SPD$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for Dishonest Majority

Ronald Cramer[1], Ivan Damgård[2], Daniel Escudero[2], Peter Scholl[2], and Chaoping Xing[3]

[1] CWI, Amsterdam & Leiden University
[2] Aarhus University
[3] Nanyang Technological University, Singapore

Abstract. Most multi-party computation protocols allow secure computation of arithmetic circuits over a finite field, such as the integers modulo a prime. In the more natural setting of integer computations modulo 2^k, which are useful for simplifying implementations and applications, no solutions with active security are known unless the majority of the participants are honest.

We present a new scheme for information-theoretic MACs that are homomorphic modulo 2^k, and are as efficient as the well-known standard solutions that are homomorphic over fields. We apply this to construct an MPC protocol for dishonest majority in the preprocessing model that has efficiency comparable to the well-known SPDZ protocol (Damgård et al., CRYPTO 2012), with operations modulo 2^k instead of over a field. We also construct a matching preprocessing protocol based on oblivious transfer, which is in the style of the MASCOT protocol (Keller et al., CCS 2016) and almost as efficient.

1 Introduction

In the context of secure multi-party computation (MPC) there are n parties $P_1, \ldots, P_n$ who want to compute a function $f : \mathcal{R}^n \to \mathcal{R}^n$ securely on an input $(x_1, \ldots, x_n)$, where each party P_i holds x_i, without revealing the inputs to each other and only by exchanging messages between them. The main security guarantee we would like to achieve is that at the end of the interaction each party P_i only learns x_i and the i-th component of $f(x_1, \ldots, x_n)$, and nothing else. This should hold even if an adversary corrupts some of the parties and, in case of active or malicious corruption, takes control of the corrupted parties and have them do what the adversary wants. These ideas are formalized by requiring that using the protocol should be essentially equivalent to having a trusted third party compute the function. For such a formalization see, for example, the Universal Composability Framework (UC) [4].

It is well known that the hardest case to handle efficiently is the dishonest majority case, where $t \geq n/2$ parties are actively corrupted. Here we cannot guarantee that the protocol terminates correctly, and we have to use computationally heavy public-key technology — unconditional security is not possible in this scenario. However, in a recent line of work [2,9], it was observed that we can push the use of public-key tools into a preprocessing phase, where one does not need to know the inputs or even the function to be computed. This phase produces “raw material” (correlated randomness) that can be used later in an online phase to compute the function much more efficiently and with unconditional security (given the correlated randomness).

In all existing protocols that handle a dishonest majority and active corruptions, the function being computed must be expressed in terms of arithmetic operations (i.e. additions and multiplications) over a finite field, such as the integers modulo a prime. However, in many applications one would like to use numbers modulo some M that is chosen by the application and is not necessarily a prime. In particular, $M = 2^k$ is interesting because computation modulo 2^k matches closely what happens on standard CPUs and hence protocol designers can take advantage of the tricks found in this domain. For instance, functions containing comparisons and bitwise operations are typically easier to implement using arithmetic modulo 2^k; these kinds of operations are expensive to emulate with finite field arithmetic, and also very common in applications of MPC such as secure benchmarking based on linear programming [6]. This has been done successfully by the team behind the Sharemind suite of protocols [3], which allows bitwise operations and integer arithmetic mod 2^{32}. However, in their basic setting, they could only get a passively secure solution: here, even corrupt players are assumed to follow the protocol. Also, the security of Sharemind completely breaks down if half (or more) of the players are corrupted, and the efficiency does not scale well beyond three parties.

To obtain active security over fields, the main idea of modern protocols is to use unconditionally secure message authentication codes (MACs) to prevent players from lying about the data they are given in the preprocessing phase. A typical example is the SPDZ protocol [9,7], where security reduces to the following game: we have a data value x, a random MAC key α and a MAC $m = \alpha x$, all in some finite field $\mathbb{F}$. The adversary is given x but not α or αx. He may now specify errors to be added to x, α and m, and we let x', α', m' be the resulting values. The adversary wins if $x \neq x'$ and $m' = \alpha' x'$. It is easy to see that the adversary must guess α to win, and so the probability of winning is $1/|\mathbb{F}|$. This authentication scheme is additively homomorphic, which is exploited heavily in the SPDZ protocol and is crucial for its efficiency.

However, the security proof depends on the fact that any non-zero value in $\mathbb{F}$ is invertible, and it is easy to see that if we replace the field by a ring, say $\mathbb{Z}_{2^k}$, then the adversary can cheat with large probability. For instance, in the ring $\mathbb{Z}_{2^k}$ he can choose $x' = x + 2^{k-1}$ and cheat with probability $1/2$. Up to now, it has been an open problem to design a homomorphic authentication scheme that would work over $\mathbb{Z}_{2^k}$ or more generally $\mathbb{Z}_M$ for any M, and is as efficient as the SPDZ scheme.

1.1 Our contributions

In this paper we solve the above question: we design a new additively homomorphic authentication scheme that works in $\mathbb{Z}_{2^k}$[4], and is as efficient as the standard solution over a field. The main idea is to choose the MAC key α randomly in $\mathbb{Z}_{2^s}$, where s is the security parameter, and compute the MAC αx in $\mathbb{Z}_{2^{k+s}}$. We explain below why this helps. We also design a method for checking large batches of MACs with a communication complexity that does not depend on the size of the batch. We believe that these techniques will be of independent interest.

We then use the MAC scheme to design a SPDZ-style online protocol that securely computes an arithmetic circuit over $\mathbb{Z}_{2^k}$ with statistical security, assuming access to a preprocessing functionality that outputs multiplication triples in a suitable format. The total computational work done is dominated by $O(|C|n)$ elementary operations in the ring $\mathbb{Z}_{2^{k+s}}$, where C is the circuit to be computed. So if $k \geq s$, the work needed per player is equal to the work needed to compute C in the clear, up to a constant factor — as is the case for the SPDZ protocol. As in other protocols from this line of work, the overhead becomes more significant when k is small. Each player stores data from the preprocessing of size $O(|C|(k+s))$ bits. However, the communication complexity is $O(|C|k)$ bits plus an overhead that does not depend on C. This is due to the batch-checking of MACs mentioned above.

Our final result is an implementation of the preprocessing functionality to generate multiplication triples. It has communication complexity $O((k+s)^2)$ bits per multiplication gate, and is roughly as efficient as the MASCOT protocol [14], which is the state of the art for preprocessing over a field using oblivious transfer. Concretely, our triple generation protocol has around twice the communication cost of MASCOT, due to the overhead incurred when we have to work over larger rings in certain scenarios. However, this additional cost seems like a small price to pay for the potential benefits to applications from working modulo 2^k instead of in a field.

1.2 Overview of our techniques

For the authentication scheme, as mentioned, we have a data item $x \in \mathbb{Z}_{2^{k+s}}$, a key $\alpha \in \mathbb{Z}_{2^{k+s}}$ and we define the MAC as $m = \alpha x \mod 2^{k+s}$. Note that we want to authenticate k-bit values, so although $x \in \mathbb{Z}_{2^{k+s}}$, only the least significant k bits matter. The adversary is given x, and specifies errors e_x, e_α, e_m, which define modified values $x' = x + e_x, \alpha' = \alpha + e_\alpha, m' = m + e_m$. He wins if $m' = \alpha' x' \bmod 2^{k+s}$, but note that since we store data in the least significant k bits only, this is only a forgery if $e_x \bmod 2^k \neq 0$. As we show in detail in Section 3, if the adversary wins, he is able to compute $e_x \alpha \bmod 2^{k+s}$. From this, and $e_x \bmod 2^k \neq 0$, it follows that the adversary can effectively guess $\alpha \bmod 2^s$, which is only possible with probability 2^{-s}.

We also want to batch-check many MACs using only a small amount of communication. The SPDZ protocol [9] uses a method that basically takes a random

[4] We use modulus 2^k throughout, but the scheme easily extends to any modulus.

linear combination of all messages and MACs and checks only the resulting message and MAC. Unfortunately, applying the analysis we just sketched to this scenario does not give a negligible probability of cheating, unless we 'lift' again and compute MACs modulo 2^{k+2s}, but then our storage and preprocessing costs would become significantly bigger. We provide a more complicated but tighter analysis showing that we can still compute MACs mod 2^{k+s} and the batch checking works with $2^{-s+\log s}$ error probability, so we only need increase s by a few bits.

Using these MACs, we can create an information-theoretically secure MPC protocol over $\mathbb{Z}_{2^k}$ in the preprocessing model, similar to the online phase of SPDZ from [7]. To implement the preprocessing phase, we follow the style of MASCOT [14], which uses oblivious transfer to produce shares of authenticated multiplication triples. We first design a protocol for authenticating values using correlated oblivious transfer, which allows creating the secret-shared MACs that will be added to the preprocessing data. This stage is similar to MASCOT, whereby first a passively secure protocol is used to compute shares of the MACs αx_i, for each value x_i that is to be authenticated, and then a random linear combination of these values is opened, and the resulting MAC checked for correctness. The main change we need to make here is that, depending on the size of the x_i's being authenticated, we may need to first compute the MACs over a larger ring in order to apply our analysis of taking random linear combinations.

