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

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 [19] 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. [16]. 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 [19] 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. [17] 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 of independent interest. Additionally, we revisit the multi-party TinyOT protocol by Larraia et al. from CRYPTO 2014 [17], 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. [17]. 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 [15]. 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.

Finite field	Protocol	# Correlated OTs	# Random OTs	Time estimate, ms ($n = 2$)
$\mathbb{F}_2$	2-party TinyOT [19,5]	0	54	0.07
	n-party TinyOT [17,5]	$81n(n-1)$	$27n(n-1)$	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.

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. [19], 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. [17] has two main stages: first, unauthenticated shares of triples are created (using the aBit protocol by Nielsen et al. [19] 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. [19]. 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 [19,17]

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"

[3] We stress that this attack only applies to the multi-party protocol from CRYPTO 2014 [17], and not the original two-party protocol of Nielsen et al. [19].

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 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 [19,17], we use just a passively secure protocol, which is simply the passive OT extension of Ishai et al. [13], 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.

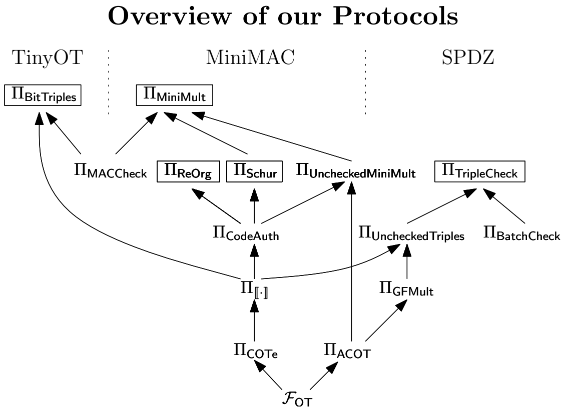
Overview of our Protocols
TinyOT
MiniMAC
SPDZ
$\Pi_{\mathsf{BitTriples}}$
Π_{MiniMult}
Π_{MACCheck}
Π_{ReOrg}
Π_{Schur}
$\Pi_{\mathsf{UncheckedMiniMult}}$
$\Pi_{\mathsf{TripleCheck}}$
Π_{CodeAuth}
$\Pi_{\mathsf{UncheckedTriples}}$
$\Pi_{\mathsf{BatchCheck}}$
$\Pi_{[\![\cdot]\!]}$
Π_{GFMult}
Π_{COTe}
Π_{ACOT}
$\mathcal{F}_{\mathsf{OT}}$

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 the full version 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 $H : \{0,1\}^* \rightarrow \{0,1\}^\kappa$, 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. Many protocols and proofs are omitted due to space reasons; we refer the reader to the full version for details [12].

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 $a \overset{\$}{\leftarrow} A$ the random sampling of a from a distribution A, and by $[d]$ the set of integers $\{1, \ldots d\}$.

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 $\mathcal{P} = \{P_1, \ldots, P_n\}$, and sometimes abuse notation identifying subsets $\mathcal{I} \subseteq \{1, \ldots, n\}$ 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. [17] 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. 4.

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

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

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 $(P_R, (\mathbf{x}_1, \dots, \mathbf{x}_\ell))$ from P_R, where $\mathbf{x}_h \in \mathbb{F}_2^\kappa$, it does the following:

- It samples $\mathbf{t}_h \in \mathbb{F}_2^\kappa$, $h = 1, \dots, \ell$, for P_R. If P_R is corrupted then it waits for $\mathcal{S}$ to input $\mathbf{t}_h$.
- It computes $\mathbf{q}_h = \mathbf{t}_h + \mathbf{x}_h * \Delta$, $h = 1, \dots, \ell$, 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 $\mathcal{F}^{\kappa,\ell}_{\mathsf{COTe}}$

OT functionality and a passively secure correlated OT functionality – 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 $\mathcal{F}^{\kappa,\ell}_{\mathsf{OT}}$ to denote the functionality that provides ℓ $\binom{2}{1}$ OTs in $\mathbb{F}_2^\kappa$. Note that $\mathcal{F}^{\kappa,\ell}_{\mathsf{OT}}$ can be implemented very efficiently for any $\ell = \mathsf{poly}(\kappa)$ using just one call to $\mathcal{F}^{\kappa,\kappa}_{\mathsf{OT}}$ and symmetric primitives, for example with actively secure OT extensions [19,1,14].

