mathcal{Z}$ emulating $\mathcal{F}^{k,s}_{\mathsf{GFMult}}$ between party i and j for all $(i,j) \in C$.

Fig. 26. Simulator for $\mathcal{F}_{\mathsf{UncheckedTriples}}$

F.6 SPDZ Triple Generation – Lemma 3

Proof. If all parties behave honestly in $\Pi_{\mathsf{UncheckedTriples}}$ in Figure 10, it holds that

$$\begin{aligned}\sum_{i\in\mathcal{P}} \mathbf{c}^{(i)} &= \sum_{i\in\mathcal{P}} \mathbf{a}^{(i)} \cdot \mathbf{b}^{(i)} + \big(\sum_{j\neq i} \mathbf{a}^{(i)} \cdot \mathbf{b}^{(j)}\big) \\ &= \sum_{i\in\mathcal{P}} \sum_{j\in\mathcal{P}} \mathbf{a}^{(i)} \cdot \mathbf{b}^{(j)} \\ &= \mathbf{a} \cdot \mathbf{b}.\end{aligned}$$

For the case of corrupted parties, we use the simulator in Figure 26. Most aspects of the indistinguishability can be seen with a straightforward computation that merely involves summing up the outputs of all instance of $\mathcal{F}^{k,s}_{\mathsf{GFMult}}$.

F.7 SPDZ Triple Checking – Theorem 1

Proof. We construct a simulator $\mathcal{S}$ for the ideal functionality $\mathcal{F}_{\mathsf{Triples}}$ such that no environment $\mathcal{Z}$ distinguishes with a non-negligible probability whether it is interacting with the $\mathcal{A}$ in the real setting or with $\mathcal{S}$ in the ideal setting.

1. Simulating the *Initialize* phase: Emulating $\mathcal{F}_{\mathsf{UncheckedTriples}}$, $\mathcal{S}$ receives $\{\Delta^{(i)}\}_{i\in A}$ internally from $\mathcal{A}$ and it inputs $\{\Delta^{(i)}\}_{i\in A}$ to $\mathcal{F}_{\mathsf{Triples}}$.
2. Simulating the *Triple generation* phase:

(a) It emulates $\mathcal{F}_{\mathsf{UncheckedTriples}}$ with input of corrupt parties, $\{\mathbf{a}_j^{(i)}, \mathbf{b}_j^{(i)}\}_{i\in A, j\in[2N]}$, specified by $\mathcal{A}$, together with $\{\mathbf{f}_{\mathbf{a}_j}^{(i)}, \mathbf{f}_{\mathbf{b}_j}^{(i)}\}_{i\in B'}$, $\{\varphi_{\mathbf{a}_j,h}^{(i)}, \varphi_{\mathbf{b}_j,h}^{(i)}, \varphi_{\mathbf{c}_j,h}^{(i)}\}_{i\in B, h\in[k]}$, $\bar{\mathbf{f}}_j, \psi_{\mathbf{a}_j}, \psi_{\mathbf{b}_j}, \psi_{\mathbf{c}_j} \in \mathbb{F}_{2^k}$, $C_j \subset B \times A$, and $\{Y_j^{(i,i')}\}_{(i,i')\in C_j}$ for all $j \in [2N]$.

(b) Let $\mathcal{I}_j^{(i,i')} = \{h \in [\ell'] \mid \mathbf{y}_h \neq 0\}$ as in $\mathcal{F}_{\mathsf{GFMult}}^{k,s}$. The functionality samples $\delta_j^{(i,i')} \xleftarrow{\$} \mathbb{F}_{2^k}$ and $\bar{M}_j^{(i,i')}$ such that $\bar{M}_j^{(i,i')} \odot_{\mathcal{I}_j^{(i,i')}} Y_j^{(i,i')}$ for every $(i,i') \in C$, and sends $\{\delta_j^{(i,i')}\}_{(i,i')\in C_j}$ to $\mathcal{A}$.

(c) Then, it provides $\{\mathbf{c}_j^{(i)}, \mathbf{m}_{\mathbf{a}_j}^{(i)}, \mathbf{m}_{\mathbf{b}_j}^{(i)}, \mathbf{m}_{\mathbf{c}_j}^{(i)}\}_{i\in A, j\in[2N]} \xleftarrow{\$} \mathbb{F}_{2^k}$ to $\mathcal{A}$, and it inputs $\{(\mathbf{a}_j^{(i)}, \mathbf{b}_j^{(i)}, \mathbf{c}_j^{(i)}, \mathbf{m}_{\mathbf{a}_j}^{(i)}, \mathbf{m}_{\mathbf{b}_j}^{(i)}, \mathbf{m}_{\mathbf{c}_j}^{(i)})\}_{i\in A, j\in[N]}$ to $\mathcal{F}_{\mathsf{Triples}}$.

(d) Emulating $\mathcal{F}_{\mathsf{Rand}}$, it samples $\mathbf{t}, \mathbf{t}', \mathbf{t}'' \xleftarrow{\$} \mathbb{F}_{2^k}$, and forwards these values to all corrupted parties.

(e) For every $i \in B$ and $j \in [N]$, $\mathcal{S}$ samples $\mathbf{r}_j^{(i)}, \mathbf{s}_j^{(i)} \xleftarrow{\$} \mathbb{F}_{2^k}$, and computes

$$\mathbf{r}_j = \sum_{i\in B} \mathbf{r}_j^{(i)} + \sum_{i'\in A} (\mathbf{t}\cdot\mathbf{b}_j^{(i')} + \mathbf{t}'\cdot\mathbf{b}_{j+N}^{(i')}), \quad \mathbf{s}_j = \sum_{i\in B} \mathbf{s}_j^{(i)} + \sum_{i'\in A} (\mathbf{t}'\cdot\mathbf{a}_j^{(i')} + \mathbf{t}''\cdot\mathbf{a}_{j+N}^{(i')}).$$

(f) $\mathcal{S}$ emulates the communication channels: it sends $\{\mathbf{r}_j^{(i,i')}, \mathbf{s}_j^{(i,i')}\}_{j\in[N]}$ from every party $i \in B$ to every party $i' \in A$; and, in the same way, for every $j \in [N]$, it receives $\mathbf{r}_j^{(i',i)}$ and $\mathbf{s}_j^{(i',i)}$ sent from every party $i' \in A$ to every party $i \in B$.

(g) For every $j \in [N]$ and $i \in B$, compute $\mathbf{f}_{\mathbf{r}_j}^{(i)} = \sum_{i'\in A}(\mathbf{r}_j^{(i',i)} + \mathbf{r}_j^{(i)})$ and $\mathbf{f}_{\mathbf{s}_j}^{(i)} = \sum_{i'\in A}(\mathbf{s}_j^{(i',i)} + \mathbf{s}_j^{(i)})$.

(h) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{BatchCheck}}$: it computes $\{\chi_{\mathbf{r}_j}\}_{j\in[N]}$ and $\{\chi_{\mathbf{s}_j}\}_{j\in[N]}$, and it sends them to $\mathcal{A}$. Then it receives $\zeta_{\mathbf{r}}$ and $\zeta_{\mathbf{s}}$. Hence, for every $j \in [N]$, it computes

$$\varphi_{\mathbf{r}_j,h}^{(i)} = \mathbf{t}\cdot\varphi_{\mathbf{b}_j,h}^{(i)} + \mathbf{t}'\cdot\varphi_{\mathbf{b}_{j+N},h}^{(i)}, \quad \varphi_{\mathbf{s}_j,h}^{(i)} = \mathbf{t}'\cdot\varphi_{\mathbf{a}_j,h}^{(i)} + \mathbf{t}''\cdot\varphi_{\mathbf{a}_{j+N},h}^{(i)},$$

for every $i \in B, h \in [k]$, as well as

$$\psi_{\mathbf{r}_j} = \mathbf{t}\cdot\psi_{\mathbf{b}_j} + \mathbf{t}'\cdot\psi_{\mathbf{b}_{j+N}}, \quad \psi_{\mathbf{s}_j} = \mathbf{t}'\cdot\psi_{\mathbf{a}_j} + \mathbf{t}''\cdot\psi_{\mathbf{a}_{j+N}}$$

$$\xi_{\mathbf{r}_j} = \sum_{i\in A}((\mathbf{t}\cdot\mathbf{b}_j + \mathbf{t}'\cdot\mathbf{b}_{j+N})\cdot\Delta^{(i)} + \mathbf{t}\cdot\mathbf{m}_{\mathbf{b}_j}^{(i)} + \mathbf{t}'\cdot\mathbf{m}_{\mathbf{b}_{j+N}}^{(i)})$$

and

$$\xi_{\mathbf{s}_j} = \sum_{i\in A}((\mathbf{t}'\cdot\mathbf{a}_j + \mathbf{t}''\cdot\mathbf{a}_{j+N})\cdot\Delta^{(i)} + \mathbf{t}'\cdot\mathbf{m}_{\mathbf{a}_j}^{(i)} + \mathbf{t}''\cdot\mathbf{m}_{\mathbf{a}_{j+N}}^{(i)}).$$

The simulator solves

$$\sum_{j\in[N]} \chi_{\mathbf{r}_j}\cdot\Big(\sum_{h\in[k],i\in B} \Delta_h^{(i)}\cdot X^{h-1}\cdot(\mathbf{f}_{\mathbf{r}_j}^{(i)} + \varphi_{\mathbf{r}_j,h}^{(i)}) + \psi_{\mathbf{r}_j} + \xi_{\mathbf{r}_j}\Big) = \zeta_{\mathbf{r}} \tag{3}$$

$$\sum_{j\in[N]} \chi_{\mathbf{s}_j}\cdot\Big(\sum_{h\in[k],i\in B} \Delta_h^{(i)}\cdot X^{h-1}\cdot(\mathbf{f}_{\mathbf{r}_j}^{(i)} + \varphi_{\mathbf{s}_j,h}^{(i)}) + \psi_{\mathbf{s}_j} + \xi_{\mathbf{s}_j}\Big) = \zeta_{\mathbf{s}} \tag{4}$$

for bits $\{\Delta_h^{(i)}\}_{i\in B,h\in[k]}$. If such a solution exists, it queries $\mathcal{F}_{\mathsf{Triples}}$ with the solution space, otherwise it sends Abort to $\mathcal{F}_{\mathsf{Triples}}$. For every $j \in [N]$, $\mathcal{S}$ computes

$$\sum_{i\in B,h\in[k]} \Delta_h^{(i)}\cdot X^{h-1}\cdot\varphi_{\mathbf{a}_j,h}^{(i)} + \varphi_{\mathbf{a}_j}, \quad \sum_{i\in B,h\in[k]} \Delta_h^{(i)}\cdot X^{h-1}\cdot\varphi_{\mathbf{b}_j,h}^{(i)} + \varphi_{\mathbf{b}_j}$$

(see below for details), and inputs them to $\mathcal{F}_{\mathsf{Triples}}$ with $\mathsf{MacError}$.

(i) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{BatchCheck}}$: it computes $\{\chi_{\mathbf{d}_j}\}_{j\in[N]}$ and it outputs it to $\mathcal{A}$. Then it receives $\zeta_{\mathbf{d}}$. For every $j \in [N]$, it computes

$$\varphi^{(i)}_{\mathbf{d}_j,h} = \mathbf{t}\cdot\varphi^{(i)}_{\mathbf{c}_j} + \mathbf{t}''\cdot\varphi^{(i)}_{\mathbf{c}_{j+N}} + (\mathbf{r}_j + \mathbf{f}^{(i)}_{\mathbf{r}_j})\cdot(\varphi^{(i)}_{\mathbf{a}_j,h} + \psi_{\mathbf{a}_j}) + (\mathbf{s}_j + \mathbf{f}^{(i)}_{\mathbf{s}_j})\cdot(\varphi^{(i)}_{\mathbf{b}_{j+N}} + \psi_{\mathbf{b}_{j+N}}),$$

for every $i \in B, h \in [k]$, and

$$\psi_{\mathbf{d}_j} = \mathbf{t}\cdot\psi_{\mathbf{c}_j} + \mathbf{t}''\cdot\psi_{\mathbf{c}_{j+N}} + \mathbf{r}_j\cdot\psi_{\mathbf{a}_j} + \mathbf{s}_j\cdot\psi_{\mathbf{b}_{j+N}}$$

$$\xi_{\mathbf{d}_j} = \sum_{i\in A} \mathbf{t}\cdot\mathbf{m}^{(i)}_{\mathbf{c}_j} + \mathbf{t}''\cdot\mathbf{m}^{(i)}_{\mathbf{c}_{j+N}} + \mathbf{r}_j\cdot\mathbf{m}^{(i)}_{\mathbf{a}_j} + \mathbf{s}_j\cdot\mathbf{m}^{(i)}_{\mathbf{b}_{j+N}}.$$

For every $j \in [N]$, $\mathcal{S}$ solves

$$\sum_{j\in[N]} \chi_{\mathbf{d}_j}\cdot\Big(\sum_{h\in[k],i\in B} \Delta^{(i)}_h\cdot X^{h-1}\cdot\varphi^{(i)}_{\mathbf{d}_j,h} + \psi_{\mathbf{d}_j} + \xi_{\mathbf{d}_j}\Big) = \zeta_{\mathbf{d}}$$

for bits $\{\Delta^{(i)}_h\}_{i\in B,h\in[k]}$. If such a solution exists, $\mathcal{S}$ queries $\mathcal{F}_{\mathsf{Triples}}$ with the solution space, otherwise it sends Abort to $\mathcal{F}_{\mathsf{Triples}}$. For every $j \in [N]$, compute $\sum_{i\in B,h\in[k]} \Delta^{(i)}_h\cdot X^{h-1}\cdot\varphi^{(i)}_{\mathbf{c}_j,h} + \varphi_{\mathbf{c}_j}$ (see below for details), and it inputs them to $\mathcal{F}_{\mathsf{Triples}}$ with $\mathsf{MacError}$. Furthermore, it inputs $\sum_{(i,i')\in C}\delta^{(i,i')}_j + \bar{\mathbf{f}}_j$ to $\mathcal{F}_{\mathsf{Triples}}$ with $\mathsf{ValueError}$ for $\mathbf{c}_j$.

At a high level, we can see the simulation consisting of two main parts, simulating the opening of $(\mathbf{r}_j, \mathbf{s}_j)$ and simulating the MAC checks. The first is straightforward: We receive the corrupted parties' shares of $\{\mathbf{a}_j, \mathbf{b}_j, \mathbf{c}_j\}_{j\in[N]}$ from $\mathcal{A}$ and randomly chose their shares of $\{\mathbf{a}_j, \mathbf{b}_j, \mathbf{c}_j\}_{j\in[N+1,2N]}$ as well all shares of $\{\mathbf{r}_j, \mathbf{s}_j\}_{j\in[N]}$. This is indistinguishable to the real protocol because the environment never learns $\{\mathbf{a}_j, \mathbf{b}_j, \mathbf{c}_j\}_{j\in[N+1,2N]}$ and thus $\{\mathbf{r}_j, \mathbf{s}_j\}_{j\in[N]}$ are uniformly random from the environment's view.