Once the authentication scheme has been implemented, the main task is to create the multiplication triples needed in the online phase of our protocol. For this we also follow a similar approach to MASCOT, where the overall idea is that each party P_i chooses its shares (a^i, b^i) and then is engaged in an oblivious transfer subprotocol with P_j for each $j \neq i$, where shares of the cross products $a^i b^j$ and $a^j b^i$ are obtained. This yields shares of the product $(\sum_{i=1}^n a^i)(\sum_{j=1}^n b^j) = \sum_{i=1}^n a^i b^i + \sum_{i \neq j}(a^i b^j + a^j b^i)$, as required. Behind this simplification lies the problem that some information about the honest parties' shares can be leaked to a cheating adversary. In MASCOT this potential leakage is mitigated by "spreading out" the randomness by taking random linear combinations on correlated triples (with the same b value). When working over fields, the inner product yields a 2-universal hash function so the new distribution can be argued to be close to uniform using the Leftover Hash Lemma. However, this is not true anymore over rings like $\mathbb{Z}_{2^k}$. We overcome this issue by starting with triples where the shares of a are *bits* instead of ring elements, and then taking linear combinations over the bits. These combinations correspond to a subset sum over $\mathbb{Z}_{2^k}$, which *is* a 2-universal hash function, so allows for removing the leakage.

Additionally, random combinations are used in MASCOT to check the correctness of a triple by "sacrificing" another one. The security argument is that if the adversary manages to authenticate an incorrect triple, then it will have to guess the randomness used in the sacrifice step, which is unlikely. This is argued by deriving an equation from which we can solve for the random value. In order to extend this argument to the ring case, we use the technique sketched at the beginning of this section, working over $\mathbb{Z}_{2^{k+s}}$ to check correctness modulo 2^k.

Organization of this document. Section 2 introduces the notation we will use throughout this document. It also introduces the oblivious transfer and coin tossing functionalities, $\mathcal{F}_{\mathsf{ROT}}$ and $\mathcal{F}_{\mathsf{Rand}}$, which constitute our most basic building blocks and will be used to implement the offline phase of our protocol. We then describe our information-theoretic MAC scheme in Section 3, and we show how to check correctness of several authenticated values assuming a functionality $\mathcal{F}_{\mathsf{MAC}}$ that generates keys and MACs. Next, in Section 4 we show how to use our scheme to realise the functionality $\mathcal{F}_{\mathsf{Online}}$, i.e. to evaluate securely any arithmetic circuit modulo 2^k, in the preprocessing model.

The next two sections are concerned with the implementation of the preprocessing functionality $\mathcal{F}_{\mathsf{Prep}}$. Section 5 deals with the implementation of the functionality $\mathcal{F}_{\mathsf{MAC}}$, i.e. the distribution of the MAC key and the generation of MACs. Our construction is based on a primitive called vector Oblivious Linear Function Evaluation ($\mathcal{F}_{\mathsf{vOLE}}$). This can be implemented using Correlated Oblivious Transfer ($\mathcal{F}_{\mathsf{\Delta\text{-}OT}}$), which as we mention in that section can be implemented using our basic primitive $\mathcal{F}_{\mathsf{ROT}}$. On the other hand, Section 6 builds on top of our MAC scheme and generates multiplication triples that will be used during the online phase of our protocol to evaluate multiplication gates. Finally, in Section 7 we provide an efficiency analysis of our protocol.

Related work. There are only a few previous works that study MPC over rings, and none of these offer security against an active adversary who corrupts a dishonest majority of the parties. Cramer et al. showed how to contruct actively secure MPC over black-box rings [5] using secret-sharing techniques for honest majority, but this is only a feasibility result and the concrete efficiency is not clear. As already mentioned, Sharemind [3] allows mixing of secure computation over the integers modulo 2^k with boolean computations, but is restricted to the three-party setting when at most one party is corrupted. In some settings Sharemind can also provide active security [18].

More recently, Damgård, Orlandi and Simkin [8] present a compiler that transforms a semi-honest secure protocol for t corruptions into a maliciously secure protocol that is secure against a smaller number of corruptions (approximately $\sqrt{t}$). This also works for protocols in the preprocessing model, but will always result in a protocol for honest majority, so they can tolerate a smaller number of corruptions. On the other hand, their compiler is perfectly secure, so it introduces no overhead that depends on the security parameter. Thus, their results are incomparable to ours.

2 Preliminaries

Notation We denote by $\mathbb{Z}_M$ the set of integers x such that $0 \leq x \leq M - 1$. The congruence $x \equiv y \mod 2^k$ will be abbreviated as $x \equiv_k y$. We let $x \bmod M$

denote the remainder of x when divided by M, and we take this representative as an element of the set $\mathbb{Z}_M$. Given two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ of the same dimensions, $\boldsymbol{x} * \boldsymbol{y}$ denotes their component-wise product, $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ denotes their dot product and $\boldsymbol{x}[i]$ denotes the i-th entry of $\boldsymbol{x}$.

2.1 Oblivious Transfer and Coin Tossing Functionalities

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

On input $(\mathsf{Sender}, P_j, \ell)$ from P_j and $(\mathsf{Receiver}, b, P_i)$ from P_i, the functionality samples random values $r_0, r_1 \leftarrow_R \mathbb{Z}_{2^\ell}$, then sends (r_0, r_1) to P_j and r_b to P_i.
If P_j is corrupted then the functionality instead allows the adversary to choose (r_0, r_1) before sending r_b to P_i.

Fig. 1. Random Oblivious Transfer functionality between a sender and receiver

We use a standard functionality for oblivious transfer on random ℓ-bit strings, shown in Fig. 1. This can be efficiently realised using OT extension techniques with an amortized cost of κ bits per random OT, where κ is a computational security parameter [13]. We use the notation $\mathcal{F}^{\tau}_{\mathsf{ROT}}$ to denote τ parallel copies of $\mathcal{F}_{\mathsf{ROT}}$ functionalities.

We also use a coin tossing functionality, which on input (Rand) from all parties, sample $r \leftarrow_R \mathcal{R}$ and output r to all parties. This can be implemented in the random oracle model by having each party P_i first commit to a random seed s_i with $H(i \| s_i)$, then opening all commitments and using $\bigoplus_i s_i$ as a seed to sample from $\mathcal{R}$.

3 Information-Theoretic MAC Scheme

In this section we introduce our secret-shared, information-theoretic message authentication scheme. This forms the backbone of our MPC protocol over $\mathbb{Z}_{2^k}$. The scheme has two parameters, k, where 2^k is the size of the ring in which computations are performed, and a security parameter s. In the MAC scheme itself and the online phase of our MPC protocol there is no restriction on k, whilst in the preprocessing phase k also affects security.

There is a single, global key $\alpha = \sum_i \alpha^i \bmod 2^{k+s}$, where each party holds a random additive share $\alpha^i \in \mathbb{Z}_{2^s}$. For every authenticated, secret value $x \in \mathbb{Z}_{2^k}$, the parties will have additive shares on this value over the *larger ring* modulo 2^{k+s}, namely shares $x_i \in \mathbb{Z}_{2^{k+s}}$ such that $x' = \sum_i x^i \bmod 2^{k+s}$ and $x \equiv_k x'$. The parties will also have additive shares modulo 2^{k+s} of the MAC $m = \alpha \cdot x' \bmod 2^{k+s}$. We will denote this representation by $[x]$, so we have:

$$[x] = \left(x^i, m^i, \alpha^i\right)_{i=1}^{n} \in \left(\mathbb{Z}_{2^{k+s}} \times \mathbb{Z}_{2^{k+s}} \times \mathbb{Z}_{2^s}\right)^n, \quad \sum_i m^i \equiv_{k+s} \big(\sum_i x^i\big) \cdot \big(\sum_i \alpha^i\big)$$

Notice that if the parties have $[x]$ and $[y]$, then it is straightforward to obtain by means of local operations $[x + y]$, $[c \cdot x]$ and $[x + c]$, where the arithmetic is modulo 2^{k+s} and c is a constant. We state the procedures that allow the parties to do this in Fig. 2.

Procedure AffineComb

This procedure allows the parties to compute authenticated shares of $y = c + c_1 \cdot x_1 + \cdots + c_t \cdot x_t \mod 2^{k+s}$ given $c, c_1, \ldots, c_t, [x_1], \ldots, [x_t]$. The input to this procedure are the constants $c, c_1, \ldots, c_t \in \mathbb{Z}_{2^{k+s}}$, the shares of the values $\{x_i^j\}_{i=1}^t$, the shares of the MACs $\{m_i^j\}_{i=1}^t$, owned by each party P_j, and the shares of the MAC key $\{\alpha^j\}_j$.

1. Party P_1 sets $y^1 = c + c_1 \cdot x_1^1 + \cdots + c_t \cdot x_t^1 \mod 2^{k+s}$;
2. Each party P_j, $j \neq 1$, sets $y^j = c_1 \cdot x_1^j + \cdots + c_t \cdot x_t^j \mod 2^{k+s}$;
3. Each party P_j sets $m^j = \alpha_j \cdot c + c_1 \cdot m_1^j + \cdots + c_t \cdot m_t^j \mod 2^{k+s}$.

At the end of the procedure $\{y^j\}_j$ are additive shares of y modulo 2^{k+s} and $\{m^j\}_j$ are shares of $\alpha \cdot y$ mod 2^{k+s}, the MAC of y. To simplify the exposition, we write

$$[c + c_1 \cdot x_1 + \cdots + c_t \cdot x_t] = c + c_1 \cdot [x_1] + \cdots + c_t \cdot [x_t]$$

whenever this procedure is called.

Fig. 2. Procedure for obtaining authenticated shares of affine combinations of shared values

In Fig. 3 we define the functionality $\mathcal{F}_{\mathsf{MAC}}$, which acts as a trusted dealer who samples and distributes shares of the MAC key, and creates secret-shared MACs of additively shared values input by the parties. As with previous works, it allows corrupt parties to choose their own shares instead of sampling them at random, since our protocols allow the adversary to influence the distribution of these. We will show how to implement this functionality in Section 5.