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. [13] 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 the full version. The security was proven e.g. by Nielsen [18], 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 [19] 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 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 $\mathcal{F}_{\mathsf{ACOT}}^{k,s}$ (see the full version), the amplification manifests itself in the fact that the environment does not learn $\mathbf{x}'$ which amplifies the error Y'.

The protocol $\mathit{\Pi}_{\mathsf{ACOT}}^{k,s}$ (Fig. 3) 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 $\mathit{\Pi}_{\mathsf{ACOT}}^{k,s}$ (Fig. 3) implements the functionality $\mathcal{F}_{\mathsf{ACOT}}^{k,s}$ (see the full version) in the $\mathcal{F}_{\mathsf{OT}}^{k,\ell'}$-hybrid model with statistical security s.*

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

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 $\mathcal{F}^{k,\ell'}_{\mathsf{OT}}$:
   - (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 $\mathcal{F}_{\mathsf{Rand}}$ (see full version).
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. 3. Amplified correlated OT

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 the desired entropy in the output. The two cases are modeled by two different possible adversarial inputs to the functionality. See the full version 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. [17] (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] (see full version), this turns out to be sufficient for our purposes, avoiding the need for additional consistency checks in the OTs.

Our authentication protocol $\Pi_{[\![\cdot]\!]}$ (see the full version) begins with an $\mathsf{Initialize}$ 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 $\mathcal{F}_{\mathsf{COTe}}.\mathsf{Initialize}$ is only called once, the MAC key shares are fixed for the entire protocol, so it is clear that

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 $\{\varDelta^{(j)}\}_{j\in A}$ in $\mathbb{F}$. It samples random $\{\varDelta^{(i)}\}_{i\notin A}$ in $\mathbb{F}$ for the honest parties, defining $\varDelta := \sum_{i\in[n]} \varDelta^{(i)}$. 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 $\mathbf{m}_h = \mathbf{x}_h \cdot \varDelta$. [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 $\mathbf{m}_h^{(k)} = \mathbf{m}_h^{(k)} + \sum_{j=1}^{M} e_{h,j}^{(k)} \cdot \varDelta_j^{(k)} \cdot X^{j-1}$ where $\varDelta_j^{(k)}$ denotes the j-th bit of $\varDelta^{(k)}$.
- 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 $\mathsf{Success}$ if $(\varDelta^{(1)}, \ldots, \varDelta^{(n)}) \in S$. Otherwise return Abort.

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

Fig. 4. Ideal Generation of $[\![\cdot]\!]$-representations

P_j could not remain undetected if enough random values are authenticated and checked. To ensure this in our protocol we add a consistency check to the $\mathsf{Initialize}$ 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 $\varPi_{[\![\cdot]\!]}$, realizing the ideal functionality of Fig. 4, more in 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 $\varDelta$ is initialized, the parties call the subprotocol $\Pi_{[\cdot]}$ (see the full version) 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 $\sum_{k=1}^{u} [\![x_k]\!] \cdot X^{k-1}$ (see the full version for details). 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 $\Pi_{[\![\cdot]\!]}$ implements $\mathcal{F}_{[\![\cdot]\!]}$ against any static adversary corrupting up to $n-1$ parties.*

Proof. See the full version.

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 $(\mathsf{Triples})$, 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 $(\mathsf{ValueError}, e)$ and $(\mathsf{MacError}, \mathbf{e})$.
- 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 $\mathsf{Success}$ if $(\Delta^{(1)}, \ldots, \Delta^{(n)}) \in S$. Otherwise return Abort.

Fig. 5. Ideal functionality for triples generation

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. 7 and 8), and then the protocol for bit triples (Fig. 9). Both approaches implement the functionality $\mathcal{F}^{\mathbb{F}}_{\mathsf{Triples}}$, given in Fig. 5. 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 two functionalities $\mathcal{F}^{k,s}_{\mathsf{GFMult}}$ and $\mathcal{F}^{\mathbb{F}_{2^k}}_{[\![\cdot]\!]}$ (see the full version). We realize the functionality $\mathcal{F}^{k,s}_{\mathsf{GFMult}}$ with protocol $\Pi^{k,s}_{\mathsf{GFMult}}$ (see the full version). 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 the values or the authentication key, but an active

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. 6. $\mathbb{F}_{2^k}$ 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. 7. Protocol for generation of unchecked $\mathbb{F}_{2^k}$ triples.