In the $\mathcal{F}_{\mathsf{BatchCheck}}$ we simulate the fact the adversary can test the single bits of honest parties' shares of Δ using the $\varphi^{(i)}_h$ variables. The simulator computes the successful space of those variables and tests it with $\mathcal{F}_{\mathsf{Triples}}$, which will abort if unsuccessful.

The first invocation of $\mathcal{F}_{\mathsf{BatchCheck}}$ in the real world succeeds if the first equality of the following holds:

$$\begin{aligned}
\zeta_{\mathbf{r}} &= \sum_{\substack{i\in B\\ j\in[N]}} \chi_{\mathbf{r}_j}\cdot\big((\mathbf{r}_j + \mathbf{f}^{(i)}_{\mathbf{r}_j})\cdot\Delta^{(i)} + \mathbf{t}\cdot\mathbf{m}^{(i)}_{\mathbf{b}_j} + \mathbf{t}'\cdot\mathbf{m}^{(i)}_{\mathbf{b}_{j+N}}\big)\\
&= \sum_{\substack{i\in B\\ j\in[N]}} \chi_{\mathbf{r}_j}\cdot\Big(\Big(\sum_{i'\in\mathcal{P}}(\mathbf{t}\cdot\mathbf{b}^{(i')}_j + \mathbf{t}'\cdot\mathbf{b}^{(i')}_{j+N}) + \mathbf{f}^{(i)}_{\mathbf{r}_j}\Big)\cdot\Delta^{(i)} + \mathbf{t}\cdot\mathbf{m}^{(i)}_{\mathbf{b}_j} + \mathbf{t}'\cdot\mathbf{m}^{(i)}_{\mathbf{b}_{j+N}}\Big)\\
&= \sum_{\substack{i\in B\\ j\in[N]}} \chi_{\mathbf{r}_j}\cdot\big((\mathbf{t}\cdot\mathbf{b}_j + \mathbf{t}'\cdot\mathbf{b}_{j+N} + \mathbf{f}^{(i)}_{\mathbf{r}_j})\cdot\Delta^{(i)} + \mathbf{t}\cdot\mathbf{m}^{(i)}_{\mathbf{b}_j} + \mathbf{t}'\cdot\mathbf{m}^{(i)}_{\mathbf{b}_{j+N}}\big)\\
&= \sum_{j\in[N]} \chi_{\mathbf{r}_j}\cdot\Big((\mathbf{t}\cdot\mathbf{b}_j + \mathbf{t}'\cdot\mathbf{b}_{j+N})\cdot\Big(\Delta + \sum_{i\in A}\Delta^{(i)}\Big) + \sum_{i\in B}\mathbf{f}^{(i)}_{\mathbf{r}_j}\cdot\Delta^{(i)}\\
&\qquad + \mathbf{t}\cdot\Big(\mathbf{b}_j\cdot\Delta + \sum_{i\in B,h\in[k]}\Delta^{(i)}_h\cdot X^{h-1}\cdot\varphi^{(i)}_{\mathbf{b}_j} + \psi_{\mathbf{b}_j} + \sum_{i\in A}\mathbf{m}^{(i)}_{\mathbf{b}_j}\Big)\\
&\qquad + \mathbf{t}'\cdot\Big(\mathbf{b}_{j+N}\cdot\Delta + \sum_{i\in B,h\in[k]}\Delta^{(i)}_h\cdot X^{h-1}\cdot\varphi^{(i)}_{\mathbf{b}_{j+N}} + \psi_{\mathbf{b}_{j+N}} + \sum_{i\in A}\mathbf{m}^{(i)}_{\mathbf{b}_{j+N}}\Big)\Big)\\
&= \sum_{j\in[N]} \chi_{\mathbf{r}_j}\cdot\Big((\mathbf{t}\cdot\mathbf{b}_j + \mathbf{t}'\cdot\mathbf{b}_{j+N})\cdot\sum_{i\in A}\Delta^{(i)} + \sum_{i\in B}\mathbf{f}^{(i)}_{\mathbf{r}_j}\cdot\Delta^{(i)}\\
&\qquad + \mathbf{t}\cdot\Big(\sum_{i\in B,h\in[k]}\Delta^{(i)}_h\cdot X^{h-1}\cdot\varphi^{(i)}_{\mathbf{b}_j} + \psi_{\mathbf{b}_j} + \sum_{i\in A}\mathbf{m}^{(i)}_{\mathbf{b}_j}\Big)
\end{aligned}$$

$$+\mathbf{t}'\cdot\Big(\sum_{i\in B,h\in[k]}\Delta_h^{(i)}\cdot X^{h-1}\cdot\varphi^{(i)}_{\mathbf{b}_{j+N}}+\psi_{\mathbf{b}_{j+N}}+\sum_{i\in A}\mathbf{m}^{(i)}_{\mathbf{b}_{j+N}}\Big)\Big)$$
$$=\sum_{j\in[N]}\chi_{\mathbf{r}_j}\cdot\Big(\sum_{i\in B,h\in[k]}\Delta_h^{(i)}\cdot X^{h-1}\cdot(\mathbf{f}^{(i)}_{\mathbf{r}_j}+\mathbf{t}\cdot\varphi^{(i)}_{\mathbf{b}_j,h}+\mathbf{t}''\cdot\varphi^{(i)}_{\mathbf{a}_{j+N},h})+\psi_{\mathbf{r}_j}+\xi_{\mathbf{r}_j}\Big).$$

This equals the equality 3 in the simulation. Applying Corollary 1 with

$$f_{j,1}:\{\Delta_h^{(i)}\}_{i\in B,h\in[k]}\mapsto\sum_{i\in B,h\in[k]}\Delta_h^{(i)}\cdot X^{h-1}\cdot\varphi^{(i)}_{\mathbf{b}_j,h}$$
$$f_{j,2}:\{\Delta_h^{(i)}\}_{i\in B,h\in[k]}\mapsto\sum_{i\in B,h\in[k]}\Delta_h^{(i)}\cdot X^{h-1}\cdot\varphi^{(i)}_{\mathbf{a}_{j+N},h}$$

as the linear maps for $j\in[N]$, we get that the probability of the check passing and

$$\sum_{i\in B,h\in[k]}\Delta_h^{(i)}\cdot X^{h-1}\cdot\varphi^{(i)}_{\mathbf{b}_j,h}$$

not being uniquely defined for any $j\in[N]$ is 2^{-k+1}. Similarly, one can prove the same for

$$\sum_{i\in B,h\in[k]}\Delta_h^{(i)}\cdot X^{h-1}\cdot\varphi^{(i)}_{\mathbf{a}_j,h}.$$

By solving the linear equation, this allows to compute $\sum_{i\in B,h\in[k]}\Delta_h^{(i)}\cdot X^{h-1}\cdot\varphi^{(i)}_{\mathbf{a}_j,h}+\varphi_{\mathbf{a}_j}$ and $\sum_{i\in B,h\in[k]}\Delta_h^{(i)}\cdot X^{h-1}\cdot\varphi^{(i)}_{\mathbf{b}_j,h}+\varphi_{\mathbf{b}_j}$ as required by the simulator.

The last invocation of $\mathcal{F}_{\mathsf{BatchCheck}}$ in the real world succeeds if the first equality of the following holds:

$$\begin{aligned}
\zeta_{\mathbf{d}}&=\sum_{i\in B,j\in[N]}\chi_{\mathbf{d}_j}\cdot\big(\mathbf{t}\cdot\mathbf{m}^{(i)}_{\mathbf{c}_j}+\mathbf{t}''\cdot\mathbf{m}_{\mathbf{c}_{j+N}}+\mathbf{r}_j\cdot\mathbf{m}^{(i)}_{\mathbf{a}_j}+\mathbf{s}_j\cdot\mathbf{m}^{(i)}_{\mathbf{b}_{j+N}}\big)\\
&=\sum_{j\in[N]}\chi_{\mathbf{d}_j}\cdot\Big(\mathbf{t}\cdot\Big(\mathbf{c}_j\cdot\Delta+\sum_{i\in B,h\in[k]}\Delta_h^{(i)}\cdot X^{h-1}\cdot\varphi^{(i)}_{\mathbf{c}_j}+\psi_{\mathbf{c}_j}+\sum_{i\in A}\mathbf{m}^{(i)}_{\mathbf{c}_j}\Big)\\
&\quad+\mathbf{t}''\cdot\Big(\mathbf{c}_{j+N}\cdot\Delta+\sum_{i\in B,h\in[k]}\Delta_h^{(i)}\cdot X^{h-1}\cdot\varphi^{(i)}_{\mathbf{c}_{j+N}}+\psi_{\mathbf{c}_{j+N}}+\sum_{i\in A}\mathbf{m}^{(i)}_{\mathbf{c}_{j+N}}\Big)\\
&\quad+\sum_{i\in B}\Big(\mathbf{r}_j\cdot\mathbf{m}^{(i)}_{\mathbf{a}_j}+\mathbf{s}_j\cdot\mathbf{m}^{(i)}_{\mathbf{b}_{j+N}}\Big)\Big)\\
&=\sum_{j\in[N]}\chi_{\mathbf{d}_j}\cdot\Big(\mathbf{t}\cdot\Big(\Big(\mathbf{a}_j\cdot\mathbf{b}_j+\sum_{i\in B'}\mathbf{a}^{(i)}_j\cdot\mathbf{f}^{(i)}_{\mathbf{a}_j}+\sum_{i\in B'}\mathbf{b}^{(i)}_j\cdot\mathbf{f}^{(i)}_{\mathbf{b}_j}+\mathbf{f}_j\Big)\cdot\Delta+\sum_{i\in A}\mathbf{m}^{(i)}_{\mathbf{c}_j}\Big)\\
&\quad+\mathbf{t}''\cdot\Big(\Big(\mathbf{a}_{j+N}\cdot\mathbf{b}_{j+N}+\sum_{i\in B'}\mathbf{a}^{(i)}_{j+N}\cdot\mathbf{f}^{(i)}_{\mathbf{a}^{j+N}}+\sum_{i\in B'}\mathbf{b}^{(i)}_{j+N}\cdot\mathbf{f}^{(i)}_{\mathbf{b}_{j+N}}+\mathbf{f}_{j+N}\Big)\cdot\Delta+\sum_{i\in A}\mathbf{m}^{(i)}_{\mathbf{c}_{j+N}}\Big)\\
&\quad+\mathbf{t}'\cdot\Big(\sum_{i\in B,h\in[k]}\Delta_h^{(i)}\cdot X^{h-1}\cdot\varphi^{(i)}_{\mathbf{c}_j,h}+\psi_{\mathbf{c}_j}\Big)\\
&\quad+\mathbf{t}''\cdot\Big(\sum_{i\in B,h\in[k]}\Delta_h^{(i)}\cdot X^{h-1}\cdot\varphi^{(i)}_{\mathbf{c}_{j+N},h}+\psi_{\mathbf{c}_{j+N}}\Big)\\
&\quad+(\mathbf{t}\cdot\mathbf{b}_j+\mathbf{t}'\cdot\mathbf{b}_{j+N})\cdot\mathbf{a}_j\cdot\Delta+\mathbf{r}_j\cdot\Big(\sum_{i\in B,h\in[k]}\Delta_h^{(i)}\cdot X^{h-1}\cdot\varphi^{(i)}_{\mathbf{a}_j,h}+\psi_{\mathbf{a}_j}+\sum_{i\in A}\mathbf{m}^{(i)}_{\mathbf{a}_j}\Big)\\
&\quad+(\mathbf{t}'\cdot\mathbf{a}_j+\mathbf{t}''\cdot\mathbf{a}_{j+N})\cdot\mathbf{b}_{j+N}\cdot\Delta+\mathbf{s}_j\cdot\Big(\sum_{i\in B,h\in[k]}\Delta_h^{(i)}\cdot X^{h-1}\cdot\varphi^{(i)}_{\mathbf{b}_{j+N},h}+\psi_{\mathbf{b}_{j+N}}+\sum_{i\in A}\mathbf{m}^{(i)}_{\mathbf{b}_{j+N}}\Big)
\end{aligned}$$

$$= \sum_{j\in[N]} \chi_{\mathbf{d}_j} \cdot \Big(\Big(\mathbf{t} \cdot \Big(\sum_{i\in B'} \mathbf{a}_j^{(i)} \cdot \mathbf{f}_{\mathbf{a}_j}^{(i)} + \sum_{i\in B'} \mathbf{b}_j^{(i)} \cdot \mathbf{f}_{\mathbf{b}_j}^{(i)} + \mathbf{f}_j \Big) + \mathbf{t}'' \cdot \Big(\sum_{i\in B'} \mathbf{a}_{j+N}^{(i)} \cdot \mathbf{f}_{\mathbf{a}_{j+N}}^{(i)} + \sum_{i\in B'} \mathbf{b}_{j+N}^{(i)} \cdot \mathbf{f}_{\mathbf{b}_{j+N}}^{(i)} + \mathbf{f}_{j+N} \Big) \Big) \cdot \Delta + \sum_{i\in B, h\in[k]} \Delta_h^{(i)} \cdot X^{h-1} \cdot \varphi_{\mathbf{d}_j,h}^{(i)} + \psi_{\mathbf{d}_j} + \xi_{\mathbf{d}_j} \Big).$$

In the real world, if the adversary inputs $\bar{\mathbf{f}}$ C, and $\{Y^{(i,i')}\}_{(i,i')\in C}$ to $\mathcal{F}_{\mathsf{UncheckedTriples}}$ in the j-th triple generation, $\mathbf{f}_j = \sum_{(i,i')\in C} \mathbf{f}_j^{(i,i')} + \bar{\mathbf{f}}$ for $\mathbf{f}_j^{(i,i')}$ uniformly distributed in $\hat{M}^{(i,i')}(\bar{M}^{(i,i')})^{-1}(\delta_j^{(i,i')} + \mathbf{a}_j^{(i)})$. Clearly, the larger the set is the lower the probability that the above equality holds and the MAC check succeeds. Therefore, we assume that the set collapses to a set of size one. This holds if $\ker \bar{M}^{(i,i')} \subset \ker \hat{M}^{(i,i')}$. By definition, $\hat{M}^{(i,i')} = (\bar{M}^{(i,i')} \odot Y')$. Lemma 12 implies that the condition only holds with probability at most 2^{-k} if not all rows of Y' are the same, say $\mathbf{y}^{(i,i')}$. Hence, $\mathbf{f}_j^{(i,i')} = \delta_j^{(i,i')} + \mathbf{a}_j^{(i)} \cdot \mathbf{y}^{(i,i')}$, and

$$\mathbf{f}_j = \sum_{(i,i')\in C} \mathbf{f}_j^{(i,i')} + \bar{\mathbf{f}}_j = \sum_{(i,i')\in C} \delta_j^{(i,i')} + \mathbf{a}_j^{(i)} \cdot \mathbf{y}^{(i,i')} + \bar{\mathbf{f}}_j.$$