3.1 Opening Values and Checking MACs

Given an authenticated sharing $[x]$, a natural (but insufficient) approach to opening and reconstructing x is for each party to first broadcast the share x^i and then compute $x' = \sum_i x^i \mod 2^{k+s}$. The parties can then check the MAC relation $x' \cdot \alpha$ without revealing the key α using the method from [7]. Although this method guarantees *integrity* of the opened result modulo 2^k (by the same argument sketched in the introduction), it does not suffice for *privacy* when accounting for the fact that x may be a result of applying linear combinations on other private inputs. For example, suppose $x = y + z$ for some previous inputs y, z. When opening x modulo 2^{k+s}, although for correctness we only care about the lower k bits of x, to verify the MAC relation we have to reveal the *entire* shares modulo 2^{k+s}. This leaks whether or not the sum $y + z$ overflowed modulo 2^k.

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

The functionality generates shares of a global MAC key and, on input shares of a value, distributes shares of a tag of this value. Let A be the set of corrupted parties and s be a security parameter.

Initialize: On receiving (Init) from all parties, sample random values $\alpha^j \leftarrow_R \mathbb{Z}_{2^s}$ for $j \notin A$ and receive shares $\alpha^j \in \mathbb{Z}_{2^s}$, for $j \in A$, from the adversary. Store the MAC key $\alpha = \sum_{j=1}^{n} \alpha^j$ (over $\mathbb{Z}$) and output α^j to party P_j.

Macro $\mathsf{Auth}(\ell, x^1, \ldots, x^n)$ (this is an internal subroutine only)

1. Let $x = \sum_{j=1}^{n} x^j \mod 2^\ell$ and $m = \alpha \cdot x \mod 2^\ell$
2. Wait for input $\{m^j\}_{j \in A}$ from the adversary and sample $\{m^j\}_{j \notin A}$ at random conditioned on $m \equiv_\ell \sum_{j=1}^{n} m^j$. Output $(m^1, \ldots, m^n)$.

Authentication: On input $(\mathsf{MAC}, \ell, r, \{x_i^j\}_{i=1}^{t})$ from each party P_j, where $x_i^j \in \mathbb{Z}_{2^r}$ and $\ell \geq r$:

1. Wait for the adversary to send messages (guess, j, S_j), for every $j \notin A$, where S_j efficiently describes a subset of $\{0,1\}^s$. If $\alpha^j \in S_j$ for all j then send ($\mathsf{success}$) to $\mathcal{A}$. Otherwise, send $\perp$ to all parties and abort.
2. Execute $\mathsf{Auth}(\ell, x_i^1, \ldots, x_i^n)$ for $i = 1, \ldots, t$, and then wait for the adversary to send either OK or Abort. If the adversary sends OK then send the MAC shares $m_i^j \in \mathbb{Z}_{2^\ell}$ to party P_j, otherwise abort.

Fig. 3. Functionality for generating shares of global MAC key, distributing shares of inputs and tags

To prevent this leakage we use an authenticated, random s-bit mask to hide the upper s bits of x when opening. The complete protocol for doing this is shown below.

Procedure $\mathsf{SingleCheck}([x])$**:**

1. Generate a random, shared value $[r]$ using $\mathcal{F}_{\mathsf{MAC}}$, where $r \in \mathbb{Z}_{2^s}$
2. Compute $[y] = [x + 2^k r]$
3. Each party broadcasts their shares y^i and reconstructs $y = \sum_i y^i \mod 2^{k+s}$
4. P_i commits to $z^i = m^i - y \cdot \alpha^i \mod 2^{k+s}$, where m^i is the MAC share on y
5. All parties open their commitments and check that $\sum_i z^i \equiv_{k+s} 0$
6. If the check passes then output $y \bmod 2^k$

Claim 1 *If the MAC check passes then $y \equiv_k x$, except with probability at most 2^{-s}.*

Proof. Suppose a corrupted party opens $[y]$ to some $y' = y + \delta$, where $\delta \in \mathbb{Z}_{2^{k+s}}$ can be chosen by $\mathcal{A}$, and $\delta \not\equiv_k 0$. To pass the MAC check, they must also come up with an additive error Δ in the committed values z^i such that $\sum_i z^i + \Delta$ is zero modulo 2^{k+s}. This simplifies to finding $\Delta \in \mathbb{Z}_{2^{k+s}}$ such that

$$\sum_i (m^i - (x + \delta) \cdot \alpha^i) + \Delta \equiv_{k+s} 0$$

$$\Leftrightarrow \delta \cdot \alpha \equiv_{k+s} -\Delta$$

Let v be the largest integer such that 2^v divides δ, and note that because $\delta \not\equiv_k 0$ we have $v < k$. This means that we can divide the above by 2^v, reducing the modulus from 2^{k+s} to 2^{k+s-v} accordingly:

$$\frac{\delta}{2^v} \cdot \alpha \equiv_{k+s-v} -\frac{\Delta}{2^v}$$

By definition of v, $\frac{\delta}{2^v}$ must be an odd integer, hence invertible modulo 2^{k+s-v}. Multiply by its inverse gives

$$\alpha \equiv_{k+s-v} -\frac{\Delta}{2^v} \cdot \left(\frac{\delta}{2^v}\right)^{-1}$$

Note that $k+s-v > s$, since $v < k$, which implies that $\mathcal{A}$ must have guessed $\alpha \bmod 2^s$ to come up with δ and Δ which pass the check. This requires guessing the s least significant bits of α, which are uniformly random, so the probability of success is at most 2^{-s}. □

3.2 Batch MAC Checking with Random Linear Combinations

Procedure BatchCheck

Procedure for opening and checking the MACs on t shared values $[x_1], \ldots, [x_t]$. Let x_i^j, m_i^j, α^j be P_j's share, MAC share and MAC key share for $[x_i]$.

Open phase:

1. Each party P_j broadcasts for each i the value $\tilde{x}_i^j = x_i^j \bmod 2^k$.
2. The parties compute $\tilde{x}_i = \sum_{j=1}^n \tilde{x}_i^j \mod 2^{k+s}$.

MAC check phase:

3. The parties call $\mathcal{F}_{\mathsf{Rand}}(\mathbb{Z}_{2^s}^t)$ to sample public random values $\chi_1, \ldots, \chi_t \in \mathbb{Z}_{2^s}$ and then compute $\tilde{y} = \sum_{i=1}^t \chi_i \cdot \tilde{x}_i \mod 2^{k+s}$.
4. Each party P_j samples $r^j \leftarrow_R \mathbb{Z}_{2^s}$, and then calls $\mathcal{F}_{\mathsf{MAC}}$ on input $(s, s, r^j, \mathsf{MAC})$ to obtain $[r]$. Denote P_j's MAC share on r by ℓ^j.
5. Each party P_j computes $p^j = \sum_{i=1}^t \chi_i \cdot p_i^j \mod 2^s$ where $p_i^j = \frac{x_i^j - \tilde{x}_i^j}{2^k}$ and broadcasts $\tilde{p}^j = p^j + r^j \mod 2^s$.
6. Parties compute $\tilde{p} = \sum_{j=1}^n \tilde{p}^j \mod 2^s$.
7. Each party P_j computes $m^j = \sum_{i=1}^t \chi_i \cdot m_i^j \mod 2^{k+s}$ and $z^j = m^j - \alpha^j \cdot \tilde{y} - 2^k \cdot \tilde{p} \cdot \alpha^j + 2^k \cdot \ell^j \mod 2^{k+s}$. Then it commits to z^j, and then all parties open their commitments.
8. Finally, the parties verify that $\sum_{j=1}^n z^j \equiv_{k+s} 0$. If the check passes then the parties accept the values $\tilde{x}_i \mod 2^k$, otherwise they abort.

Fig. 4. Procedure for checking a batch of MACs

The method described in the previous section allows the parties to open and then check one shared value $[x]$. However, in our MPC protocol many such values will be opened, and using the previous method to check each one of these would have the drawback that we need shared, authenticated random masks for each value to be opened, consuming a lot of additional preprocessing data.[5] In order to avoid this, we present a batch MAC checking procedure for opening and checking t shared values $[x_1], \ldots, [x_t]$, which uses just *one random mask* to check the whole batch.

Technically speaking, our main contribution here is a new analysis of the distribution of random linear combinations of adversarially chosen errors modulo 2^k, when lifting these combinations to the larger ring $\mathbb{Z}_{2^{k+s}}$. If we naively apply the analysis from Claim 1 to this case, then we would have to lift to an even bigger ring $\mathbb{Z}_{2^{k+2s}}$ to prove security, adding extra overhead when creating and storing the MACs. With our more careful analysis in Lemma 1 below, we can still work over $\mathbb{Z}_{2^{k+s}}$ and obtain failure probability around $2^{-s+\log s}$, which gives a significant saving.

Suppose the parties wish to open $[x_1], \ldots, [x_t]$, hence learn the values $x_1, \ldots, x_t$ modulo 2^k. Denote the shares, MAC shares and MAC key share held by P_j as x_i^j, m_i^j, α^j respectively. To initially open the values, the parties simply broadcast their shares $\tilde{x}_i^j = x_i^j \bmod 2^k$ and reconstruct $\tilde{x}_i = \sum_j \tilde{x}_i^j$ (as before, we cannot send the upper s bits of x_i^j for privacy reasons). As the parties do not have MACs on the values modulo 2^k, these s dropped bits will have to be used at some point during the MAC check, by adding them back in to the linear combination of MACs being checked. Crucially, by postponing the use of these s bits until the MAC check phase, our protocol only needs one authenticated random value to mask them, instead of t. The procedure that achieves this is described in Fig. 4, and its guarantees are stated in the following theorem.