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.

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

Proof. The proof is straightforward using an appropriate simulator. See the full version 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

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 $\mathbf{t}, \mathbf{t}', \mathbf{t}'' \overset{\$}{\leftarrow} \mathbb{F}_{2^k}$ 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. 8. Triple checking protocol.

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 $\mathcal{F}^{k,s}_{\mathsf{GFMult}}$ and $\mathcal{F}^{\mathbb{F}_{2^k}}_{[\![\cdot]\!]}$ 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 $\mathcal{F}^{k,s}_{\mathsf{GFMult}}$. On the other hand, $\mathcal{F}^{\mathbb{F}_{2^k}}_{[\![\cdot]\!]}$ provides less security guarantees. However, we will also prove that the more deviation there is with $\mathcal{F}^{\mathbb{F}_{2^k}}_{[\![\cdot]\!]}$, 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 $\mathcal{F}_{\mathsf{BatchCheck}}$, which checks that a batch of shared values are equal to zero, and can be easily implemented using commitment and $\mathcal{F}_{\mathsf{Rand}}$ (see the full version for details). The first use of $\mathcal{F}_{\mathsf{BatchCheck}}$ corresponds to using the SPDZ MAC check protocol for $\mathbf{r}_j$ and $\mathbf{s}_j$ 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. 8, 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 the full version.

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 $\Pi_{\mathsf{CheckTriples}}$ (see the full version).

Fig. 9. $\mathbb{F}_2$-triples generation

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. 9. The main difference with respect to the protocol by Larraia et al. [17] is that here we use the outputs of $\mathcal{F}_{\mathsf{COTe}}$ 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 $\varDelta$, giving $[\![x_h]\!] = (\langle x_h \rangle, \langle \mathbf{m}_h \rangle, \langle \varDelta \rangle)$. 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 [17]) and then sacrifice to check for correctness (as in the previous protocol).

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

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 $\hat{\mathbf{q}}_h^{(j,i)} = \hat{\mathbf{t}}_h^{(i,j)} + x_h^{(i)} \cdot \hat{\varDelta}^{(j)}$, $h \in [\ell]$, 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 [17]) 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:

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

[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.

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 bucket-based cut-and-choose technique by Larraia et al. [17]. In the full version we generalize and optimize the parameters for this method.

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. 5) 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 the full version.

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 the full version. 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. 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 $\varDelta \in \mathbb{F}_{2^u}^m$. 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 $[\![C(\mathbf{x})]\!] = \left\{\langle C(\mathbf{x})\rangle, \{\langle \mathbf{m}_{\mathbf{x}_i}\rangle\}_{i\in[m]}, \langle \varDelta\rangle\right\}$ 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 $[\![C(\mathbf{x})]\!] = \{\langle C(\mathbf{x})\rangle, \langle \mathbf{m}_{\mathbf{x}}\rangle, \langle \varDelta\rangle\}$ to denote a MiniMAC element, where $\mathbf{m}_{\mathbf{x}}$ is made up of one component of each of the m BigMACs. The steps are described in detail in the full version.

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.

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 (see the full version). Then a random pairwise sacrificing is done to ensure that it was in fact shares of multiplication being authenticated. This is done using protocol $\Pi_{\mathsf{MiniTriples}}$ (see the full version). 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} and Π_{Reorg} (see the full version for more details) 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 the full version 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 [19] 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. [17] 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 the full version 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. T. K. Frederiksen, M. Keller, E. Orsini, and P. Scholl. A unified approach to MPC with preprocessing using OT. Cryptology ePrint Archive (to appear), 2015. http://eprint.iacr.org/.
13. 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.
14. 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.
15. 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.
16. 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.
17. 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.
18. J. B. Nielsen. Extending oblivious transfers efficiently - how to get robustness almost for free. *IACR Cryptology ePrint Archive*, 2007:215, 2007.

19. 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.