It follows that the equation above can be rewritten as

$$\zeta_{\mathbf{d}} = \sum_{j\in[N]} \chi_{\mathbf{d}_j} \cdot \Big(\Big(\mathbf{t} \cdot \Big(\sum_{i\in B'} \mathbf{a}_j^{(i)} \cdot \Big(\mathbf{f}_{\mathbf{a}_j}^{(i)} + \sum_{(i,i')\in C} \mathbf{y}^{(i,i')} \Big) + \sum_{i\in B'} \mathbf{b}_j^{(i)} \cdot \mathbf{f}_{\mathbf{b}_j}^{(i)} + \sum_{(i,i')\in C} \delta_j^{(i,i')} + \bar{\mathbf{f}}_j \Big) + \mathbf{t}'' \cdot \Big(\sum_{i\in B'} \mathbf{a}_{j+N}^{(i)} \cdot \Big(\mathbf{f}_{\mathbf{a}_{j+N}}^{(i)} + \sum_{(i,i')\in C} \mathbf{y}^{(i,i')} \Big) + \sum_{i\in B'} \mathbf{b}_{j+N}^{(i)} \cdot \mathbf{f}_{\mathbf{b}_{j+N}}^{(i)} + \sum_{(i,i')\in C} \delta_j^{(i,i')} + \bar{\mathbf{f}}_{j+N} \Big) \Big) \cdot \Delta + \sum_{i\in B, h\in[k]} \Delta_h^{(i)} \cdot X^{h-1} \cdot \varphi_{\mathbf{d}_j,h}^{(i)} + \psi_{\mathbf{d}_j} + \xi_{\mathbf{d}_j} \Big).$$

Considering that $\{\mathbf{a}_j^{(i)}, \mathbf{b}_j^{(i)}\}_{i\in B', j\in[2N]}$ are uniformly random independent of all other variables and $\{\chi_{\mathbf{d}_j}\}_{j\in[N]}, \mathbf{t}, \mathbf{t}''$ are uniformly random, Lemma 7 implies that the check only passes with probability $3 \cdot 2^{-k}$ if any of $\{(\mathbf{f}_{\mathbf{a}_j}^{(i)} + \sum_{(i,i')\in C_j} \mathbf{y}_j^{(i,i')}) \cdot \Delta, \mathbf{f}_{\mathbf{b}_j}^{(i)} \cdot \Delta\}_{i\in B', j\in[2N]}$ is not zero. Since $\Delta \neq 0$ with probability 2^{-k}, $\{((\mathbf{f}_{\mathbf{a}_j}^{(i)} + \sum_{(i,i')\in C_j} \mathbf{y}_j^{(i,i')}), \mathbf{f}_{\mathbf{a}_j}^{(i)})\}_{i\in B', j\in[2N]}$ must be zero for the check to pass with non-negligible probability. This establishes that

$$\begin{aligned}
\mathbf{c}_j &= \mathbf{a}_j \cdot \mathbf{b}_j + \sum_{i\in B'} \mathbf{a}^{(i)} \cdot \mathbf{f}_{\mathbf{a}}^{(i)} + \sum_{i\in B'} \mathbf{b}^{(i)} \cdot \mathbf{f}_{\mathbf{b}}^{(i)} + \sum_{(i,i')\in C_j} \mathbf{f}^{(i,i')} + \bar{\mathbf{f}}_j \\
&= \mathbf{a}_j \cdot \mathbf{b}_j + \sum_{i\in B'} \mathbf{a}_j^{(i)} \cdot \Big(\mathbf{f}_{\mathbf{a}_j}^{(i)} + \sum_{(i,i')\in C} \mathbf{y}^{(i,i')} \Big) + \sum_{i\in B'} \mathbf{b}_j^{(i)} \cdot \mathbf{f}_{\mathbf{b}_j}^{(i)} + \sum_{(i,i')\in C} \delta_j^{(i,i')} + \bar{\mathbf{f}}_j \\
&= \mathbf{a}_j \cdot \mathbf{b}_j + \sum_{(i,i')\in C} \delta_j^{(i,i')} + \bar{\mathbf{f}}_j,
\end{aligned}$$

which is used for the $\mathsf{ValueError}$ input to $\mathcal{F}_{\mathsf{Triples}}$. Similarly, for $\hat{\mathbf{f}}_j = \bar{\mathbf{f}}_j + \sum_{(i,i')\in C} \delta_j^{(i,i')}$, we assume that

$$\begin{aligned}
\zeta_{\mathbf{d}} &= \sum_{j\in[N]} \chi_{\mathbf{d}_j} \cdot \Big((\mathbf{t} \cdot \hat{\mathbf{f}}_j + \mathbf{t}'' \cdot \hat{\mathbf{f}}_{j+N}) \cdot \Delta + \sum_{i\in B, h\in[k]} \Delta_h^{(i)} \cdot X^{h-1} \cdot \varphi_{\mathbf{d}_j,h}^{(i)} + \psi_{\mathbf{d}_j} + \xi_{\mathbf{d}_j} \Big) \\
&= \sum_{j\in[N]} \chi_{\mathbf{d}_j} \cdot \Big((\mathbf{t} \cdot \hat{\mathbf{f}}_j + \mathbf{t}'' \cdot \hat{\mathbf{f}}_{j+N}) \cdot \sum_{i\in A} \Delta^{(i)} \\
&\quad + \sum_{i\in B, h\in[k]} \Delta_h^{(i)} \cdot X^{h-1} \cdot (\varphi_{\mathbf{d}_j,h}^{(i)} + \mathbf{t} \cdot \hat{\mathbf{f}}_j + \mathbf{t}'' \cdot \hat{\mathbf{f}}_{j+N}) + \psi_{\mathbf{d}_j} + \xi_{\mathbf{d}_j} \Big)
\end{aligned}$$

$$= \sum_{j\in[N]} \chi_{\mathbf{d}_j} \cdot \Big((\mathbf{t}\cdot\hat{\mathbf{f}}_j + \mathbf{t}''\cdot\hat{\mathbf{f}}_{j+N}) \cdot \sum_{i\in A} \Delta^{(i)} + \sum_{i\in B, h\in[k]} \Delta_h^{(i)} \cdot X^{h-1} \cdot (\mathbf{t}\cdot(\varphi^{(i)}_{\mathbf{c}_j,h} + \hat{\mathbf{f}}_j) + \mathbf{t}''\cdot(\varphi^{(i)}_{\mathbf{c}_{j+N},h} + \hat{\mathbf{f}}_{j+N}) + (\mathbf{r}_j + \hat{\mathbf{f}}^{(i)}_{\mathbf{r}_j})\cdot(\varphi^{(i)}_{\mathbf{a}_j,h} + \psi_{\mathbf{a}_j}) + (\mathbf{s}_j + \hat{\mathbf{f}}^{(i)}_{\mathbf{s}_j})\cdot(\varphi^{(i)}_{\mathbf{b}_{j+N},h} + \psi_{\mathbf{b}_{j+N}})) + \psi_{\mathbf{d}_j} + \xi_{\mathbf{d}_j} \Big).$$

With

$$f_{j,1} : \{\Delta_h^{(i)}\}_{i\in B,h\in[k]} \mapsto \sum_{i\in B,h\in[k]} \Delta_h^{(i)} \cdot X^{h-1} \cdot (\varphi^{(i)}_{\mathbf{c}_j,h} + \hat{\mathbf{f}}_j)$$
$$f_{j,2} : \{\Delta_h^{(i)}\}_{i\in B,h\in[k]} \mapsto \sum_{i\in B,h\in[k]} \Delta_h^{(i)} \cdot X^{h-1} \cdot (\varphi^{(i)}_{\mathbf{c}_{j+N},h} + \hat{\mathbf{f}}_{j+N}),$$

for all $j \in [N]$, Corollary 1 implies that the probability of the check passing and

$$\sum_{i\in B,h\in[k]} \Delta_h^{(i)} \cdot X^{h-1} \cdot (\varphi^{(i)}_{\mathbf{c}_j,h} + \hat{\mathbf{f}}_j)$$

not being clearly defined for all $j \in [N]$ is at most 2^{-k+1}. Again, this allows to compute $\sum_{i\in B,h\in[k]} \Delta_h^{(i)} \cdot X^{h-1} \cdot (\varphi^{(i)}_{\mathbf{c}_j,h} + \hat{\mathbf{f}}_j) + \psi_{\mathbf{c}_j}$ as required by the simulator.

Summing up using the union bound, we find that the statistical distance between the real world and simulation is at most $2\cdot 2^{-k+1} + 2^{-k}3\cdot 2^{-k} + 2^{-k} + 2^{-k+1} = 11\cdot 2^{-k}$. We conclude that $\Pi_{\mathsf{TripleCheck}}$ $(k-4)$-securely implements $\mathcal{F}_{\mathsf{Triples}}$ in the $(\mathcal{F}_{\mathsf{UncheckedTriples}}, \mathcal{F}_{\mathsf{Rand}})$-model.

G MiniMAC Protocols and Security Proofs

In this section we describe the ideal functionalities, protocols and proofs of our MiniMAC preprocessing. This section is organized as follows: In subsection G.1 we present the ideal functionalities for the preprocessing aspects of MiniMAC. Afterwards, in subsection G.2 we present the ideal functionality, protocol and proof of the codeword authentication protocol. Following this in subsection G.3 we describe the ideal functionalities, protocols, simulators and proofs needed in the construction of the multiplication triples. Finally in subsection G.4 and G.5 we present the protocol and simulator for construction of the Schur triples, respectively reorganization triples.

Throughout this section we will interchangeably be viewing elements expressible with $u\cdot m$ (or $u\cdot k$) bits as either bit vectors ($\mathbb{F}_2^{u\cdot m}$), elements of the characteristic 2 Galois extension field of order $2^{u\cdot m}$ ($\mathbb{F}_{2^{u\cdot m}}$) or a vector of m elements in the characteristic 2 Galois extension field of order 2^u ($\mathbb{F}_{2^u}^m$).

G.1 Preprocessing Functionality

In this subsection we define the ideal functionalities needed to realize the MiniMAC preprocessing. We do this in Fig. 27 and 28, using the macros defined in Fig. 29.

G.2 Codeword Authentication

We now describe the codeword authentication protocol. This protocol allows parties to authenticate shares of components of a codeword such that the output is guaranteed to be authenticated shares of a valid codeword. There are two main stages: firstly a BigMAC is constructed which consists of a (big) MAC on each *component* of a codeword. Then these are combined and compressed into a single MiniMAC for the entire codeword.

This functionality generates offline material used in the MiniMAC online protocol. We denote by A the set of parties controlled by the adversary.

Initialize: On input $(\mathsf{Init}, m, k, d, u, G)$ from all parties, store integers m, k, d, u and generator matrix G for a linear $[m, k, d]$-code C over the field $\mathbb{F}_{2^u}$.

1. For each corrupt party P_i with $i \in A$, get element $\Delta^{(i)} \in \mathbb{F}_{2^u}^m$ from the adversary.
2. Pick each share $\Delta^{(i)}$ for $i \notin A$ uniformly at random from $\mathbb{F}_{2^u}^m$ and define $\Delta = \sum_{i=1}^n \Delta^{(i)}$.
3. If the functionality receives the signal Abort from the adversary then halt and output Abort.
4. Output $\Delta^{(i)}$ to party P_i.

Computation: On input $\mathsf{DataGen}$ from all honest parties and the adversary, and only if the functionality received $\mathsf{Proceed}$ (or $\mathsf{BreakDown}$ is set to true) it executes the data generation procedures specified in Figure 28 composed with the macro $\mathsf{Bracket}$ in Figure 29.

Fig. 27. Ideal functionality for MiniMAC offline generation

This functionality generates offline material used in the MiniMAC online protocol. We denote by A the set of parties controlled by the adversary.

Schur Pair $(\llbracket C(\mathbf{r}) \rrbracket^*, \llbracket C^*(\mathbf{s}) \rrbracket^*)$**:**

1. Receive the shares $\left\{C(\mathbf{r}^{(i)}), C^*(\mathbf{s}^{(i)})\right\}_{i \in A}$ from the adversary, where $C(\mathbf{r}^{(i)})$ and $C^*(\mathbf{s}^{(i)})$ are equal in the first k positions. Similarly pick the shares $\left\{C(\mathbf{r}^{(i)}), C^*(\mathbf{s}^{(i)})\right\}_{i \notin A}$ for each of the honest parties, such that $C(\mathbf{r}^{(i)})$ and $C^*(\mathbf{s}^{(i)})$ are equal in the first k positions and the following $k^* - k$ positions of $C^*(\mathbf{s}^{(i)})$ are chosen uniformly at random.
2. Run the $\mathsf{Bracket}$ macros $\left\{C(\mathbf{r}^{(i)}), \Delta^{(i)}\right\}_{i \in [n]}$ and $\left\{C^*(\mathbf{s}^{(i)}), \Delta^{(i)}\right\}_{i \in [n]}$ and return the output.

Reorganization $(\llbracket C(\mathbf{r}) \rrbracket^*, \llbracket C(f(\mathbf{r})) \rrbracket^*)$**:**

1. Receive the shares $\left\{C(\mathbf{r}^{(i)})\right\}_{i \in A}$ from the adversary and pick the shares $\left\{C(\mathbf{r}^{(i)})\right\}_{i \notin A}$ uniformly at random for each of the honest parties.
2. Run the $\mathsf{BigBracket}$ macro on $\left\{C(\mathbf{r}^{(i)}), \Delta^{(i)}\right\}_{i \in [n]}$ to get $\left\{C(\mathbf{r}^{(i)}), \left\{\mathbf{m}^{(i)}_{(\mathbf{r},h)}\right\}_{h \in [m]}\right\}$.
3. Letting $\llbracket \mathbf{r}_h \rrbracket$ be defined by $\left(\langle \mathbf{r}[h] \rangle, \langle \mathbf{m}^{(i)}_{(\mathbf{r},h)} \rangle, \langle \Delta \rangle\right)$ for $h \in [k]$, apply f and then C to $\llbracket \mathbf{r}_1 \rrbracket, \ldots, \llbracket \mathbf{r}_k \rrbracket$, to obtain $\llbracket C(f(\mathbf{r}))_1 \rrbracket, \ldots, \llbracket C(f(\mathbf{r}))_k \rrbracket$ where $C(f(\llbracket \mathbf{r}_h \rrbracket)) = \llbracket C(f(\mathbf{r}))_h \rrbracket$.
4. Finally return $\left\{C(\mathbf{r}^{(i)}), \mathbf{m}^{(i)*}_{\mathbf{r}}\right\}$ and $\left\{C(f(\mathbf{r}^{(i)})), \mathbf{m}^{(i)*}_{f(\mathbf{r})}\right\}$ where $\mathbf{m}^{(i)*}_{\mathbf{r}}, \mathbf{m}^{(i)*}_{f(\mathbf{r})} \in \mathbb{F}_{2^u}^m$ and $\mathbf{m}^{(i)*}_{\mathbf{r}}[h] = \mathbf{m}^{(i)}_{(\mathbf{r},h)}[h]$, respectively $\mathbf{m}^{(i)*}_{f(\mathbf{r})}[h] = \mathbf{m}^{(i)}_{(f(\mathbf{r}),h)}[h]$ for $h \in [m]$.