Theorem 1. *Suppose that the inputs $[x_1], \ldots, [x_t]$ to the* BatchCheck *procedure are consistent sharings of $x_1, \ldots, x_t$ under the MAC key $\alpha = \sum_i \alpha^i \mod 2^s$, and the honest parties' shares $\alpha^j \in \mathbb{Z}_{2^s}$ are uniformly random in the view of an adversary corrupting at most $n-1$ parties. Then, if the procedure does not abort, the values $\tilde{x}_i$ accepted by the parties satisfy $x_i \equiv_k \tilde{x}_i$ with probability at least $1 - 2^{-s+\log(s+1)}$.*

The following lemma will be used in the proof of this theorem. The lemma is very general, which will allow us to use it also when we prove the security of the preprocessing phase of our protocol. However, in the current context, this lemma will be used with $\ell = k+s$, $r = k$ and $m = s$, and the δ's can be thought of as the errors introduced by the adversary during the opening phases.

Lemma 1. *Let ℓ, r and m be positive integers such that $\ell - r \leq m$. Let $\delta_0, \delta_1, \ldots, \delta_t \in \mathbb{Z}$, and suppose that not all the δ_i's are zero modulo 2^r, for $i > 0$. Let Y be a probability distribution on $\mathbb{Z}$. Then, if the distribution Y is independent from the*

[5] Note that in previous SPDZ-like protocols these extra masks are not needed.

uniform distribution sampling α below, we have

$$\Pr_{\substack{\alpha,\chi_1,\ldots,\chi_t \leftarrow_R \mathbb{Z}_{2^m},\\ y \leftarrow_R Y}} \left[\alpha \cdot \left(\delta_0 + \sum_{i=1}^{t} \chi_i \cdot \delta_i \right) \equiv_\ell y \right] \leq 2^{-\ell+r+\log(\ell-r+1)},$$

Proof. Define $S := \delta_0 + \sum_{i=1}^{t} \chi_i \cdot \delta_i$, and define E to be the event that $\alpha \cdot S \equiv_\ell y$. Let W be the random variable defined as $\min(\ell, e)$, where 2^e is the largest power of two dividing S. We will use the following claims.

Proposition 1.

i. $\Pr[E \mid W = r + c] \leq 2^{-(\ell-r-c)}$ *for any* $c \in \{1, \ldots, \ell - r\}$
ii. $\Pr[E \mid 0 \leq W \leq r] \leq 2^{-(\ell-r)}$
iii. $\Pr[W = r + c] \leq 2^{-c}$ *for any* $c \in \{1, \ldots, \ell - r\}$

Proof. For the first part, suppose that $0 < c < \ell - r$ (the case $c = \ell - r$ is trivial), in particular, $w = r + c$ is the largest exponent such that 2^w divides S and therefore $S/2^w$ is an odd integer. From the definitions of E and w we have that E holds if and only if $\alpha \cdot S \equiv_\ell y$, which in turn is equivalent to $\alpha \cdot \frac{S}{2^w} \equiv_{\ell-w} \frac{y}{2^w}$ and therefore to $\alpha \equiv_{\ell-w} \frac{y}{2^w} \cdot \left(\frac{S}{2^w}\right)^{-1}$ Since α is uniformly random in $\mathbb{Z}_{2^m}$ and independent of the right-hand side, and also $\ell - w < m$ (as $r < w$ and $\ell - r \leq m$), we conclude that the event holds with probability $2^{-(\ell-w)} = 2^{-(\ell-r-c)}$, conditioned on $W = r + c$.

Similarly, if $0 \leq w \leq r$ then $\ell - w \geq \ell - r$ and so $\alpha \equiv_{\ell-r} \frac{y}{2^w} \cdot \left(\frac{S}{2^w}\right)^{-1}$. As $\ell - r \leq m$, the event holds with probability at most $2^{-(\ell-r)}$ if conditioned on $0 \leq W \leq r$. This proves the second part.

For the third part, we must also look at the randomness from the χ_i coefficients. Suppose without loss of generality that δ_t is non-zero modulo 2^r, and suppose that $W = r + c$ some $1 \leq c \leq \ell - r$. Since $2^W | S$, we have $S \equiv_{r+c} 0$, and so

$$\chi_t \cdot \delta_t \equiv_{r+c} \underbrace{-\delta_0 - \sum_{i \neq t} \chi_i \cdot \delta_i}_{=S'}$$

Let 2^v be the largest power of two dividing δ_t, and note that by assumption we have $v < r$ so $r + c - v > c$. Therefore,

$$\begin{aligned} \chi_t \cdot \frac{\delta_t}{2^v} &\equiv_{r+c-v} \frac{S'}{2^v} \\ \chi_t &\equiv_{r+c-v} \frac{S'}{2^v} \left(\frac{\delta_t}{2^v}\right)^{-1} \\ \chi_t &\equiv_c \frac{S'}{2^v} \left(\frac{\delta_t}{2^v}\right)^{-1} \end{aligned}$$

By the same argument as previously, and from the fact that $c \leq \ell - r \leq m$, this holds with probability 2^{-c}, over the randomness of $\chi_t \leftarrow_R \mathbb{Z}_{2^m}$, as required.

Putting things together, we apply the law of total probability over all possible values of w, obtaining:

$$\begin{aligned}\Pr[E] = \Pr[E \mid 0 \leq W \leq r]{\cdot}\Pr[0 \leq W \leq r] {+} \sum_{c=1}^{\ell-r} \Pr[E \mid W = r{+}c]{\cdot}\Pr[W = r{+}c]\\ \leq 2^{-\ell+r} \cdot 1 + \sum_{c=1}^{\ell-r} 2^{-\ell+r+c} \cdot 2^{-c} = 2^{-\ell+r} + \sum_{c=1}^{\ell-r} 2^{-\ell+r}\\ = (\ell - r + 1) \cdot 2^{-\ell+r} \leq 2^{-\ell+r+\log(\ell-r+1)}\end{aligned}$$

where the first inequality comes from applying item ii. of Proposition 1 on the left, and items i. and iii. on the right. □

Now we proceed with the proof of Theorem 1.

Proof (of Theorem 1). We first assume that $\mathcal{A}$ sends no **Key Query** messages to $\mathcal{F}_{\mathsf{MAC}}$, and later discuss how the claim still holds when this is not the case.

First of all notice that if no error is introduced by the adversary, then the check passes. Now, let $y = \sum_{i=1}^{t} \chi_i \cdot x_i \mod 2^{s+k}$, $p_i = \sum_{j=1}^{n} p_i^j \mod 2^s$ and $p = \sum_{j=1}^{n} p^j \mod 2^s$. If all parties followed the protocol then the following chain of congruences holds

$$\begin{aligned}\sum_{j=1}^{n} z^j &\equiv_{k+s} \sum_{j=1}^{n} m^j - \tilde{y} \cdot \sum_{j=1}^{n} \alpha^j - 2^k \cdot \tilde{p} \cdot \sum_{j=1}^{n} \alpha^j + 2^k \cdot \sum_{j=1}^{n} \ell^j\\ &\equiv_{k+s} \alpha \cdot y - \alpha \cdot \tilde{y} - \alpha \cdot 2^k \cdot \tilde{p} + 2^k \cdot \alpha \cdot r\\ &\equiv_{k+s} \alpha \cdot (y - \tilde{y} - 2^k \cdot (\tilde{p} - r))\\ &\equiv_{k+s} \alpha \cdot (y - \tilde{y} - 2^k \cdot p)\\ &\equiv_{k+s} \alpha \cdot \sum_{i=1}^{t} \chi_i \cdot \left(x_i - \tilde{x}_i - 2^k p_i\right) \equiv_{k+s} 0\end{aligned}$$

where the last equality holds due to the fact that for all $i = 1, \ldots, t$ we have $x_i = \tilde{x}_i + 2^k \cdot p_i$.

Now, consider the case in which the adversary does not open correctly to $\tilde{x}_i$ and $\tilde{p}$ in the execution of the procedure. Let $\tilde{x}_i + \delta_i \mod 2^{k+s}$ and $\tilde{p} + \epsilon \mod 2^s$ be the values opened in steps 1 and 5 respectively, so the value computed in step 3 is equal to $\tilde{y}' = \tilde{y} + \delta \mod 2^{k+s}$, where $\delta = \sum_{i=1}^{t} \chi_i \cdot \delta_i \mod 2^{k+s}$. As a consequence, the share that an honest P_j should open in step 7 is $z^j - \alpha^j \cdot (\delta + 2^k \epsilon) \mod 2^{k+s}$. However, the adversary can open this value plus some errors that sum up to a value $\varDelta \in \mathbb{Z}_{2^{k+s}}$. If the check passes, this means that

$$0 \equiv_{k+s} \sum_{j=1}^{n} \left(z^j - \alpha^j \cdot (\delta + 2^k \epsilon)\right) + \varDelta \quad \Leftrightarrow \quad \alpha \cdot (\delta + 2^k \epsilon) \equiv_{k+s} \varDelta.$$

Suppose that for some index it holds that $\delta_i \not\equiv_k 0$. By setting $\delta_0 = 2^k \epsilon$, $\ell = k + s$, $r = k$, $m = s$ and Y to be the distribution of Δ produced by the adversary, we observe we are in the same setting as the hypothesis of Lemma 1. This allows us to conclude that the probability that the check passes is bounded by $2^{-\ell+r+\log(\ell-r+1)} = 2^{-s+\log(s+1)}$.

Handling key queries. We now show that this probability is the same for an adversary who makes some successful queries to an honest party's α^j using the (guess) command of $\mathcal{F}_{\mathsf{MAC}}$. Let S be the set of possible keys guessed by $\mathcal{A}$ (if there is more than one query then we take S to be the intersection of all sets). The probability that all these queries are successful is no more than $|S|/2^s$, and conditioned on this event, the min-entropy of the honest party's key share is reduced to $\log|S| \leq s$. Therefore, instead of success probability $2^{-s+\log(s+1)}$ as above, the overall probability of $\mathcal{A}$ performing successful key queries *and* passing the check is bounded by

$$|S|/2^s \cdot 2^{-\log|S|+\log(\log|S|+1)} = 2^{-s+\log(\log|S|+1)} \leq 2^{-s+\log(s+1)}$$

as required.

□

4 Online Phase

Our protocol is divided in two phases, a preprocessing phase and an online phase. The preprocessing, which is independent of each party's input, implements a functionality $\mathcal{F}_{\mathsf{Prep}}$ which generates the necessary shared, authenticated values needed to compute the given function securely. This functionality is stated in Fig. 5.

The main difference, with respect to SPDZ, is that instead of generating the random input masks and multiplication triples over the same space as the inputs, we sample them over $\mathbb{Z}_{2^{k+s}}$, even though we are doing computations in $\mathbb{Z}_{2^k}$. In the input phase, this is necessary to mask the parties' input whilst also obtaining a correct MAC over $\mathbb{Z}_{2^{k+s}}$. For the triples, we sample the shares and compute the MACs in $\mathbb{Z}_{2^{k+s}}$, but only care about *correctness* of the multiplication modulo 2^k, so the upper s bits of a triple are just random.[6]

Modulo these differences, the online phase of our protocol, shown in Fig. 7, is similar to that in other secret sharing-based protocols like GMW, BeDOZa, SPDZ and MASCOT [11,2,9,14].

Shares of the inputs are distributed by means of the random shares provided by $\mathcal{F}_{\mathsf{Prep}}$. When an addition gate is found, the parties obtain the output by adding their shares locally. On the other hand, multiplication triples are used for the multiplication gates, where the fact that $x \cdot y = c + \epsilon \cdot b + \delta \cdot a + \epsilon \cdot \delta$

[6] These s bits are not actually required to be random, since whenever we open a value using BatchCheck the upper s bits of all shares are masked anyway. However, it simplifies the description of the functionality to use random shares.

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

The preprocessing functionality has all the same features as $\mathcal{F}_{\mathsf{MAC}}$, with the additional commands:

Input: On input (Input, P_i) from all parties, do the following:

1. Sample a random value $r \in \mathbb{Z}_{2^{k+s}}$ and generate random shares $r = \sum_{j=1}^{n} r^j \bmod 2^{k+s}$. If P_i is corrupted, instead let the adversary choose all shares r^j and compute r accordingly.
2. Run the Auth macro to generate shares and MAC shares of $[r]$.
3. Send r to P_i, and the relevant shares of $[r]$ to each party.

Triple: On input (Triple) from all parties, the functionality performs the following steps

1. Sample random shares $\{(a^j, b^j)\}_{j \notin A} \subseteq (\mathbb{Z}_{2^{k+s}})^2$
2. Wait for input $\{(a^j, b^j, c^j)\}_{j \in A} \subseteq (\mathbb{Z}_{2^{k+s}})^3$ from the adversary and set $c = a \cdot b \bmod 2^k$, where $a = \sum_{j=1}^{n} a^j \bmod 2^k$ and $b = \sum_{j=1}^{n} b^j \bmod 2^k$.
3. Sample $\{c^j\}_{j \notin A} \subseteq \mathbb{Z}_{2^{k+s}}$ and $r \in \mathbb{Z}_{2^s}$ subject to $c + 2^k r \equiv_{k+s} \sum_{j=1}^{n} c^j$.
4. Finally, the functionality runs the Auth macro to generate sharings $[a], [b], [c]$ and sends the j-th output of each result to party P_j.

Fig. 5. Functionality for the preprocessing phase

for $c = a \cdot b$, $\epsilon = x - a$ and $\delta = y - b$ allows us to evaluate multiplications as affine operations on x and y, once the values of ϵ and δ are known. Finally, after checking correctness of all the values opened in multiplications using the batch MAC checking procedure from section 3, the values for the output wires are revealed.

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

Initialization: The functionality receives input (Init, k) from all parties.

Input: On input $(\mathsf{Input}, P_i, vid, x)$ from party P_i and input (Input, P_i) from the other parties, where vid is a fresh, valid identifier, the functionality stores $(vid, x \bmod 2^k)$.

Add: On input $(\mathsf{add}, vid_1, vid_2, vid_3)$ from all parties, the functionality retrieves (if present in memory) the values (vid_1, x_1), (vid_2, x_2) and stores $(vid_3, x_1 + x_2 \bmod 2^k)$.

Multiply: On input $(\mathsf{multiply}, vid_1, vid_2, vid_3)$ from all parties, the functionality retrieves (if present in memory) the values (vid_1, x_1), (vid_2, x_2) and stores $(vid_3, x_1 \cdot x_2 \bmod 2^k)$.

Output: On input (output, vid) from all honest parties, the functionality looks for (vid, y) in memory and if present, sends y to the adversary. The functionality then waits for a message Abort or $\mathsf{Proceed}$ from the adversary: if it sends Abort then the functionality aborts, otherwise the value y is delivered to all parties.

Fig. 6. Ideal functionality for the online phase

Protocol Π_{Online}

The protocol is parameterized by k, which specifies the word size on which the operations are to be performed, and a security parameter s.

Initialize: The parties call the functionality $\mathcal{F}_{\mathsf{Prep}}$ as follows:

1. On input (Init) to get MAC key shares $\alpha^j \in \mathbb{Z}_{2^s}$.
2. On input (Input, P_i) for all parties to obtain random sharings $[r]$ where P_i learns r, for every input that P_i will provide.
3. On input (Triple) to get enough triples $([a], [b], [c])$.

Input: To share an input x^i held by P_i:

1. P_i broadcasts $\epsilon = x^i - r \mod 2^{k+s}$, where $[r]$ is the next unused input mask.
2. The parties compute $[x^i] = [r] + \epsilon$.

Add: To add two values $[x]$ and $[y]$ the parties compute locally $[z] = [x] + [y]$.

Multiply: To multiply two values $[x]$ and $[y]$:

1. Open $[x] - [a]$ as ϵ and $[y] - [b]$ as δ using the **Open** phase of $\mathsf{BatchCheck}$, where $([a], [b], [c])$ is the next unused triple.
2. Locally compute $[x \cdot y] = [c] + \epsilon \cdot [b] + \delta \cdot [a] + \epsilon \cdot \delta$.

Output: To output a value $[y]$:

1. Call the procedure $\mathsf{BatchCheck}$ to check the MACs on the values that have been opened so far in multiplications.
2. If this does not abort, the parties open and check the MAC on $[y]$ using the procedure $\mathsf{SingleCheck}$ from Section 3.1.

Fig. 7. Protocol for reactive secure multi-party computation over $\mathbb{Z}_{2^k}$

The proof of the following theorem is quite straightforward, given the analysis of the MACs in Section 3.

Theorem 2. *The protocol Π_{Online} implements $\mathcal{F}_{\mathsf{Online}}$ in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model, with statistical security parameter s.*

5 Preprocessing: Creating the MACs

We now show how to authenticate additively shared values with the linear MAC scheme, realising the functionality $\mathcal{F}_{\mathsf{MAC}}$ from Section 3 (Fig. 3). Recall that after sampling shares of the MAC key $\alpha \in \mathbb{Z}_{2^s}$, the functionality takes as input secret-shared values $x \in \mathbb{Z}_{2^r}$, and produces shares of the MAC $x \cdot \alpha \bmod 2^\ell$. The input and output widths r and ℓ are parameters with $\ell \geq r$. In our protocol we actually require $\ell \geq 2s$ and $\ell \geq r + s$, where s is the security parameter, but if these do not hold then we work with $\ell' = \max(r + s, 2s)$ and reduce the outputs modulo 2^ℓ.

Building block: vector oblivious linear function evaluation. To create the MACs, we will use a functionality for random vector oblivious linear function evaluation (vector-OLE) over the integers modulo 2^ℓ. This is a protocol between two parties, P_A and P_B, that takes as input a fixed element $\alpha \in \mathbb{Z}_{2^s}$ from party P_A, a vector $\boldsymbol{x}$ from party P_B, then samples a random vector $\boldsymbol{b} \in \mathbb{Z}_{2^\ell}$ as output to P_B, and sends $\boldsymbol{a} = \boldsymbol{b} + \alpha \cdot \boldsymbol{x} \mod 2^\ell$ to P_A. In the specification of our ideal functionality in Fig. 8, $\boldsymbol{x}$ is a vector of length $t+1$, with the first t components from $\mathbb{Z}_{2^r}$ and the final component from $\mathbb{Z}_{2^\ell}$. This is because our MAC generation protocol will create a batch of t MACs at once on r-bit elements, but to do this securely we also need to authenticate an additional random mask of ℓ bits.

Notice that the functionality also allows a corrupted P_B to try to guess a subset of $\mathbb{Z}_{2^s}$ in which α lies, but if the guess is incorrect the protocol aborts. This is needed in order to efficiently implement $\mathcal{F}_{\mathsf{vOLE}}$ using oblivious transfer on correlated messages, based on existing oblivious transfer extension techniques.

Functionality $\mathcal{F}^s_{\mathsf{vOLE}}$

Initialize: On receiving $(sid, \mathsf{Init}, \alpha)$ from P_A, where $\alpha \in \mathbb{Z}_{2^s}$, and (sid, Init) from P_B, store α and ignore any subsequent (sid, Init) messages.

Vector-OLE: On input $(sid, \ell, r, t, \boldsymbol{x})$ from P_B, where $\boldsymbol{x} \in \mathbb{Z}_{2^r}^t \times \mathbb{Z}_{2^\ell}$:

1. Sample $\boldsymbol{b} \leftarrow_R \mathbb{Z}_{2^\ell}^{t+1}$. If P_B is corrupted, instead receive $\boldsymbol{b}$ from $\mathcal{A}$.
2. Compute $\boldsymbol{a} = \boldsymbol{b} + \alpha \cdot \boldsymbol{x} \mod 2^\ell$
3. If P_A is corrupted, receive $\boldsymbol{a} \in \mathbb{Z}_{2^\ell}^t$ from $\mathcal{A}$ and recompute $\boldsymbol{b} = \boldsymbol{a} - \alpha \cdot \boldsymbol{x}$.
4. If P_B is corrupted, wait for $\mathcal{A}$ to input a message (guess, S), where S efficiently describes a subset of $\{0,1\}^s$. If $\alpha \in S$ then send $(\mathsf{success})$ to $\mathcal{A}$. Otherwise, send $\perp$ to both parties and terminate.
5. Output $\boldsymbol{a}$ to P_A and $\boldsymbol{b}$ to P_B.