Multiplication $(\llbracket C(\mathbf{a}) \rrbracket^*, \llbracket C(\mathbf{b}) \rrbracket^*, \llbracket C^*(\mathbf{c}) \rrbracket^*)$**:**

1. Sample shares $\left\{C(\mathbf{a}^{(i)}), C(\mathbf{b}^{(i)})\right\}_{i \in [n]}$ and compute $C^*(\mathbf{c}^{(i)}) = C(\mathbf{a}^{(i)}) * C(\mathbf{b}^{(i)})$.
2. Run the $\mathsf{Bracket}$ macro on $\left\{C(\mathbf{a}^{(i)}), \Delta^{(i)}\right\}_{i \in [n]}$, $\left\{C(\mathbf{b}^{(i)}), \Delta^{(i)}\right\}_{i \in [n]}$ and $\left\{C^*(\mathbf{c}^{(i)}), \Delta^{(i)}\right\}_{i \in [n]}$.
3. Output $\llbracket C(\mathbf{a}) \rrbracket^*, \llbracket C(\mathbf{b}) \rrbracket^*, \llbracket C^*(\mathbf{c}) \rrbracket^*$.

Key queries: On input of a description of an affine subspace $S \subset (\mathbb{F}_2^{m \cdot u})^n$, return $\mathsf{Success}$ if $(\Delta^{(1)}, \ldots, \Delta^{(n)}) \in S$. Otherwise return Abort.

Fig. 28. Ideal functionality for MiniMAC offline generation (continued)

We present the codeword authentication protocol Π_{CodeAuth} in Fig. 30 and its ideal functionality $\mathcal{F}_{\mathsf{CodeAuth}}$ in Fig. 31. The protocol uses the $\mathcal{F}_{\llbracket \cdot \rrbracket}$ functionality, which is described in Fig. 5.

The BigMAC part of the protocol consists of first having each party i give as input the non-parity components $\mathbf{x}_1^{(i)}, \ldots, \mathbf{x}_k^{(i)} \in \mathbb{F}_{2^u}$ of his shares of a systematic codeword $C(\mathbf{x})$. Each component share then gets authenticated using $\mathcal{F}_{\mathsf{CodeAuth}}$ using an $m \cdot u$ bit global key. Thus each party will thus have a MAC share in $\mathbb{F}_{2^u}^m$ of the authentication of each of the k components. The authentications of the last $m - k$ components of the shares of $C(\mathbf{x})$ are then computed through linear combinations of the BigMACs using the generator

Ideal Authentication Macros

The macros take input $\left(\{\mathbf{x}^{(i)}, \Delta^{(i)}\}, C\right)_{i\in[n]}$, where each $\mathbf{x}^{(i)} \in \mathbb{F}_{2^u}^k$, C is a code of degree k and dimension m, and $\Delta^{(i)} \in \mathbb{F}_{2^u}^m$.

Bracket:

1. Let $\mathbf{x} = \sum_{i\in[n]} \mathbf{x}^{(i)}$. Furthermore let $\mathbf{m} = C(\mathbf{x}) * \Delta$.
2. For every corrupt party P_i for $i \in \mathcal{A}$ the adversary specifies a share $\mathbf{m}^{(i)}$.
3. The functionality sets each share $\mathbf{m}^{(i)}$ for $i \notin \mathcal{A}$ uniformly random under the constraint that $\sum_{i\in[n]} \mathbf{m}^{(i)} = \mathbf{m}$.
4. If the adversary inputs $(\mathsf{Error}, \{e_{h,j}^{(i)}\}_{i\notin A, h\in[k], j\in[m\cdot u]})$. with elements in $\mathbb{F}_{2^{m\cdot u}}$, set $\mathbf{m}^{(i)} := \mathbf{m}^{(i)} + \mathbf{e}^{(i)}$, where $\mathbf{e}^{(i)} \in \mathbb{F}_{2^u}^m$ and
$$\mathbf{e}^{(i)}[h] = \sum_{j=1}^{m\cdot u} e_{h,j}^{(i)} \cdot \Delta_j^{(i)} \cdot X^{j-1}$$
where $\Delta_j^{(i)}$ denotes the j-th bit of $\Delta^{(i)}$ for $h \in [m]$.
5. Output the shares $(\mathbf{x}^{(i)}, \mathbf{m}^{(i)})$ to each party P_i.

BigBracket:

1. Let the h-th component of $\mathbf{x}^{(i)}$ be denoted by $\mathbf{x}^{(i)}[h]$.
2. Next let $\mathbf{x}[h] = \sum_{i\in[n]} \mathbf{x}^{(i)}[h]$. Furthermore let $\mathbf{m}_h = \mathbf{x}[h] \cdot \Delta$.
3. For every corrupt party P_i for $i \in \mathcal{A}$ the adversary specifies shares $\left\{\mathbf{m}_h^{(i)}\right\}_{h\in[k]}$.
4. The functionality sets the shares $\left\{\mathbf{m}_h^{(i)}\right\}_{h\in[k]}$ for $i \notin \mathcal{A}$ uniformly random under the constraint that $\sum_{i\in[n]} \mathbf{m}_h^{(i)} = \mathbf{m}_h$ for $h \in [k]$.
5. If the adversary inputs $(\mathsf{Error}, \{e_{h,j}^{(i)}\}_{i\notin A, h\in[k], j\in[m\cdot u]})$ with elements in $\mathbb{F}_{2^{m\cdot u}}$, set $\mathbf{m}_h^{(i)} = \mathbf{m}_h^{(i)} + \sum_{j=1}^{m\cdot u} e_{h,j}^{(i)} \cdot \Delta_j^{(i)} \cdot X^{j-1}$ where $\Delta_j^{(i)}$ denotes the j-th bit of $\Delta^{(i)}$.
6. For $i \in [n]$ and $h \in [k+1; m]$, compute the values $\mathbf{m}_h^{(i)}$ by applying the code C to $\{\mathbf{m}_h^{(i)}\}_{h\in[k]}$, and similarly for $\{\mathbf{x}^{(i)}[h]\}_{h\in[k+1;m]}$.
7. For each $i \notin A$ and $h \in [m]$, output the shares $\{\mathbf{x}^{(i)}, \mathbf{m}_h^{(i)}\}$ to P_i.

Fig. 29. Macro for ideal MiniMAC authentication.

matrix of C. The Compress part of the protocol then takes as input all the m BigMAC shares of each party and simply uses the h'th component of the BigMAC authenticating the h'th component of $C(\mathbf{x})$ as the h'th component of the MiniMAC. That is, if we view the BigMACs as columns in a matrix (the first column being the BigMAC of the first component of $C(\mathbf{x})$ and so on up to the m'th component) then the MiniMAC of $C(\mathbf{x})$ is simply the diagonal of this matrix.

The intuition of why this is secure is that since the authentication of the parity components of the codeword are computed from the authenticated non-parity components using a public algorithm (the generator matrix) then an adversary can only try to cheat locally. Furthermore, if he later on tries to change the value that is MAC'ed to, then he will have to guess the honest parties' share of d components, because of the code's minimal distance.

CodeAuth Security.

Lemma 13. *For every static adversary $\mathcal{A}$ corrupting up to $n-1$ parties, the protocol Π_{CodeAuth} k securely implements $\mathcal{F}_{\mathsf{CodeAuth}}$ of Figure 31 in the $\mathcal{F}_{[\![\cdot]\!]}$-hybrid model.*

Proof. Let $\mathcal{S}$ be a simulator that has access to $\mathcal{F}_{\mathsf{CodeAuth}}$, we show that no environment $\mathcal{Z}$ can distinguish between an interaction with $\mathcal{S}$ and an interaction with the real adversary $\mathcal{A}$ and real parties with access to the functionality $\mathcal{F}_{[\![\cdot]\!]}$.

Protocol Π_{CodeAuth}

Initialize: Call $\mathcal{F}_{[\![\cdot]\!]}^{\mathbb{F}_{2^{u\cdot m}}}.(\mathsf{Init})$ to initialize the BigMAC key $\Delta \in \mathbb{F}_{2^{m\cdot u}}$.

BigMAC: On input $(\mathsf{BigMAC}, C, \mathbf{x}^{(i)})$ from every party P_i, where $\mathbf{x}^{(i)} \in \mathbb{F}_{2^u}^k$ and C is a systematic, linear code over $\mathbb{F}_{2^u}$ of dimension k and length m, do the following:

1. For each party $i \in [n]$ call $\mathcal{F}_{[\![\cdot]\!]}.(n\textbf{-Share})$ with input $(\mathsf{Authenticate}, \mathbf{x}^{(i)}[1], \ldots, \mathbf{x}^{(i)}[k])$ to obtain $\{\langle \mathbf{m}_h \rangle\}_{h\in[k]}$ and in turn the authenticated shares $\{[\![\mathbf{x}[h]]\!]\}_{h\in[k]} = \{\langle \mathbf{x}[h]\rangle, \langle \mathbf{m}_h\rangle, \langle \Delta\rangle\}_{h\in[k]}$.
2. Locally encode these shares using the code C, to obtain $[\![C(\mathbf{x})]\!]$, a length m vector of $\mathbb{F}_{2^u}$ elements, where every component is authenticated under Δ. That is,

$$[\![C(\mathbf{x})]\!] = \left(\{\langle C(\mathbf{x})[h]\rangle, \langle \mathbf{m}_h\rangle\}_{h\in[m]}, \langle \Delta\rangle\right) .$$

   Where $C(\mathbf{x})[h] = (\mathbf{x}\cdot G)[h]$ and $\mathbf{m}_h^{(i)} = \sum_{l=1}^k \mathbf{m}_l^{(i)} \cdot G[l,h]$ when G is the generator matrix of the code C and $\mathbf{x}$ is viewed as a row vector.

Compress: On input $(\mathsf{Compress}, [\![C(\mathbf{x})]\!])$ from all parties, do the following:

1. View $\Delta^{(i)}$ and $\mathbf{m}_h^{(i)}$ as elements of $\mathbb{F}_{2^u}^m$ by letting each block of u bits be one component in $\mathbb{F}_{2^u}$.
2. Parse $[\![C(\mathbf{x})]\!]$ as

$$\{\langle C(\mathbf{x})[h]\rangle, \langle \mathbf{m}_h\rangle, \langle \Delta\rangle\}_{h\in[m]} = \left\{\left\{C(\mathbf{x}^{(i)})[h], \mathbf{m}_h^{(i)}, \Delta^{(i)}\right\}_{h\in[m]}\right\}_{i\in[n]} .$$

3. Now define a new componentwise sharing $[\![C(\mathbf{x})]\!]^*$ to be

$$\{\langle C(\mathbf{x})\rangle, \langle \mathbf{m}\rangle, \langle \Delta\rangle\} = \left\{C(\mathbf{x}^{(i)}), \mathbf{m}^{(i)*}, \Delta^{(i)}\right\}_{i\in[n]} ,$$

   where $\mathbf{m}^{(i)*} \in \mathbb{F}_{2^u}^m$ and $\mathbf{m}^{(i)*}[h] = \mathbf{m}_h^{(i)}[h]$ for $h \in [m]$.

Fig. 30. Protocol Π_{CodeAuth} - Used for codeword authentication with a BigMAC key and a MiniMAC key.

The simulator invokes an internal copy of $\mathcal{A}$ and sets dummy parties $\pi_i, i \in \mathcal{P}$. Let A be the set of corrupt parties, it proceeds as follows:

1. Simulating the *Initialize phase*: $\mathcal{S}$ inputs (Init) to the $\mathcal{F}_{\mathsf{CodeAuth}}$ functionality, together with the set A of corrupt parties and all their extracted inputs $\left\{\Delta^{(i)}\right\}_{i\in A}$. It then runs an internal copy of $\mathcal{F}_{[\![\cdot]\!]}.(\mathsf{Init})$ using the shares it extracted from the adversary and the shares it got back from $\mathcal{F}_{\mathsf{CodeAuth}}$. If it receives Abort, then it forwards Abort to the $\mathcal{F}_{\mathsf{CodeAuth}}$ functionality, and it halts.
2. Simulating the *BigMAC phase*: $\mathcal{S}$ extracts the adversary's input to $\mathcal{F}_{[\![\cdot]\!]}.(n\textbf{-Share})$, $\left\{\mathbf{x}_h^{(i)}, \mathbf{m}_h^{(i)}\right\}_{h\in[k]}$ for $i \in \mathcal{P}$ and passes it on to $\mathcal{F}_{\mathsf{CodeAuth}}.\mathsf{BigMAC}$. It then picks the shares $\left\{\mathbf{x}_h^{(i)}\right\}_{h\in[k]}$ for $i \notin \mathcal{P}$ uniformly at random and sends it to $\mathcal{F}_{\mathsf{CodeAuth}}.\mathsf{BigMAC}$, which sends back the honest parties' MAC shares, $\left\{\mathbf{m}_h^{(i)}\right\}_{h\in[k]}$. If the adversary inputs $(\mathsf{Error}, \{e_{h,j}^{(i)}\}_{i\notin A, h\in[k], j\in[m\cdot u]})$ then the simulator passes on this call to $\mathcal{F}_{\mathsf{CodeAuth}}$ and locally updates the honest parties share by setting $\mathbf{m}_h^{(i)} = \mathbf{m}_h^{(i)} + \sum_{j=1}^{m\cdot u} e_{h,j}^{(i)} \cdot \Delta_j^{(i)} \cdot X^{j-1}$ where $\Delta_j^{(i)}$ denotes the j-th bit of $\Delta^{(i)}$.
3. Simulating the *Compress phase*: $\mathcal{S}$ simply passes on the call to the $\mathcal{F}_{\mathsf{CodeAuth}}$ functionality and returns what it gets back.

To argue indistinguishability first notice that if during the internal execution of the protocol an abort occurred, then an abort occurs in both the ideal and the real world, and the simulation in this case is perfect.

For the rest simply notice that everything is done in perfect accordance with the Π_{CodeAuth} (and $\mathcal{F}_{[\![\cdot]\!]}$) since everything is passed on directly to the $\mathcal{F}_{\mathsf{CodeAuth}}$ functionality or done with local computations.

Functionality $\mathcal{F}_{\mathsf{CodeAuth}}$

Let A be the indices of corrupt parties. Running with parties $P_1, \ldots, P_n$ and an ideal adversary $\mathcal{S}$, the functionality operates as follows.