Fig. 8. Random vector oblivious linear function evaluation functionality over $\mathbb{Z}_{2^{k+s}}$

MAC generation protocol. Each party samples a random MAC key share α^i, and uses this to initialize an instance of $\mathcal{F}_{\mathsf{vOLE}}$ with every other party. On input a vector of additive secret shares $\boldsymbol{x}^i = (x^i_1, \ldots, x^i_t)$ from every P_i, each party samples a random ℓ'-bit mask x^i_{t+1}, and then uses $\mathcal{F}_{\mathsf{vOLE}}$ to compute two-party secret-sharings of the products $\alpha^i \cdot (\boldsymbol{x}^j \| x^j_{t+1})$ for all $j \neq i$. Each party can then obtain a share of the MACs $\alpha \cdot \boldsymbol{x}$ (where $\alpha = \sum \alpha^i$ and $\boldsymbol{x} = \sum \boldsymbol{x}^i$), by adding up all the two-party sharings together with the product $\alpha^i \cdot \boldsymbol{x}^i$.

So far, the protocol is only passively secure, since there is nothing to prevent a corrupt P_j from using inconsistent values of α^j or $\boldsymbol{x}^j$ with two different honest parties, so the corrupt parties' inputs may not be well-defined. To prevent this issue, and ensure that in the security proof the simulator can correctly extract the adversary's inputs, we add a consistency check in steps 6–11: this challenges the parties to open a random linear combination of all authenticated values. This is where we need the additional random mask x_{t+1}, to prevent any leakage on the parties inputs from opening this linear combination. The check does not rule out *all* possible deviations in the protocol, however, in what follows we show that it ensures that the *sum* of all the errors directed towards any given honest party is zero, so these errors all cancel out. Intuitively, this suffices to realise $\mathcal{F}_{\mathsf{MAC}}$ because the functionality only adds a MAC to the sum of all parties' inputs, and not the individual shares themselves.

Protocol Π_{Auth}

Initialize: Each party P_i samples a MAC key share $\alpha^i \leftarrow_R \mathbb{Z}_{2^s}$. Every pair of parties (P_i, P_j) initializes an instance of $\mathcal{F}_{\mathsf{vOLE}}$, where P_i inputs α_i.

Authentication: To authenticate the values $\boldsymbol{x} = (x_1, \ldots, x_t)$ over $\mathbb{Z}_{2^\ell}$, where each party P_j inputs an additive share $\boldsymbol{x}^j \in \mathbb{Z}_{2^r}^t$:

1. Let $\ell' = \max(\ell, r + s, 2s)$.
2. Each party P_j samples a random mask $x^j_{t+1} \leftarrow_R \mathbb{Z}_{2^{\ell'}}$ and defines $\widetilde{\boldsymbol{x}}^j := (\boldsymbol{x}^j, x^j_{t+1}) \in \mathbb{Z}_{2^r}^t \times \mathbb{Z}_{2^{\ell'}}$.
3. Every pair (P_i, P_j) (for $i \neq j$) calls their $\mathcal{F}_{\mathsf{vOLE}}$ instance with input $(\ell', r, t, \tilde{\boldsymbol{x}}^j)$ from P_j.
4. P_j receives $\boldsymbol{b}^{j,i}$ and P_i receives $\boldsymbol{a}^{i,j}$, such that $\boldsymbol{a}^{i,j} = \boldsymbol{b}^{j,i} + \alpha^i \cdot \widetilde{\boldsymbol{x}}^j \mod 2^{\ell'}$.
5. For $h = 1, \ldots, t+1$, each party P_j defines the MAC share
$$m^j_h = \alpha^j \cdot x^j_h + \sum_{i \neq j} (\boldsymbol{a}^{j,i} - \boldsymbol{b}^{j,i})[h] \mod 2^{\ell'}$$

Consistency check:

6. Sample $\chi_1, \ldots, \chi_t \leftarrow_R \mathbb{Z}_{2^s}^t$ using $\mathcal{F}_{\mathsf{Rand}}$.
7. Each party P_j computes and broadcasts $\hat{x}^j = \sum_{i=1}^t x^j_i \cdot \chi_i + x^j_{t+1} \mod 2^{\ell'}$.
8. Each party P_j defines $\hat{m}^j = \sum_{h=1}^t m^j_h \cdot \chi_h + m^j_{t+1} \mod 2^{\ell'}$ and $\hat{x} = \sum_i \hat{x}^i$.
9. Each party P_j commits to and then opens $z^j = \hat{m}^j - \hat{x} \cdot \alpha^j \mod 2^{\ell'}$.
10. All parties check that $\sum_i z^i = 0 \mod 2^{\ell'}$ and abort if the check fails.
11. Each party P_j outputs the MAC shares $m^j_1, \ldots, m^j_t \mod 2^\ell$.

Fig. 9. Protocol for authenticating secret-shared values

5.1 Security

We now analyse the consistency check of the MAC creation protocol. There are two main types of deviations that a corrupt P_j can perform, namely (1) Input inconsistent values of α^j to the initialization phase of $\mathcal{F}_{\mathsf{vOLE}}$ with different honest parties, and (2) Input inconsistent shares $\boldsymbol{x}^j$ in the authentication stage.

For both types of errors, we define the *correct* values $\alpha^j, \boldsymbol{x}^j$ to be those used in the $\mathcal{F}_{\mathsf{vOLE}}$ instance with an arbitrary, fixed honest party, say P_{i_0}. We then define the errors

$$\gamma^{j,i} = \alpha^{j,i} - \alpha^j \quad \text{and} \quad \boldsymbol{\delta}^{j,i} = \boldsymbol{x}^{j,i} - \boldsymbol{x}^j,$$

for each $j \in A$ and $i \notin A$. For an honest party P_i, we also define $\alpha^{i,j}, \boldsymbol{x}^{i,j}$ to be equal to $\alpha^i, \boldsymbol{x}^i$ for all $j \neq i$.

In Claims 2 and 3 below we will show that, if the consistency check passes, then with overwhelming probability the *sum* of all corrupted parties' values is well-defined. That is, the values $\sum_{j \in A} \alpha^j$ and $\sum_{j \in A} \boldsymbol{x}^j$ would be exactly same even if they were defined using the inputs from P_j with a *different* honest party $P_{i_1} \neq P_{i_0}$. Since the MACs are computed based only on the sum of the MAC key shares and input shares, this suffices to prove security of the protocol.

Suppose that the corrupted parties compute the MAC shares $\boldsymbol{m}^j$ as an honest P_j would, using the values $\alpha^j, \boldsymbol{x}^j$ we defined above, as well as the values $\boldsymbol{a}^{j,i}, \boldsymbol{b}^{j,i}$ sent to $\mathcal{F}_{\mathsf{vOLE}}$. Note that even though a corrupt P_j need not do this, any deviation here can be modelled by an additive error in the commitment to z^j in step 9, so we do not lose any generality.

The sum of the vector of MAC shares on $\boldsymbol{x}$ is then given by

$$\begin{aligned}
\sum_i \boldsymbol{m}^i &= \sum_i \alpha^i \cdot \boldsymbol{x}^i + \sum_i \sum_{j \neq i} (\boldsymbol{a}^{i,j} - \boldsymbol{b}^{j,i}) \\
&= \sum_i \alpha^i \cdot \boldsymbol{x}^i + \sum_i \sum_{j \neq i} \alpha^{i,j} \cdot \boldsymbol{x}^{j,i}) \\
&= \alpha \cdot \boldsymbol{x} + \sum_{i \notin A} \boldsymbol{x}^i \cdot \underbrace{\sum_{j \in A} \gamma^{j,i}}_{=\gamma^i} + \sum_{i \notin A} \alpha_i \cdot \underbrace{\sum_{j \in A} \boldsymbol{\delta}^{j,i}}_{=\boldsymbol{\delta}^i}
\end{aligned}$$

After taking random linear combinations with the vector $\boldsymbol{\chi} = (\chi_1, \ldots, \chi_t)$ to compute the MAC on $\hat{x}$, these MAC shares satisfy

$$\sum_i \hat{m}^i = \alpha \cdot \hat{x} + \sum_{i \notin A} (\langle \boldsymbol{x}^i, \boldsymbol{\chi} \rangle + x^i_{t+1}) \cdot \gamma^i + \sum_{i \notin A} \alpha^i \cdot \langle \boldsymbol{\delta}^j, \boldsymbol{\chi} \rangle \tag{1}$$

To pass the consistency check, the adversary must first open the random linear combination $\hat{x}$ to some (possibly incorrect) value, say $\hat{x} + \varepsilon$, in step 7. Then they must come up with an error $\varDelta \in \mathbb{Z}_{2^{\ell'}}$ such that

$$\begin{aligned}
0 &\equiv_{\ell'} \sum_i z^i + \varDelta \\
&\equiv_{\ell'} \sum_i (m^i - (\hat{x} + \varepsilon) \cdot \alpha^i) + \varDelta \\
\Leftrightarrow -\varDelta &\equiv_{\ell'} \sum_i m^i - (\hat{x} + \varepsilon) \cdot \alpha \\
&\equiv_{\ell'} \alpha \cdot \varepsilon + \sum_{i \notin A} \underbrace{(\langle \boldsymbol{x}^i, \boldsymbol{\chi} \rangle + x^i_{t+1})}_{=u^i} \cdot \gamma^i + \sum_{i \notin A} \alpha^i \cdot \langle \boldsymbol{\delta}^j, \boldsymbol{\chi} \rangle \\
-\varDelta - \sum_{j \in A} \alpha^j \cdot \varepsilon &\equiv_{\ell'} \sum_{i \notin A} u^i \cdot \gamma^i + \sum_{i \notin A} \alpha^i \cdot (\langle \boldsymbol{\delta}^j, \boldsymbol{\chi} \rangle + \delta^j_{t+1} + \varepsilon)
\end{aligned}$$

where the last two congruences come from substituting (1) and moving information known by the adversary to the left-hand side.