Initialize: On input (Init) the functionality activates and waits for the adversary to input a set of shares $\{\Delta^{(j)}\}_{j \in A}$ in $\mathbb{F}_2^{m \cdot u}$. It samples random $\{\Delta^{(i)}\}_{i \notin A}$ in $\mathbb{F}_2^{m \cdot u}$ for the honest parties, defining $\Delta := \sum_{i \in [n]} \Delta^{(i)}$. If any $j \in A$ outputs Abort then the functionality aborts.

BigMAC: On input $(\mathsf{BigMAC}, C, \mathbf{x}^{(i)})$ from all parties P_i, where $\mathbf{x}^{(i)} \in \mathbb{F}_{2^u}^k$ and C is a systematic, linear code over $\mathbb{F}_{2^u}$ of dimension k and length m:

- Run the macro BigBracket with input $\left((\mathbf{x}^{(i)}[1], \Delta^{(i)}), \ldots, (\mathbf{x}^{(i)}[k], \Delta^{(i)}), C\right)$ from P_i.

Compress: On input $(\mathsf{Compress}, [\![C(\mathbf{x})]\!])$ from all parties P_i, do as follows:

1. Parse $[\![C(\mathbf{x})]\!]$ as

$$\{\langle C(\mathbf{x})[h]\rangle, \langle \mathbf{m}_h \rangle, \langle \Delta \rangle\}_{h \in [m]} = \left\{ \left\{ C(\mathbf{x}^{(i)})[h], \mathbf{m}_h^{(i)}, \Delta^{(i)} \right\}_{h \in [m]} \right\}_{i \in [n]} .$$

   Now define a new componentwise sharing $[\![C(\mathbf{x})]\!]^*$ to be

$$\{\langle C(\mathbf{x})\rangle, \langle \mathbf{m} \rangle, \langle \Delta \rangle\} = \left\{ C(\mathbf{x}^{(i)}), \mathbf{m}^{(i)*}, \Delta^{(i)} \right\}_{i \in [n]} ,$$

   where $\mathbf{m}^{(i)*} \in \mathbb{F}_{2^u}^m$ and $\mathbf{m}^{(i)*}[h] = \mathbf{m}_h^{(i)}[h]$ for $h \in [m]$.
2. Return $[\![C(\mathbf{x}^{(i)})]\!]^* = \left(C(\mathbf{x}^{(i)}), \mathbf{m}^{(i)*}, \Delta^{(i)}\right)$ to each party i.

Key queries: On input of a description of an affine subspace $S \subset (\mathbb{F}_2^{m \cdot u})^n$, return Success if $(\Delta^{(1)}, \ldots, \Delta^{(n)}) \in S$. Otherwise return Abort.

Fig. 31. Functionality $\mathcal{F}_{\mathsf{CodeAuth}}$ - Used for generating authenticated codewords.

G.3 Multiplication Triples

In this part we describe the remaining ideal functionalities, protocols and proofs needed in order to construct MiniMAC multiplication triples. First, we show in Fig. 32 how to use the amplified correlation OT functionality $\mathcal{F}_{\mathsf{ACOT}}$ from Fig. 3 to generate an XOR sharing of the tensor product of two unauthenticated codewords (one chosen by each party). We then describe the protocol $\Pi_{\mathsf{UncheckedMiniTriples}}$ in Fig. 33 (with ideal functionality $\mathcal{F}_{\mathsf{UncheckedMiniTriples}}$ described in Fig. 34 and its simulator $\mathcal{S}_{\mathsf{UncheckedMiniTriples}}$ described in Fig. 35) how to use these components of unauthenticated multiplication triples, along with the codeword authentication functionality $\mathcal{F}_{\mathsf{CodeAuth}}$ from Fig. 31, to construct unchecked MiniMAC multiplication triples. We then show the protocol $\Pi_{\mathsf{MiniTriples}}$ in Fig. 36 (whose ideal functionality is part of Fig. 28 and whose simulator $\mathcal{S}_{\mathsf{MiniTriples}}$ is described in Fig. 37) how to combine a pair of unchecked MiniMAC triples along with the Schur triple into a MiniMAC multiplication triple.

CodeOT Subprotocol. The CodeOT subprotocol uses $\mathcal{F}_{\mathsf{ACOT}}$ to create an XOR sharing of the componentwise product of vectors (over $\mathbb{F}_{2^u}$) input by two parties. It does so by first getting an XOR sharing of two $u \cdot k$ bit vectors from $\mathcal{F}_{\mathsf{ACOT}}$. These shares are then converted to k elements of the field $\mathbb{F}_{2^u}$ by viewing each element as a coefficient of an up to $u-1$ degree polynomial and then constructing the polynomial, that is an element of $\mathbb{F}_{2^u}$, by summing over the appropriate coefficients, multiplied with an X power to create a term. Finally, each row/column of this matrix is then expanded from $k \times k$ to $m \times m$ by viewing each row/column as an element in $\mathbb{F}_{2^u}^k$ and then using the linearity of C to encode each of these. This makes it possible for the parties to end up with an XOR sharing of the outer product of an encoding of a $\mathbb{F}_{2^u}^k$ value of each their choice.

Subprotocol CodeOT^C

Let C be a systematic, $[m, k, d]$ linear code over $\mathbb{F}_{2^u}$, and let s be a statistical security parameter.

Initialize: Run $\mathcal{F}_{\mathsf{ACOT}}.\mathsf{Initialize}$.
Input: P_R inputs $\mathbf{a} \in \mathbb{F}_{2^u}^k$ and P_S $\mathbf{b} \in \mathbb{F}_{2^u}^k$
Correlated OT: Run $\mathcal{F}_{\mathsf{ACOT}}^{u \cdot k, s}$ with input $\mathbf{a}, \mathbf{b}$, so P_R receives a matrix $T' \in \mathbb{F}_2^{u \cdot k \times u \cdot k}$ and P_S receives $Q' \in \mathbb{F}_2^{u \cdot k \times u \cdot k}$ such that

$$Q' = T' + \mathbf{a} \otimes \mathbf{b}$$

Convert to field:

1. Consider Q' and T' as $k \times k$ block matrices, where entry (i, j) is given by an XOR share of the $u \times u$ matrix over $\mathbb{F}_2$:
$$\begin{pmatrix} \mathbf{a}_i[1] \cdot \mathbf{b}_j[1] & \mathbf{a}_i[1] \cdot \mathbf{b}_j[2] & \dots & \mathbf{a}_i[1] \cdot \mathbf{b}_j[u] \\ \mathbf{a}_i[2] \cdot \mathbf{b}_j[1] & \mathbf{a}_i[2] \cdot \mathbf{b}_j[2] & \dots & \mathbf{a}_i[2] \cdot \mathbf{b}_j[u] \\ \vdots & & \ddots & \vdots \\ \mathbf{a}_i[u] \cdot \mathbf{b}_j[1] & \mathbf{a}_i[u] \cdot \mathbf{b}_j[2] & \dots & \mathbf{a}_i[u] \cdot \mathbf{b}_j[u] \end{pmatrix}$$
2. Let entry (i, j) be the $\mathbb{F}_{2^u}$ field element given by:
$$f_{i,j} = \sum_{i'=1}^{u} \sum_{j'=1}^{u} \mathbf{a}_i[i'] \cdot \mathbf{b}_j[j'] \cdot X^{i'+j'-2}$$
so now Q', T' are $k \times k$ matrices over $\mathbb{F}_{2^u}$, where entry (i, j) contains an XOR share of the product of $\mathbf{a}[i]$ and $\mathbf{b}[j]$.

Encode: Now expand Q' and T' into $m \times m$ matrices of codewords:

1. P_R sets T to be the matrix obtained by applying $C(\cdot)$ to each row and each column of T', seen as a vector in $\mathbb{F}_{2^u}^k$.
2. P_S sets Q to be the matrix obtained by applying $C(\cdot)$ to each row and each column of Q', seen as a vector in $\mathbb{F}_{2^u}^k$.

Note that Q and T are $u \cdot m \times u \cdot m$ matrices over $\mathbb{F}_2$, whose rows and columns are codewords in C when viewed as vectors in $\mathbb{F}_{2^u}^m$.
Note that if Q_i, T_i are the i-th *rows* of Q, T then $Q_i = T_i + \mathbf{a}_i \cdot \mathbf{b}$ for $i \in [m]$, and now every column of Q, T is a codeword.
Now P_R has $T \in \mathbb{F}_{2^u}^{m \times m}$ and P_S has $Q \in \mathbb{F}_{2^u}^{m \times m}$ such that:

$$Q = T + C(\mathbf{a}) \otimes C(\mathbf{b}).$$

Fig. 32. Subprotocol for codeword OT extension between P_r and P_s.

Since it only consists of local computation we do not model this with a separate functionality, instead just using it as a subprotocol in triple generation. We describe this subprotocol in Fig. 32.

Unchecked Multiplication Triples. The protocol $\Pi_{\mathsf{UncheckedMiniTriples}}$ constructs weakly authenticated multiplication triples. This is done by having each party pick two random elements in $\mathbb{F}_{2^u}^k$ and executing the CodeOT protocol with each other party on each of these elements to get an XOR sharing. Every party then computes a share of the Schur product based on his own chosen random values and the diagonal of each of the tensor products from CodeOT. Finally each party authenticates their respective shares using the $\mathcal{F}_{\mathsf{CodeAuth}}$ functionality.

Lemma 14. *For every static adversary $\mathcal{A}$ corrupting up to $n-1$ parties, the protocol $\Pi_{\mathsf{UncheckedMiniTriples}}$ k securely implements the $\mathcal{F}_{\mathsf{UncheckedMiniTriples}}$ functionality in the $(\mathcal{F}_{\mathsf{ACOT}}, \mathcal{F}_{\mathsf{CodeAuth}})$-hybrid model.*

Initialize: Call $\mathcal{F}_{\mathsf{CodeAuth}}.\mathsf{Initialize}$.

Triple Generation: This generates a triple $\{\langle C(\mathbf{a})\rangle, \langle C(\mathbf{b})\rangle, \langle C^*(\mathbf{c})\rangle\}$ with $C(\mathbf{a}), C(\mathbf{b}), C^*(\mathbf{c}) \in \mathbb{F}_{2^u}^m$ for which it holds that $C^*(\mathbf{c}) = C(\mathbf{a}) * C(\mathbf{b})$.

1. Each party P_i generates $\mathbf{a}^{(i)}, \mathbf{b}^{(i)} \xleftarrow{\$} \mathbb{F}_{2^u}^k$.
2. Each pair of parties (P_i, P_j) $(i \neq j)$ calls $\mathcal{F}_{\mathsf{CodeOT}}^C$ with input $\mathbf{a}^{(i)}, \mathbf{b}^{(j)}$, to obtain a random XOR sharing of the $m \times m$ matrix $C_i^{i,j} + C_j^{i,j} = C(\mathbf{a}^{(i)}) \otimes C(\mathbf{b}^{(j)})$. (Note the tensor product is over $\mathbb{F}_{2^u}$, so each entry of the matrix is a product of $\mathbb{F}_{2^u}$ elements.)
3. Each party P_i computes $C^*(\mathbf{c}^{(i)}) = C(\mathbf{a}^{(i)}) * C(\mathbf{b}^{(i)}) + \mathsf{diag}(\sum_{j \neq i} C_i^{i,j} + C_i^{j,i})$, where $\mathsf{diag}(M)$ is the vector containing the diagonal entries in the matrix M.
4. P_i calls $\mathcal{F}_{\mathsf{CodeAuth}}.\mathsf{BigMAC}$ with input $(C, \mathbf{a}^{(i)}), (C, \mathbf{b}^{(i)})$ and $(C^*, \mathbf{c}^{(i)})$ to obtain shares $[\![C(\mathbf{a})]\!], [\![C(\mathbf{b})]\!], [\![C^*(\mathbf{c})]\!]$ and then calls $\mathcal{F}_{\mathsf{CodeAuth}}.\mathsf{Compress}$ to obtain $[\![\cdot]\!]^*$ sharings.

Fig. 33. Protocol $\Pi_{\mathsf{UncheckedMiniTriples}}$ - Generation of unchecked MiniMAC triples

Functionality $\mathcal{F}_{\mathsf{UncheckedMiniTriples}}$

Let B denote the set of honest parties, and let $\hat{\imath}$ be the lowest index in B. Furthermore, let $B' = B \setminus \{\hat{\imath}\}$ and $A = [n] \setminus B$ the set of corrupted parties.

Initialize:

1. Sample $\Delta \xleftarrow{\$} \mathbb{F}_{2^{u \cdot m}}$ and output a random share $\Delta^{(i)}$ to P_i, consistent with shares for corrupted parties input by $\mathcal{S}$.

Triple generation:

1. Sample random shares of codewords $\langle C(\mathbf{a})\rangle, \langle C(\mathbf{b})\rangle$, using shares for corrupted parties input by the adversary.
2. Wait for $\mathcal{S}$ to input $\{\mathbf{f}_{\mathbf{a}}^{(i)}, \mathbf{f}_{\mathbf{b}}^{(i)}\}_{i \in B'}$, and $\mathbf{f} \in \mathbb{F}_{2^u}^k$.
3. Compute
$$C^*(\mathbf{c}) = C(\mathbf{a}) * C(\mathbf{b}) + \sum_{i \in B'} \left(C(\mathbf{a}^{(i)}) * C(\mathbf{f}_{\mathbf{a}}^{(i)}) + C(\mathbf{b}^{(i)}) * C(\mathbf{f}_{\mathbf{b}}^{(i)}) \right) + C(\mathbf{f})$$
and shares of $C^*(\mathbf{c})$ that are consistent with any adversarial inputs.
4. Run the macro $\mathsf{Bracket}$ on input $\langle C(\mathbf{a})\rangle, \langle C(\mathbf{b})\rangle, \langle C^*(\mathbf{c})\rangle$.
5. Output $[\![C(\mathbf{a})]\!]^*, [\![C(\mathbf{b})]\!]^*, [\![C^*(\mathbf{c})]\!]^*$.

Fig. 34. Functionality $\mathcal{F}_{\mathsf{UncheckedMiniTriples}}$ - Used for generation of unchecked MiniMAC triples

Proof. The simulation for unchecked MiniMAC triples, given in Fig. 35, is very close to the proof for unchecked SPDZ triples in $\mathbb{F}_{2^k}$. The main aspect of arguing indistinguishability is if the adversary inputs Error in one of the $\mathcal{F}_{\mathsf{ACOT}}$ instances. Following the same argument as the SPDZ proof, it follows that the resulting error term in $\mathbb{F}_{2^{k \cdot u}}$ in the real world is statistically close to uniform, corresponding to the uniform value $\mathbf{f}$ that is added in the simulation.

For the remaining indistinguishability argument, observe that the codeword $C^*(\mathbf{c})$ that results from the output of the simulation (from $\mathcal{F}_{\mathsf{UncheckedMiniTriples}}$), assuming the adversary does not input Error, is given by:

$$C^*(\mathbf{c}) = C(\mathbf{a}) * C(\mathbf{b}) + \sum_{i \in B'} \left(C(\mathbf{a}^{(i)}) * C(\mathbf{f}_{\mathbf{a}}^{(i)}) + C(\mathbf{b}^{(i)}) * C(\mathbf{f}_{\mathbf{b}}^{(i)}) \right)$$

where $\mathbf{f}_{\mathbf{a}}^{(i)}, \mathbf{f}_{\mathbf{b}}^{(i)}$ are adversarially chosen. In the protocol, if $\mathcal{A}$ does not input $\mathsf{MultError}$ to $\mathcal{F}_{\mathsf{ACOT}}$, we have

$$C^*(\mathbf{c}) = \sum_{i=1}^{n} C^*(\mathbf{c}^{(i)}) = \sum_{i=1}^{n} \left(C(\mathbf{a}^{(i)}) * C(\mathbf{b}^{(i)}) + \mathsf{diag}(\sum_{j \neq i}^{n} C_i^{i,j} + C_i^{j,i}) \right)$$

Simulator $\mathcal{S}_{\mathsf{UncheckedMiniTriples}}$

Initialize:

1. Receive $\Delta^{(i)} \in \mathbb{F}_2^{m \cdot u}$ for $i \in A$ as input to $\mathcal{F}_{\mathsf{CodeAuth}}$.
2. Input $\{\Delta^{(i)}\}_{i \in A}$ to $\mathcal{F}_{\mathsf{UncheckedMiniTriples}}$.

Triple generation:

1. Emulate $\mathcal{F}_{\mathsf{ACOT}}^{k \cdot u, k \cdot u}$ similarly to step 1 of the simulator for SPDZ triples (Fig. 26):
   - Receive inputs $\mathbf{a}^{(i,j)}, \mathbf{b}^{(i,j)}$ for $i \in A$ and $j \in B$ from corrupt parties P_i, corresponding to their input to the $\mathcal{F}_{\mathsf{ACOT}}$ instance with honest P_j.
   - Calculate the errors for these inputs as in Fig. 26, giving $\mathbb{F}_2^{u \cdot k}$ elements $\{\mathbf{f}_{\mathbf{a}}^{(i)}, \mathbf{f}_{\mathbf{b}}^{(i)}\}_{i \in B'}$, corresponding to multiplicative errors, and an additive error $\mathbf{f}$, which is uniformly random if the adversary input Error to $\mathcal{F}_{\mathsf{ACOT}}$ and zero otherwise.
   - For each $i \in A$ and $j \in B$, output a uniformly random matrix $U^{(i,j)} \in \mathbb{F}_2^{k \cdot u \times k \cdot u}$ as P_i output for $\mathcal{F}_{\mathsf{ACOT}}$ with P_j. Wait for Z to input $T^{(i,j)}$ and adjust the additive error $\mathbf{f}$ accordingly (as in Fig. 26).
2. Emulate $\mathcal{F}_{\mathsf{CodeAuth}}$ as follows:
   - Receive new shares $\mathbf{a}'^{(i)}, \mathbf{b}'^{(i)}, \mathbf{c}'^{(i)}$ for $i \in A$.
   - If $\mathcal{A}$ inputs $(\mathsf{Error}, e_{h,j}^{(i)})$ for honest $i \in B$ for one of the inputs, then submit the corresponding errors to $\mathcal{F}_{\mathsf{UncheckedMiniTriples}}$, along with the errors $\{\mathbf{f}_{\mathbf{a}}^{(i)}, \mathbf{f}_{\mathbf{b}}^{(i)}\}_{i \in B'}$, $\mathbf{f}$ and the shares $\mathbf{a}'^{(i)}, \mathbf{b}'^{(i)}, \mathbf{c}'^{(i)}$, passing the output to the adversary.

Fig. 35. Simulator for unchecked MiniMAC triples.

$$
\begin{aligned}
&= \sum_{i=1}^{n} \left(C(\mathbf{a}^{(i)}) * C(\mathbf{b}^{(i)}) + \mathsf{diag}(\sum_{j \neq i}^{n} C(\mathbf{a}^{(i,j)}) \otimes C(\mathbf{b}^{(j,i)})) \right) \\
&= \sum_{i=1}^{n} \left(C(\mathbf{a}^{(i)}) * C(\mathbf{b}^{(i)}) + \sum_{j \neq i}^{n} C(\mathbf{a}^{(i,j)}) * C(\mathbf{b}^{(j,i)}) \right) \\
&= C(\mathbf{a}) * C(\mathbf{b}) + \sum_{i \in B'} \left(C(\mathbf{a}^{(i)}) * C(\mathbf{f}_{\mathbf{b}}^{(i)}) + C(\mathbf{b}^{(i)}) * C(\mathbf{f}_{\mathbf{a}}^{(i)}) \right)
\end{aligned}
$$

where $\mathbf{f}_{\mathbf{a}}^{(i)}, \mathbf{f}_{\mathbf{b}}^{(i)}$ are the sum of the deviations in the adversary's interactions with P_i in $\mathcal{F}_{\mathsf{ACOT}}$. This clearly corresponds perfectly to the simulated result.

□

Checked Multiplication Triples. To construct fully secure MiniMAC multiplication triples we implement a pairwise sacrificing to verify both the MACs, the multiplicative relation and that shares indeed sum up to codewords.

To prove that this is enough to securely implement the triples stage of the functionality, we use the following proposition.

Proposition 1. *If the MiniMAC pairwise sacrifice is performed on any two triples where at least one is incorrect, then the probability the check passes is at most* $2^{-u \cdot \min(d,k)}$*, where* d *is the minimum distance and* k *is the dimension of the code over* $\mathbb{F}_{2^u}$*.*

Proof. This follows because the inputs to the sacrifice step are guaranteed to be authenticated codewords. Note that if the check passes, we have:

$$[\![C^* (\mathbf{t} * (\mathbf{c}_0 - \mathbf{a}_0 * \mathbf{b}_0))]\!]^* = [\![C^* (\mathbf{c}_1 - \mathbf{a}_1 * \mathbf{b}_1)]\!]$$

Initialization:

1. Call $\mathcal{F}_{\mathsf{UncheckedMiniTriples}}.\mathsf{Initialize}$.
2. Generate two unchecked triples $[\![C(\mathbf{a}_0)]\!]^*, [\![C(\mathbf{b}_0)]\!]^*, [\![C^*(\mathbf{c}_0)]\!]^*$ and $[\![C(\mathbf{a}_1)]\!]^*, [\![C(\mathbf{b}_1)]\!]^*, [\![C^*(\mathbf{c}_1)]\!]^*$ using $\mathcal{F}_{\mathsf{UncheckedMiniTriples}}$.
3. Generate a random Schur pair $[\![C(\mathbf{r})]\!]^*, [\![C^*(\mathbf{r}')]\!]^*$ using $\mathcal{F}_{\mathsf{Schur}}$.

Sacrificing:

1. Sample $\mathbf{t} \xleftarrow{\$} \mathbb{F}_{2^u}^k$ using $\mathcal{F}_{\mathsf{Rand}}$.
2. Open $C(\mathbf{t}) * C(\mathbf{b}_0) - C(\mathbf{b}_1)$ as $\boldsymbol{\rho}$ and $C(\mathbf{a}_0) + C(\mathbf{a}_1)$ as $\boldsymbol{\sigma}$.
3. Open $[\![C^*(\mathbf{c}_0)]\!]^* - [\![C^*(\mathbf{r}')]\!]^*$ to get $C^*(\mathbf{y})$.
4. Re-encode the first k components of $\mathbf{y}$ into C and add this to $[\![C(\mathbf{r})]\!]^*$ to get $[\![C(\mathbf{c}_0)]\!]^*$.
5. Locally compute

$$\begin{aligned}[\![C^*(\mathbf{x})]\!]^* &= C(\mathbf{t}) * [\![C(\mathbf{c}_0)]\!]^* + [\![C^*(\mathbf{c}_1)]\!] - [\![C(\mathbf{a}_0)]\!]^* * \boldsymbol{\rho} - [\![C(\mathbf{b}_1)]\!] * \boldsymbol{\sigma} \\ &= [\![C^*(\mathbf{t} * (\mathbf{c}_0 - \mathbf{a}_0 * \mathbf{b}_0) + \mathbf{c}_1 - \mathbf{a}_1 * \mathbf{b}_1)]\!]^*\end{aligned}$$

   and check the shares open to $\mathbf{0}$.
6. Check that all opened values are valid codewords in C^* and check their MACs using Π_{MACCheck}. If any checks fail, output Abort.
7. Output $([\![C(\mathbf{a}_0)]\!]^*, [\![C(\mathbf{b}_0)]\!]^*, [\![C^*(\mathbf{c}_0)]\!]^*)$ as a valid multiplication triple.

Fig. 36. Protocol $\Pi_{\mathsf{MiniTriples}}$ - Generation of MiniMAC triples

Simulator $\mathcal{S}_{\mathsf{MiniTriples}}$

Let A denote the set of corrupted parties and B the honest parties. Let $\mathcal{F}_{\mathsf{MiniTriples}}$ denote the functionality consisting of the **Multiplication** stage of $\mathcal{F}_{\mathsf{MiniPrep}}$.

Initialize:

1. Receive $\Delta^{(i)}$ for corrupt P_i for the $\mathsf{Initialize}$ step of $\mathcal{F}_{\mathsf{UncheckedMiniTriples}}$.
2. Input $\Delta^{(i)}$ to $\mathcal{F}_{\mathsf{MiniPrep}}$ for $i \in A$.

Triple generation:

1. Emulate $\mathcal{F}_{\mathsf{UncheckedMiniTriples}}$ for the both triples, receiving shares $C(\mathbf{a}_0^{(i)}), C(\mathbf{b}_0^{(i)})$ and $C(\mathbf{a}_1^{(i)}), C(\mathbf{b}_1^{(i)})$, as well as sets of errors E_j for $j \in B$ if the adversary inputs the Error flag.
2. Emulating $\mathcal{F}_{\mathsf{Schur}}$, receive shares $C(\mathbf{r}^{(i)}), C(\mathbf{r}'^{(i)})$ and corresponding MAC shares $C(\mathbf{m}_{\mathbf{r}}^{(i)}), C(\mathbf{m}_{\mathbf{r}'}^{(i)})$ for $i \in A$.
3. Emulating $\mathcal{F}_{\mathsf{UncheckedMiniTriples}}$, receive shares $C(\mathbf{c}_0^{(i)}), C(\mathbf{c}_1^{(i)})$ and MAC shares $C(\mathbf{m}_{\mathbf{a}_0}^{(i)}), C(\mathbf{m}_{\mathbf{b}_0}^{(i)})$, etc for $i \in A$.
4. Call $\mathcal{F}_{\mathsf{MiniTriples}}$ with input $C(\mathbf{a}_0^{(i)}), C(\mathbf{b}_0^{(i)}), C(\mathbf{c}_0^{(i)})$ and their MACs for corrupt P_i, along with the errors E_j for $j \in B$.
5. Emulating $\mathcal{F}_{\mathsf{Rand}}$, sample $\mathbf{t} \xleftarrow{\$} \mathbb{F}_{2^u}^k$ and send this to $\mathcal{A}$.
6. Perform each of the sacrifice checks as in the protocol, using shares input by the adversary for corrupt parties and generating random codewords for the partial openings of $\boldsymbol{\rho}, \boldsymbol{\sigma}, C^*(\mathbf{y})$ and $C^*(\mathbf{x})$. If any of the checks fail then abort.
7. Simulate the $\mathsf{MACCheck}$ procedure in the same manner as the $\mathcal{F}_{\mathsf{BatchCheck}}$ procedure in the $\mathbb{F}_{2^k}$ simulation (described in Section F.7). If the MAC check passes despite some errors introduced by the adversary, compute the solution spaces $S^{(i)} \subset \mathbb{F}_{2^u}^m$ corresponding to the possible values of each honest party's $\Delta^{(i)}$ for which the check would have passed, and submit $\{S^{(i)}\}_{i \in B}$ to the key query stage of $\mathcal{F}_{\mathsf{MiniPrep}}$. If this aborts then abort.

Fig. 37. Simulator for MiniMAC triples, after sacrificing.

for a uniformly random $\mathbf{t} \in \mathbb{F}_{2^u}^k$. By the minimum distance of the code C, $C(\mathbf{t})$ must be non-zero in at least d positions with overwhelming probability in $u \cdot k$. Now if one of the triples is incorrect, it follows that they

Protocol Π_{Schur}

Initialization: Call $\mathcal{F}_{\mathsf{CodeAuth}}.\mathsf{Initialize}$ with input (Init).

Schur Pair Generation:

1. Each party P_i generates random $\mathbf{r}^{(i)} \in \mathbb{F}_{2^u}^k$ and $\mathbf{r}^{(i)'} \in \mathbb{F}_{2^u}^{k^*-k}$. Let $\mathbf{0} \in \mathbb{F}_{2^u}^k$ be the 0-vector (in $\mathbb{F}_{2^u}$) of k elements. Now define $\mathbf{r}^{(i)''} = (\mathbf{0} \| \mathbf{r}^{(i)'}) \in \mathbb{F}_{2^u}^{k^*}$.
2. Call $\mathcal{F}_{\mathsf{CodeAuth}}.\mathsf{BigMAC}$ with input $(\mathsf{BigMAC}, C, \mathbf{r}^{(i)})$ and $(\mathsf{BigMAC}, C^*, \mathbf{r}^{(i)''})$ from P_i to obtain $[\![C(\mathbf{r})]\!]$ and $[\![C^*(\mathbf{r}'')]\!]$.
3. For $h \in [k]$ let $\mathbf{m}^{(i)}_{(\mathbf{r},h)}$ be the MAC share of the h'th components of P_i's shares of $[\![C^*(\mathbf{r}'')]\!]$. Each party commits to $\mathbf{m}^{(i)}_{(\mathbf{r},h)}$ using $\mathcal{F}_{\mathsf{Comm}}$. Then all parties open their commitments and check that $\sum_{i=1}^n \mathbf{m}^{(i)}_{(\mathbf{r},h)} = 0$ for $h \in [k]$.
4. Call $\mathcal{F}_{\mathsf{CodeAuth}}.\mathsf{Compress}$ with input $(\mathsf{Compress}, C, [\![C(\mathbf{r})]\!]))$ and $(\mathsf{Compress}, C^*, [\![C(\mathbf{r})]\!] + [\![C^*(\mathbf{r}'')]\!])$ to obtain $[\![C(\mathbf{r})]\!]^*$ and $[\![C^*(\mathbf{s})]\!]^*$, so that $\mathbf{r}$ is equal to $\mathbf{s}$ in the first k components.
5. Output $[\![C(\mathbf{r})]\!]^*, [\![C^*(\mathbf{s})]\!]^*$

Fig. 38. Protocol Π_{Schur} - Generation of Schur pairs.

both must be incorrect, otherwise just one side of the above equation would be zero. If j is an index of one of the (at least) d positions where $\mathbf{t}$ is non-zero then we can write

$$C(\mathbf{t})[j] = \left(C(\mathbf{c}_1[j]) - C(\mathbf{a}_1[j]) \cdot C(\mathbf{b}_1[j])\right) / \left(C(\mathbf{c}_0[j]) - C(\mathbf{a}_0[j]) \cdot C(\mathbf{b}_0[j])\right)$$

for every such j. Since $\mathbf{t}_j$ is uniformly random and unknown to the adversary when they choose shares of the triples, each of these equations is satisfied with probability 2^{-u}. However, note that $C(\mathbf{t})$ is fully determined by any of its k positions, so at most k positions are *independently* uniform. This means we get a total success probability of no more than $2^{-u \cdot \min(d,k)}$.