When proving the two claims below we assume that the adversary does not send any (guess) messages to $\mathcal{F}_{\mathsf{vOLE}}$. Similarly to the proof of Theorem 1, these can easily be extended to handle this case.

Claim 2 *If at least one $\gamma^i \neq 0$ then the probability of passing the check is no more than $2^{-s+\log n}$.*

Proof. Let i be an index for where $\gamma^i \neq 0$. Recall that $\gamma^i = \sum_{j \notin A} \gamma^{j,i}$, where each $\gamma^{j,i} < 2^s$, therefore $\gamma^i < 2^{s+\log n}$. Note that the distribution of u^i is uniform in $\mathbb{Z}_{2^{\ell'}}$ and independent of all other terms, due to the extra mask x^i_{t+1}, so we can write $u^i \cdot \gamma^i \equiv_{\ell'} \Delta'$, for some Δ' that is independent of u^i. Dividing by 2^v, the largest power of two dividing γ^i, we get

$$u^i \cdot \frac{\gamma^i}{2^v} \equiv_{\ell'-v} \frac{\Delta'}{2^v}$$

$$u^i \equiv_{\ell'-v} \frac{\Delta'}{2^v} \cdot \left(\frac{\gamma^i}{2^v}\right)^{-1}$$

Since $v < s + \log n$, this holds with probability at most $2^{-\ell'+s+\log n} \leq 2^{-s+\log n}$ since $\ell' \geq 2s$.

Claim 3 *Suppose $\gamma^i = 0$ for all $i \notin A$, and $\boldsymbol{\delta}^j$ is non-zero modulo 2^k in at least one component for some j. Then, the probability of passing the check is no more than $2^{-s+\log(\ell'-r+1)}$.*

Proof. Pick an honest party, say P_{i_0}, and similarly to the previous claim, we can write the equivalence as

$$\alpha_{i_0} \cdot (\langle \boldsymbol{\delta}^i, \boldsymbol{\chi} \rangle + \delta^i_{t+1} + \varepsilon) \equiv_{\ell'} \Delta'$$

for some Δ' that is independent of the honest party's MAC key share α_{i_0}. We can then apply Lemma 1 with $r = r, m = s, \ell = \ell'$ and $\delta_0 = \delta^i_{t+1} + \varepsilon$ to obtain the bound $2^{-\ell'+r+\log(\ell'-r+1)}$, which proves the claim since $\ell' \geq r + s$.

The above two claims show that, except with negligible probability in s and r, the sum of all errors directed towards any given honest party is zero, so all errors introduced by corrupt parties cancel out and the outputs form a correct MAC on the underlying shared value. In particular, for the security proof, this implies that in the ideal world the MAC shares seen by the environment (including those of honest parties') are identically distributed to the MAC shares output in the real world.

We have the following theorem.

Theorem 3. *The protocol Π_{Auth} securely realises $\mathcal{F}_{\mathsf{MAC}}$ in the $(\mathcal{F}_{\mathsf{vOLE}}, \mathcal{F}_{\mathsf{Rand}})$-hybrid model.*

6 Preprocessing: Creating Multiplication Triples

In this section we focus on developing a protocol that implements the Triple command in the preprocessing functionality. More precisely, let $\mathcal{F}_{\mathsf{Triple}}$ be the functionality that has the same features as $\mathcal{F}_{\mathsf{Prep}}$ (Fig. 5), but without the **Input** command. Our protocol, described in Fig. 10, implements the functionality $\mathcal{F}_{\mathsf{Triple}}$ in the $(\mathcal{F}_{\mathsf{ROT}}, \mathcal{F}_{\mathsf{MAC}}, \mathcal{F}_{\mathsf{Rand}})$-hybrid model.

Protocol Π_{Triple}

The integer parameter $\tau = 4s + 2k$ specifies the size of the input triple used to generate each output triple.

Multiply:

1. Each party P_i samples $\boldsymbol{a}^i = (a_1^i, \ldots, a_\tau^i) \leftarrow_R (\mathbb{Z}_2)^\tau$, $b^i \leftarrow_R \mathbb{Z}_{2^{k+s}}$
2. Every ordered pair of parties (P_i, P_j) does the following:
   (a) Both parties call $\mathcal{F}_{\mathsf{ROT}}^\tau$ with P_i as the receiver and P_j as the sender. P_i inputs the bits $(a_1^i, \ldots, a_\tau^i) \in (\mathbb{Z}_2)^\tau$.
   (b) P_j receives $q_{0,h}^{j,i}, q_{1,h}^{j,i} \in \mathbb{Z}_{2^{k+s}}$ and P_i receives $s_h^{i,j} = q_{a_h^i,h}^{j,i}$ for $h = 1, \ldots, \tau$.
   (c) P_j sends $d_h^{j,i} = q_{0,h}^{j,i} - q_{1,h}^{j,i} + b^j \mod 2^{k+s}$, for $h = 1, \ldots, \tau$.
   (d) P_i sets $t_h^{i,j} = s_h^{i,j} + a_h^i \cdot d_j^{j,i} \mod 2^{k+s}$ for $h = 1, \ldots, \tau$. In particular

$$\begin{aligned} t_h^{i,j} &\equiv_{k+s} s_h^{i,j} + a_h^i \cdot d_j^{j,i} \\ &\equiv_{k+s} q_{a_h^i,h}^{j,i} + a_h^i \cdot \left(q_{0,h}^{j,i} - q_{1,h}^{j,i} + b^j \right) \\ &\equiv_{k+s} q_{0,h}^{j,i} + a_h^i b^j . \end{aligned}$$

   Therefore, the following equation holds modulo 2^{k+s} on each entry

$$\begin{pmatrix} t_1^{i,j} \\ t_2^{i,j} \\ \vdots \\ t_\tau^{i,j} \end{pmatrix} = \begin{pmatrix} q_{0,1}^{j,i} \\ q_{0,2}^{j,i} \\ \vdots \\ q_{0,\tau}^{j,i} \end{pmatrix} + b^j \begin{pmatrix} a_1^i \\ a_2^i \\ \vdots \\ a_\tau^i \end{pmatrix}$$

   (e) P_i sets $\boldsymbol{c}_{i,j}^i = \left(t_1^{i,j}, t_2^{i,j}, \ldots, t_\tau^{i,j} \right) \in (\mathbb{Z}_{2^{k+s}})^\tau$.
   (f) P_j sets $\boldsymbol{c}_{i,j}^j = -\left(q_{0,1}^{j,i}, q_{0,2}^{j,i}, \ldots, q_{0,\tau}^{j,i} \right) \in (\mathbb{Z}_{2^{k+s}})^\tau$.
   (g) The following congruence holds

$$\boldsymbol{c}_{i,j}^i + \boldsymbol{c}_{i,j}^j \equiv_{k+s} \boldsymbol{a}^i \cdot b^j ,$$

   where the modulo congruence is component-wise.
3. Each party P_i computes:

$$\boldsymbol{c}^i = \boldsymbol{a}^i \cdot b^i + \sum_{j \neq i} (\boldsymbol{c}_{i,j}^i + \boldsymbol{c}_{j,i}^i) \mod 2^{k+s}$$

Fig. 10. Triple generation protocol

Protocol Π_{Triple} (continuation)

Combine:

1. Sample $\boldsymbol{r}, \hat{\boldsymbol{r}} \leftarrow_R \mathcal{F}_{\mathsf{Rand}}\left((\mathbb{Z}_{2^{k+s}})^{\tau}\right)$.
2. Each party P_i sets

$$a^i = \sum_{h=1}^{\tau} r_h \boldsymbol{a}^i[h] \mod 2^{k+s}, \qquad c^i = \sum_{h=1}^{\tau} r_h \boldsymbol{c}^i[h] \mod 2^{k+s} \qquad \text{and}$$

$$\hat{a}^i = \sum_{h=1}^{\tau} \hat{r}_h \boldsymbol{a}^i[h] \mod 2^{k+s}, \qquad \hat{c}^i = \sum_{h=1}^{\tau} \hat{r}_h \boldsymbol{c}^i[h] \mod 2^{k+s}$$

Authenticate: Each party P_i runs $\mathcal{F}_{\mathsf{MAC}}$ on their shares to obtain authenticated shares $[a], [b], [c], [\hat{a}], [\hat{c}]$.

Sacrifice: Check correctness of the triple $([a], [b], [c])$ by sacrificing $[\hat{a}], [\hat{c}]$.

1. Sample $t := \mathcal{F}_{\mathsf{Rand}}(\mathbb{Z}_{2^s})$.
2. Execute the procedure AffineComb to compute $[\rho] = t \cdot [a] - [\hat{a}]$
3. Execute the procedure BatchCheck on $[\rho]$ to obtain ρ.
4. Execute the procedure AffineComb to compute $[\sigma] = t \cdot [c] - [\hat{c}] - [b] \cdot \rho$.
5. Run BatchCheck on $[\sigma]$ to obtain σ, and abort if this value is not zero modulo 2^{k+s}.

Output: Generate using $\mathcal{F}_{\mathsf{MAC}}$ a random value $[r]$ with $r \in \mathbb{Z}_{2^s}$. Output $([a], [b], [c + 2^k r])$ as a valid triple.