□

Lemma 15. *For every static adversary $\mathcal{A}$ corrupting up to $n-1$ parties, the protocol $\Pi_{\mathsf{MiniTriples}}$ implements the **Multiplication** stage of Fig. 28 in the $(\mathcal{F}_{\mathsf{Schur}}, \mathcal{F}_{\mathsf{CodeAuth}})$-hybrid model with statistical security $u \cdot \min(k, d)$, where $\mathcal{F}_{\mathsf{Schur}}$ is a functionality representing the **Schur pair** stage of Fig. 28.*

Proof. Correctness and indistinguishability of the transcripts when there are no errors follows from straightforward calculations, since the adversary's shares and errors in the protocol align exactly with what is allowed by the functionality, similarly to the protocol for $\mathbb{F}_{2^k}$ triples.

By Proposition 1, if the sacrifice check passes in the real world then with overwhelming probability we are guaranteed that the shares of triples are correct. This means that the errors $\mathbf{f}^{(i)}_{\mathbf{a}_b}$ and $\mathbf{f}^{(i)}_{\mathbf{b}_b}$ (for $b = 0, 1$) submitted to $\mathcal{F}_{\mathsf{UncheckedMiniTriples}}$ were all zero. In this case the only errors possible (in both worlds) are the MAC errors, which are modeled identically in both the protocol and the functionality, so will be distributed the same. Finally, the MAC checking stage is simulated as in the proof of $\mathbb{F}_{2^k}$ triples – the simulator computes the solution space (in the unknown honest parties' shares of Δ) for which the adversary's inputs to the MAC check would pass, and tests this using the key query stage of $\mathcal{F}_{\mathsf{MiniPrep}}$. This implies that the probability of abort corresponds to the probability of passing the MAC check in the real world.

□

G.4 Schur Pairs

The construction of a Schur pair is described by the Π_{Schur} protocol (Fig. 38). It consists of having each party generate a random element in $\mathbb{F}_{2^u}^k$ and a random element of $\mathbb{F}_{2^u}^{k^*}$, but under the constraint that the first k elements are 0. The random element in $\mathbb{F}_{2^u}^k$ is then encoded and BigMACed with C using $\mathcal{F}_{\mathsf{CodeAuth}}$. Similarly the random element in $\mathbb{F}_{2^u}^{k^*}$ is then encoded and BigMACed with C^* using $\mathcal{F}_{\mathsf{CodeAuth}}$. The first is compressed, so is the addition of the two. This results in two MiniMAC authentications of the same same element, but where one is in C and the other in C^*.

Simulator $\mathcal{S}_{\mathsf{Schur}}$

Let A denote the set of corrupted parties.

Initialization: On input (Init) from all parties, receive $\left\{\Delta^{(i)}\right\}_{i\in A}$. It then passes on these values to the ideal **Initialize** call in $\mathcal{F}_{\mathsf{CodeAuth}}$ and receives back the shares $\left\{\Delta^{(i)}\right\}_{i\notin A}$, defining $\Delta = \sum_{i\in[n]}\Delta^{(i)}$.

Schur Pair Generation:

1. Receive the shares $\left\{\mathbf{r}^{(i)}, \left\{\mathbf{m}^{(i)}_{(\mathbf{r},h)}\right\}_{h\in[k]}\right\}$, respectively $\left\{\mathbf{r}^{(i)''}, \left\{\mathbf{m}^{(i)}_{(\mathbf{r}'',h)}\right\}_{h\in[k^*]}\right\}$ for each corrupted party $i \in A$ from the environment as input to $\mathcal{F}_{\mathsf{CodeAuth}}.\mathsf{BigMAC}$ for $[\![C(\mathbf{r})]\!]$, respectively $[\![C^*(\mathbf{r}'')]\!]$. If $\mathbf{r}'' \neq \mathbf{0}\|\mathbf{r}'$ for some $\mathbf{r}' \in \mathbb{F}_{2^u}^{k^*}$ then output **abort**.
2. It then passes the values $C(\mathbf{r}^{(i)})$, $C^*(\mathbf{r}^{(i)} + \mathbf{r}^{(i)''})$ and $\mathbf{m}^{(i)}_{\mathbf{r}}$, $\mathbf{m}^{(i)}_{\mathbf{r}} + \mathbf{m}^{(i)}_{\mathbf{r}''}$ on to the ideal functionality for Schur pair generation on behalf of the corrupted parties. Here it lets $\mathbf{m}^{(i)}_{\mathbf{r}}[h] = \mathbf{m}^{(i)}_{(\mathbf{r},h)}[h]$ and $(\mathbf{m}^{(i)}_{\mathbf{r}} + \mathbf{m}^{(i)}_{\mathbf{r}''})[h] = (\mathbf{m}^{(i)}_{(\mathbf{r},h)} + \mathbf{m}^{(i)}_{(\mathbf{r}'',h)})[h]$ for $h \in [k]$ and then for $h \in [k+1;m]$ it defines the shares $\mathbf{m}^{(i)}_{(\mathbf{r},h)}[h] = \sum_{l=1}^{k}\mathbf{m}^{(i)}_{(\mathbf{r},l)}[h]\cdot G[l,h]$, where G is the generator matrix for C. Similarly for the shares of $\mathbf{m}^{(i)}_{\mathbf{r}} + \mathbf{m}^{(i)}_{\mathbf{r}''}$.
3. The simulator receives back from the functionality the shares $\left\{C(\mathbf{r}^{(i)}), \mathbf{m}^{(i)}_{\mathbf{r}}\right\}$ and $\left\{C^*(\mathbf{s}^{(i)}), \mathbf{m}^{(i)}_{\mathbf{s}}\right\}$ for each $i \notin A$, under the constraint that $\mathbf{m}_{\mathbf{r}} = C(\mathbf{r}) * \Delta$ and $\mathbf{m}_{\mathbf{s}} = C^*(\mathbf{s}) * \Delta$ (where $\Delta^{(i)}$ for $i \in A$ has previously been given by the environment during initialization).
4. The simulator then emulates the execution of $(\mathsf{BigMAC}, C, \mathbf{r}^{(i)})$ and $(\mathsf{BigMAC}, C^*, \mathbf{r}^{(i)''})$ by using the shares it got from the adversary and the ideal functionality. More specifically for $h \in [k]$ when $i \notin A$ it picks a uniformly random value $\mathbf{m}^{(i)}_{(\mathbf{r},h)} \in \mathbb{F}_{2^{u\cdot m}}$ under the constraint that $\mathbf{m}^{(i)}_{(\mathbf{r},h)}[h] = \mathbf{m}^{(i)}_{\mathbf{r}}[h]$ and when $i \in A$ it uses the values given by the adversary. For $h \in [k+1;m]$ and all $i \in [n]$ it sets $\mathbf{m}^{(i)}_{(\mathbf{r},h)}[h] = \sum_{l=1}^{k}\mathbf{m}^{(i)}_{(\mathbf{r},l)}[h]\cdot G[l,h]$, where G is the generator matrix for C. Similarly for the shares of $\mathbf{m}^{(i)}_{\mathbf{r}} + \mathbf{m}^{(i)}_{\mathbf{r}''}$.
5. If the adversary inputs $(\mathsf{Error}, \{e^{(i)}_{h,j}\}_{i\notin A, h\in[k], j\in[m\cdot u]})$ with elements in $\mathbb{F}_{2^{m\cdot u}}$ then it locally sets $\mathbf{m}^{(i)}_{(\mathbf{r},h)} = \mathbf{m}^{(i)}_{(\mathbf{r},h)} + \sum_{j=1}^{m\cdot u} e^{(i)}_{h,j}\cdot\Delta^{(i)}_j\cdot X^{j-1}$ where $\Delta^{(i)}_j$ denotes the j-th bit of $\Delta^{(i)}$. Similarly for the shares of $\mathbf{m}^{(i)}_{\mathbf{r}} + \mathbf{m}^{(i)}_{\mathbf{r}''}$.
6. The simulator receives the shares of the MAC on the first k components and then broadcasts the shares of the MAC of the honest parties, emulating $\mathcal{F}_{\mathsf{Comm}}$. It checks that all shares sum up to 0 and outputs **abort** if that is not the case.
7. It then emulates the **Compress** phase of $\mathcal{F}_{\mathsf{CodeAuth}}$. It does so just like the protocol Π_{CodeAuth} since this part is non-interactive.

Fig. 39. Simulator for generation of Schur pairs.

Lemma 16. *For every static adversary $\mathcal{A}$ corrupting up to $n-1$ parties, the protocol Π_{Schur} in Fig. 38 k securely implements the **Schur Pair** call of Fig. 28 in the $(\mathcal{F}_{\mathsf{CodeAuth}}, \mathcal{F}_{\mathsf{Comm}})$-hybrid model.*

Proof. We use the simulator $\mathcal{S}_{\mathsf{Schur}}$ of Figure 39 which has access to the functionality **Schur Pair** of Fig. 28 and show that no environment $\mathcal{Z}$ can distinguish between an interactions with $\mathcal{S}_{\mathsf{Schur}}$ and a real adversary $\mathcal{A}$ and real parties with access to the functionality $\mathcal{F}_{\mathsf{CodeAuth}}$. This is relatively straightforward; we see that in the simulation, like the protocol, each malicious party gets to choose its shares $\mathbf{r}^{(i)}$, $\mathbf{r}^{(i)'}$ and that $\mathbf{r}^{(i)''}$ is uniquely defined from $\mathbf{r}^{(i)'}$. Using these values and the adversarial shares of the MACs given to the BigMAC call we construct each adversarial share of the last $m-k$ components of $\mathbf{m}^{(i)}_{\mathbf{r}}$, respectively $\mathbf{m}^{(i)}_{\mathbf{s}}$, just like they would be after the call to $\mathsf{Compress}$ in the end of the protocol. So for the corrupted parties the simulator is indistinguishable from the protocol.

Next, consider the honest parties $i \notin A$. We notice that based on the output of the ideal functionality we simulate their shares in the BigMAC call exactly like in Π_{Schur} by picking the values uniformly at random and using the parts $\mathbf{m}^{(i)}_{(\mathbf{r},h)}[h]$ we got from the ideal functionality while keeping the constraint that $\mathbf{m}_{(\mathbf{r},h)} = \mathbf{r}[h]\cdot\Delta$. We see that for $h \in [k]$ this follows since we are given the adversarial values for $\mathbf{m}^{(i)}_{(\mathbf{r},h)}$ and that we are free

Protocol Π_{Reorg}

Initialize: Take as input a linear function $f : \mathbb{F}_{2^u}^k \to \mathbb{F}_{2^u}^k$ and call $\mathcal{F}_{\mathsf{CodeAuth}}.\mathsf{Initialize}$ with input (Init).
Reorganization Pair Generation:

1. Call Π_{CodeAuth} with input $(\mathsf{BigMAC}, C, \mathbf{r}^{(i)})$, for random $\mathbf{r}^{(i)} \in \mathbb{F}_{2^u}^k$ from P_i, to obtain $[\![C(\mathbf{r})]\!]$.
2. Let $[\![\mathbf{r}_{(1)}]\!], \ldots, [\![\mathbf{r}_{(k)}]\!]$ be the first k components of the BigMAC-authenticated codeword $[\![C(\mathbf{r})]\!]$.
3. Each party locally applies f and then C to $[\![\mathbf{r}_{(1)}]\!], \ldots, [\![\mathbf{r}_{(k)}]\!]$, to obtain $[\![C(f(\mathbf{r}))]\!]$.
4. Call Π_{CodeAuth} with input $(\mathsf{Compress}, C, [\![C(\mathbf{r})]\!])$ and $(\mathsf{Compress}, C, [\![C(f(\mathbf{r}))]\!])$ to obtain $[\![C(\mathbf{r})]\!]^*$ and $[\![C(f(\mathbf{r}))]\!]^*$
5. Output $[\![C(\mathbf{r})]\!]^*, [\![C(f(\mathbf{r}))]\!]^*$

Fig. 40. Protocol Π_{Reorg} - Generation of pairs of authenticated codewords for reorganization of elements.

to choose the values for the honest parties (of which there will be at least one), thus this is trivial to ensure since we also know $\Delta^{(i)}$ for all parties from the Initialization. Since the last $m-k$ values are uniquely defined from the first k values, by the generator matrix G, and since we have correctness for $\mathcal{F}_{\mathsf{CodeAuth}}.\mathsf{Compress}$ this also follows trivially. Similarly for the $\mathbf{m}^{(i)}_{(\mathbf{r}'',h)}$ values.

Now assume w.l.o.g. that there is only one honest party, party i (more honest parties will give us more wriggle room). To see we can still keep the constraint while using the component $\mathbf{m}^{(i)}_{(\mathbf{r},h)}[h]$ given by the ideal functionality for $h \in [k+1; m]$, first remember that this component is uniquely determined by the formula $\mathbf{m}^{(i)}_{(\mathbf{r},h)}[h] = C(\mathbf{r})[h] \cdot \Delta[h] - \sum_{j \in A} \left(\sum_{l=1}^{k} \mathbf{m}^{(j)}_{(\mathbf{r},l)}[h] \cdot G[l,h] \right)$ in the ideal functionality. We simulate this share by defining it to be

$$
\begin{aligned}
\mathbf{m}^{(i)}_{(\mathbf{r},h)}[h] :&= \sum_{l=1}^{k} \mathbf{m}^{(i)}_{(\mathbf{r},l)}[h] \cdot G[l,h] \\
&= \sum_{l=1}^{k} \left(C(\mathbf{r})[l] \cdot \Delta[h] - \left(\sum_{j \in A} \mathbf{m}^{(j)}_{(\mathbf{r},l)}[h] \right) \right) \cdot G[l,h] \\
&= \left(\sum_{l=1}^{k} C(\mathbf{r})[l] \cdot \Delta[h] \cdot G[l,h] \right) - \sum_{l=1}^{k} \left(\sum_{j \in A} \mathbf{m}^{(j)}_{(\mathbf{r},l)}[h] \right) \cdot G[l,h] \\
&= \left(\sum_{l=1}^{k} C(\mathbf{r})[l] \cdot G[l,h] \right) \cdot \Delta[h] - \sum_{j \in A} \sum_{l=1}^{k} \mathbf{m}^{(j)}_{(\mathbf{r},l)}[h] \cdot G[l,h] \\
&= C(\mathbf{r})[h] \cdot \Delta[h] - \sum_{j \in A} \sum_{l=1}^{k} \mathbf{m}^{(j)}_{(\mathbf{r},l)}[h] \cdot G[l,h]
\end{aligned}
$$

We see that they are both equally defined and so the simulation is good.

Next we see that the potential input of Error by the adversary is emulated like in the protocol. Then we notice that the MAC check is also handled (without interaction) as in the protocol and the same goes for the $\mathsf{Compress}$ call and so we are done.

G.5 Reorganization Pairs

The reorganization pairs are constructed in almost the same manner as the Schur pairs: Each party selects a random element in $\mathbb{F}_{2^u}^k$ and authenticates it with a BigMAC using $\mathcal{F}_{\mathsf{CodeAuth}}$. The linear function $f : \mathbb{F}_{2^u}^k \to \mathbb{F}_{2^u}^k$, given as input, is then applied component wise to each of the authenticated components. Finally, the results are compressed to MiniMACs.

Simulator $\mathcal{S}_{\mathsf{ReOrg}}$

Initialization: On input (Init) from all parties, receive $\left\{\Delta^{(i)}\right\}_{i\in A}$. It then passes on these values to the ideal **Initialize** functionality in $\mathcal{F}_{\mathsf{CodeAuth}}$ and receives back the shares $\left\{\Delta^{(i)}\right\}_{i\notin A}$, defining $\Delta = \sum_{i\in[n]} \Delta^{(i)}$.

Reorganization Pair Generation:

1. Receive the shares $\left\{\mathbf{r}^{(i)}, \left\{\mathbf{m}^{(i)}_{(\mathbf{r},h)}\right\}_{h\in[k]}\right\}$ for each corrupted party $i \in \mathcal{A}$ from the environment as input to $\mathcal{F}_{\mathsf{CodeAuth}}.\mathsf{BigMAC}$ for $[\![C(\mathbf{r})]\!]$. It then passes these values on to the ideal functionality for Reorganization pair generation.
2. If the adversary inputs $(\mathsf{Error}, \{e^{(i)}_{h,j}\}_{i\notin A, h\in[k], j\in[m\cdot u]})$ with elements in $\mathbb{F}_{2^{m\cdot u}}$, it passes on the call to the ideal functionality.
3. The simulator receives back from the functionality the shares $\left\{C(\mathbf{r}^{(i)}), \left\{\mathbf{m}^{(i)}_{(\mathbf{r},h)}\right\}_{h\in[k]}\right\}$ for the honest parties.
4. The simulator then emulates the execution of $(\mathsf{BigMAC}, C, \mathbf{r}^{(i)})$ by using exactly the shares it got from the adversary and the ideal functionality.
5. The simulator applies f and then C to $[\![\mathbf{r}[1]]\!], \ldots, [\![\mathbf{r}[k]]\!]$, to obtain $[\![C(f(\mathbf{r}))]\!]$, just like in the real protocol.
6. It then emulates the **Compress** phase of $\mathbb{F}_{\mathsf{ReOrg}}$. It does so just like the protocol Π_{Reorg} since this part is non-interactive.

Fig. 41. Simulator for generation of Reorganization Pairs.

Lemma 17. *For every static adversary $\mathcal{A}$ corrupting up to $n-1$ parties, the protocol Π_{Reorg} in Fig. 40 k securely implements the **Reorganization** call of Fig. 28 in the $\mathcal{F}_{\mathsf{CodeAuth}}$-hybrid model.*

Proof. We use the simulator $\mathcal{S}_{\mathsf{ReOrg}}$ of Figure 41 which has access to the functionality **Schur Pair** of Fig. 28 and show that no environment $\mathcal{Z}$ can distinguish between an interactions with $\mathcal{S}_{\mathsf{ReOrg}}$ and a real adversary $\mathcal{A}$ and real parties with access to the functionality $\mathcal{F}_{\mathsf{CodeAuth}}$. This is relatively straightforward; we see that in the simulation, like the protocol, each malicious party gets to choose its shares $\mathbf{r}^{(i)}$. Using these values and the adversarial shares of the MACs given to the BigMAC call, the simulator uses the ideal functionality to get the adversarial shares $\left\{\mathbf{m}^{(i)}_{(\mathbf{r},h)}\right\}_{h\in[k+1;m]}$ for $i \in A$. These shares are constructed by the ideal functionality exactly like in the real protocol so for the corrupted parties the simulator is indistinguishable from the protocol.

Next, consider the honest parties $i \notin A$. We notice that the output of the ideal functionality for the honest parties is consistent with the values constructed by the BigMAC call in the real protocol.

Finally we see that the potential input of Error by the adversary is emulated similarly to the real protocol. Then we notice that the construction of $[\![C(f(\mathbf{r}))]\!]$ is also handled (without interaction) as in the real protocol and the same goes for the Compress call and so we are done.

H MiniMAC Online Phase

For completeness we here describe the online phase of the MiniMAC protocol. Furthermore, we have introduced a few small optimizations, such as removing the need for a public codeword and a more efficient amortized MAC check, so our online phase is slightly different from the one by Damgård et al. [11].

First consider arithmetic operations on shares: Remember that shares $[\![C(\mathbf{x})]\!]^*$ and $[\![C(\mathbf{y})]\!]^*$ are given by triples $(\langle C(\mathbf{x})\rangle, \langle \mathbf{m}_{\mathbf{x}}\rangle, \langle \Delta\rangle)$ and $(\langle C(\mathbf{y})\rangle, \langle \mathbf{m}_{\mathbf{y}}\rangle, \langle \Delta\rangle)$ respectively. Now see how we do some basic operations on such shares:

$$[\![C(\mathbf{x})]\!]^* + [\![C(\mathbf{y})]\!]^* = \Big(\big(C(\mathbf{x}^{(1)}) + C(\mathbf{y}^{(1)}), C(\mathbf{x}^{(2)}) + C(\mathbf{y}^{(2)}), \ldots, C(\mathbf{x}^{(n)}) + C(\mathbf{y}^{(n)})\big),$$

$$
\begin{aligned}
&\left(\mathbf{m}_{\mathbf{x}}^{(1)}+\mathbf{m}_{\mathbf{y}}^{(1)}, \mathbf{m}_{\mathbf{x}}^{(2)}+\mathbf{m}_{\mathbf{y}}^{(2)}, \ldots \mathbf{m}_{\mathbf{x}}^{(n)}+\mathbf{m}_{\mathbf{y}}^{(n)}\right), \langle \Delta \rangle\Big) \\
&= [\![C\left(\mathbf{x}+\mathbf{y}\right)]\!]^{*} \\
C(\mathbf{x})+[\![C\left(\mathbf{y}\right)]\!]^{*} &= \Big(\Big(C(\mathbf{x})+C(\mathbf{y}^{(1)}), C(\mathbf{y}^{(2)}), \ldots, C(\mathbf{y}^{(n)})\Big), \\
&\quad \Big(C(\mathbf{x}) * \Delta^{(1)}+\mathbf{m}_{\mathbf{y}}^{(1)}, C(\mathbf{x}) * \Delta^{(2)}+\mathbf{m}_{\mathbf{y}}^{(2)}, \ldots, \\
&\quad\quad C(\mathbf{x}) * \Delta^{(n)}+\mathbf{m}_{\mathbf{y}}^{(n)}\Big), \langle \Delta \rangle\Big) \\
&= [\![C\left(\mathbf{x}+\mathbf{y}\right)]\!]^{*} \\
C(\mathbf{x}) * [\![C\left(\mathbf{y}\right)]\!]^{*} &= \Big(\Big(C(\mathbf{x}) * C(\mathbf{y}^{(1)}), C(\mathbf{x}) * C(\mathbf{y}^{(2)}), \ldots, C(\mathbf{x}) * C(\mathbf{y}^{(n)})\Big), \\
&\quad \Big(C(\mathbf{x}) * \mathbf{m}_{\mathbf{y}}^{(1)}, C(\mathbf{x}) * \mathbf{m}_{\mathbf{y}}^{(2)}, \ldots, C(\mathbf{x}) * \mathbf{m}_{\mathbf{y}}^{(n)}\Big), \langle \Delta \rangle\Big) \\
&= [\![C^{*}\left(\mathbf{x} * \mathbf{y}\right)]\!]^{*}
\end{aligned}
$$

We now present the online protocol in Fig. 42 which uses the MAC checking procedure from Fig. 16. It also uses the ideal triple macros from Fig. 29 and assumes pools of preprocessed material in accordance with the ideal descriptions in Fig. 27 and 28.

Initialize: The parties first invoke the preprocessing protocols to get a sufficient amount of multiplication triples $\{[\![C(\mathbf{a})]\!]^*, [\![C(\mathbf{b})]\!]^*, [\![C^*(\mathbf{c})]\!]^*\}$, Schur triples $\{[\![C(\mathbf{r})]\!]^*, [\![C^*(\mathbf{s})]\!]^*\}$ and reorganization triples for each of the different linear functions needed $\{[\![C(\mathbf{r})]\!]^*, [\![C(f(\mathbf{r}))]\!]^*\}$.

Rand: The parties call $(\mathsf{BigMAC}, C, \mathbf{r}^{(i)})$ for a random $\mathbf{r}^{(i)} \in \mathbb{F}_{2^u}^k$ and thus learn a sharing $[\![C(\mathbf{r})]\!]$. This is followed by a call to $(\mathsf{Compress}, [\![C(\mathbf{r})]\!])$ to learn a sharing $[\![C(\mathbf{r})]\!]^*$.

Input: To share party P_i's input $\mathbf{x} \in \mathbb{F}_{2^u}^k$ all parties call $(\mathsf{BigMAC}, C, \mathbf{r}^{(i)})$ for a random $\mathbf{r}^{(i)} \in \mathbb{F}_{2^u}^k$ and thus learn a sharing $[\![C(\mathbf{r})]\!]$. This is followed by a call to $(\mathsf{Compress}, [\![C(\mathbf{r})]\!])$ to learn a sharing $[\![C(\mathbf{r})]\!]^*$. Party P_i then locally computes the encoding of $\mathbf{x}$, $C(\mathbf{x}) \in \mathbb{F}_{2^u}^m$ and then does as follows:

1. Party P_i sets $\mathbf{m}^{(i)} := \mathbf{m}^{(i)} + \left(C(\mathbf{x}) - C(\mathbf{r}^{(i)})\right) \cdot \Delta^{(i)}$ and stores $C(\mathbf{x})$ instead of $C(\mathbf{r}^{(i)})$ as its message share.
2. Party P_i then sends $\boldsymbol{\delta} = C(\mathbf{x}) - C(\mathbf{r}^{(i)})$ to party P_j for each $j \neq i$.
3. Each party P_j where $j \neq i$ verifies that $\boldsymbol{\delta}$ is in fact a codeword and aborts if that is not the case. P_j then sets $\mathbf{m}^{(j)} := \mathbf{m}^{(j)} + \boldsymbol{\delta} \cdot \Delta^{(j)}$.

Add: To add the triples $[\![C(\mathbf{x})]\!]^*$ and $[\![C(\mathbf{y})]\!]^*$, the parties locally compute $[\![C(\mathbf{x}+\mathbf{y})]\!]^* = [\![C(\mathbf{x})]\!]^* + [\![C(\mathbf{y})]\!]^*$.

Multiply: To multiply $[\![C(\mathbf{x})]\!]^*$ and $[\![C(\mathbf{y})]\!]^*$ the parties take a multiplication triple $\{[\![C(\mathbf{a})]\!]^*, [\![C(\mathbf{b})]\!]^*, [\![C^*(\mathbf{c})]\!]^*\}$ and a Schur pair $\{[\![C(\mathbf{r})]\!]^*, [\![C^*(\mathbf{s})]\!]^*\}$ from the pool of available ones and do:

1. Partially open $[\![C(\mathbf{x})]\!]^* - [\![C(\mathbf{a})]\!]^*$ to get $\boldsymbol{\epsilon}$ and $[\![C(\mathbf{y})]\!]^* - [\![C(\mathbf{b})]\!]^*$ to get $\boldsymbol{\delta}$.
2. Compute $[\![C^*(\mathbf{c})]\!]^* + \boldsymbol{\epsilon} * [\![C(\mathbf{b})]\!]^* + \boldsymbol{\delta} * [\![C(\mathbf{a})]\!]^* + \boldsymbol{\epsilon} * \boldsymbol{\delta} = [\![C^*(\mathbf{x} * \mathbf{y})]\!]^*$.
3. Partially open $[\![C^*(\mathbf{x} * \mathbf{y})]\!]^* - [\![C^*(\mathbf{s})]\!]^*$ to get $C^*(\mathbf{x} * \mathbf{y} - \mathbf{s}) = \boldsymbol{\eta}^* \in C^*$. Party P_1 then computes $C(\mathbf{x} * \mathbf{y} - \mathbf{r}) = \boldsymbol{\eta}$, which he broadcasts.
4. All parties then check that both $\boldsymbol{\eta}$ and $\boldsymbol{\eta}^*$ are in fact codewords for the same value. The parties then locally compute $\boldsymbol{\eta} + [\![C(\mathbf{r})]\!]^*$.

Reorganize: Let $\mathbf{x} = (x_1, x_2, \ldots, x_k)$ be the vector of blocks containing the output bits of a given layer in the circuit. To reorganize these bits as inputs for the next layer the parties first identify the f matching this reorganizing and take the preprocessed pair $\{[\![C(\mathbf{r})]\!]^*, [\![C(f(\mathbf{r}))]\!]^*\}$. The parties then do as follows:

1. Partially open $[\![C(\mathbf{x})]\!]^* - [\![C(\mathbf{r})]\!]^* = [\![C(\mathbf{x}-\mathbf{r})]\!]^*$.
2. The parties then extracts $\mathbf{x}-\mathbf{r}$ from $C(\mathbf{x}-\mathbf{r})$ and locally computes $C(f(\mathbf{x}-\mathbf{r}))$.
3. The parties then compute $[\![C(f(\mathbf{x}))]\!]^* = C(f(\mathbf{x}-\mathbf{r})) + [\![C(f(\mathbf{r}))]\!]^*$.

Output: This stage is entered when the parties have $[\![C(\mathbf{x})]\!]^*$ for (possibly incorrect) but not opened output value $\mathbf{x}$.

1. Let $[\![\mathbf{z}_1]\!]^*, [\![\mathbf{z}_2]\!]^*, \ldots, [\![\mathbf{z}_n]\!]^*$ be all partially opened values so far, the parties then call $\mathsf{MACCheck}$ with their shares of these values.
2. The parties then partially open $[\![C(\mathbf{x})]\!]^*$ and call $\mathsf{MACCheck}$ with their shares of this value.

If no calls to $\mathsf{MACCheck}$ aborts the parties output $\mathbf{x}$.

Fig. 42. Protocol Online - MiniMAC online protocol