Fig. 11. Triple generation protocol (continuation)

The protocol itself is very similar to the one used in MASCOT [14], with several changes introduced in order to cope with the fact that our ring $\mathbb{Z}_{2^k}$ has non-invertible elements. Most of these changes involve taking the coefficients of random linear combinations in a different ring $\mathbb{Z}_{2^s}$, which is useful to argue that certain equations of the form $r \cdot a \equiv_{k+s} b$ are satisfied with low probability. This can be seen for example in the sacrifice step, where the random value t is chosen to have at least s random bits, instead of k. Additionally, in our protocol (like in MASCOT) random linear combinations must be used to extract randomness from partially leaked values $a_1, \ldots, a_t$, which still have reasonably high entropy. In order to use the Leftover Hash Lemma in this context one needs to make sure that taking random linear combinations yields a universal hash function. However, in contrast to the field case it is not true in general that the function $r_1 \cdot a_1 + \cdots + r_t \cdot a_t \mod 2^k$ is universal, unless we make some assumptions about the set the values a_i are picked from. In the case of our protocol, we force the a_i to be $-1, 0$ or 1. With this additional condition it can be shown that the function above is universal.

The **Multiply** phase generates shares $\{(\boldsymbol{a}^i, b^i, \boldsymbol{c}^i)\}_{i=1}^n$ such that P_i has $(\boldsymbol{a}^i, b^i, \boldsymbol{c}^i)$, where $\boldsymbol{a}^i$ is a vector of bits, b^i is a random element of $\mathbb{Z}_{2^{k+s}}$ and $\boldsymbol{c}^i$ is a vector of random elements of $\mathbb{Z}_{2^{k+s}}$. These values satisfy $\boldsymbol{c} = \boldsymbol{a} \cdot b$, where $\boldsymbol{c} = \sum_{i=1}^n \boldsymbol{c}^i \mod 2^{k+s}$, $\boldsymbol{a} = \sum_{i=1}^n \boldsymbol{a}^i \mod 2^{k+s}$ and $b = \sum_{i=1}^n b^i \mod 2^{k+s}$. This is achieved by letting the parties choose their shares on $\boldsymbol{a}$ and b, and using oblivious transfer to compute the cross products $\boldsymbol{a}^i \cdot b^j$. However, this is not a fully functional multiplication triple yet as it might not satisfy the right multiplicative relation (besides other technical issues like $\boldsymbol{a}$ being a short vector, and not a value in $\mathbb{Z}_{2^{k+s}}$). To check that the triple is correct, the **Sacrifice** phase uses another triple to check correctness. As the name suggests, one triple is "sacrificed" (i.e. opened) so that we can check correctness of the other while keeping it secret.

On the other hand, we must also ensure that the triple looks random to all parties. As we will see shortly in the proof of Theorem 4, if the triple is correct this will reveal some partial information about the honest parties' shares to the adversary. This means that the adversary can guess a particular bit of these shares, which would allow him to distinguish in the simulation. This issue is addressed by the step **Combine**, which takes place before the Sacrifice step. Here the parties take a random linear combination of $\boldsymbol{a}$. Now, in order to pass the check, the adversary has to guess a random combination of the bits of $\boldsymbol{a}$, which is much harder.

At this point a triple $([a], [b], [c])$ has been created, with $c \equiv_k a \cdot b$. However, the s most significant bits of c have some information that could allow the adversary to guess the shares of a of the honest parties. Moreover, correctness of the triple is only required modulo 2^k, as this is the modulus in the circuit the parties want to compute. Therefore, in order to mitigate this issue the parties use a random authenticated mask to hide the s most significant bits of c. This mask is very similar to the one used in the procedure SingleCheck from Section 3.1. In fact, in an actual implementation we could ignore the mask on the triples, as these will be masked before opening in the MAC checking procedures. However, if we wish to apply the Composition Theorem to our final protocol, each subprotocol must be UC secure by itself, regardless of any further composition.

Now we proceed with the main theorem of the section, which states the security of the protocol in Fig. 10.

Theorem 4. *If $\tau \geq 4k + 2s$, then the protocol Π_{Triple} (Protocol 10) securely implements $\mathcal{F}_{\mathsf{Triple}}$ in the $(\mathcal{F}_{\mathsf{ROT}}, \mathcal{F}_{\mathsf{MAC}}, \mathcal{F}_{\mathsf{Rand}})$-hybrid model, with statistical security parameter k.*

Proof. Let $\mathcal{Z}$ be an environment, which we also refer to as adversary, corrupting a set A of at most $n-1$ parties. We construct a simulator $\mathcal{S}$ that has access to the ideal functionality $\mathcal{F}_{\mathsf{Triple}}$ and interacts with $\mathcal{Z}$ in such a way that the real interaction and the simulated interaction are indistinguishable to $\mathcal{Z}$. Our simulator $\mathcal{S}$ proceeds as follows:

Simulating the Multiply phase The simulator emulates the functionality $\mathcal{F}^{\tau}_{\mathsf{ROT}}$ and sends $q^{j,i}_{0,h}, q^{j,i}_{1,h} \in \mathbb{Z}_{2^k}$ for $h \in \{1, \ldots, \tau\}$ to every $j \in A$ (on behalf of each honest party P_i). When a corrupted party P_j sends $d^{j,i}_h$ to an honest party P_i, $h \in \{1, \ldots, \tau\}$, the simulator uses its knowledge on the q's to extract the values of b used by the adversary as $b^j_h = d^{j,i}_h - q^{j,i}_{0,h} + q^{j,i}_{1,h} \mod 2^k$ (notice that if all the parties were honest we would have that all b^j_h for $h \in \{1, \ldots, \tau\}$ are equal, however, the adversary can take any strategy and this may not

be the case here). The simulator then emulates the multiplication procedure according to the protocol using a fixed consistent value b^j for each $j \in A$ (say the value of b_1^j used with a fixed honest party P_{i_0}). We let $\boldsymbol{b}^{j,i} \in (\mathbb{Z}_{2^{k+s}})^\tau$ denote the vector of values of b that P_j tried to use in interaction with the emulated honest party P_i in step (c) and we define $\boldsymbol{\delta}_b = \boldsymbol{b}^{j,i} - \boldsymbol{b}^j$ (modulo 2^{k+s} on each entry) where $\boldsymbol{b}^j$ is the vector $(b^j, \dots, b^j)$.
In a similar way, we define $\boldsymbol{\delta}_a^{j,i} = \boldsymbol{a}^{j,i} - \boldsymbol{a}^j$ where $\boldsymbol{a}^j = \boldsymbol{a}^{j,i_0}$ (these are the errors introduced by P_j when interacting with P_i with respect to the values used in the interaction with P_{i_0}) and $\boldsymbol{a}^{j,i}$ is the vector that the corrupt party P_j used in the random OT when interacting with honest party P_i. Notice that $\boldsymbol{\delta}_a^{j,i} \in \{-1, 0, 1\}^\tau$.

Simulating the Combining phase All the computations are local, so $\mathcal{S}$ just emulates $\mathcal{F}_{\mathsf{Rand}}$ and proceeds according to the protocol.

Simulating the Authentication phase Now $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{MAC}}$ with inputs from the corrupt parties provided by $\mathcal{Z}$. Notice that $\mathcal{S}$ can compute the actual values that each corrupt party should authenticate. The simulator authenticates these and defines e_{Auth} and $\hat{e}_{Auth}$ to be the total error introduced by the adversary in this step. Note that here $e_{Auth}, \hat{e}_{Auth} \neq 0$ essentially means that the adversary authenticates values different from those computed in the previous phases. If $\mathcal{Z}$ sends Abort to $\mathcal{F}_{\mathsf{MAC}}$ then $\mathcal{S}$ sends Abort to $\mathcal{F}_{\mathsf{Triple}}$.

Simulating the Sacrifice step The simulator opens a uniform value in $\mathbb{Z}_{2^{k+s}}$ as the value of ρ, and aborts if the triple that it has internally stored is incorrect modulo 2^k. Otherwise it stores this triple as a valid triple.

Now we argue that the environment $\mathcal{Z}$ cannot distinguish between the hybrid execution and the simulated one. We begin by noticing that in the **Multiply** phase the adversary only learns the mask $d_h^{i,j}$ for each $i \in A$, but they look perfectly random as the values $q_{1-a_h^j,h}^{i,j}$ are uniformly random and never revealed to $\mathcal{Z}$. On the other hand, we still need to argue that the value ρ during the **Sacrifice** step has indistinguishable distributions in both executions, and that the triple $([a], [b], [c])$ obtained in the real execution is indistinguishable from the triple generated in the ideal execution (where a and b are uniformly random).

In order to analyze these distributions, we study what is the effect of the adversarial behavior in the final shared value $\boldsymbol{c}$, and we do this by considering what happens in the real execution at the end of step 2 when executed by a pair of parties (P_i, P_j). If both j and i are honest, then the vectors $\boldsymbol{c}_{i,j}^i$ and $\boldsymbol{c}_{i,j}^j$ computed at the end of the execution satisfy $\boldsymbol{c}_{i,j}^i + \boldsymbol{c}_{i,j}^j \equiv_{k+s} \boldsymbol{a}^i \cdot b^j$. Also, if j and i are both corrupt then we can safely assume that $\boldsymbol{c}_{i,j}^i + \boldsymbol{c}_{i,j}^j \equiv_{k+s} \boldsymbol{a}^i \cdot b^j$ also holds, since any variation on this will result in an additive error term which depends only in adversarial values and therefore it will get absorbed by the authentication phase. Now suppose that j is corrupt and i is honest, then P_i uses $\boldsymbol{a}^i$ and P_j uses $\boldsymbol{b}^{j,i}$, so the vectors $\boldsymbol{c}_{i,j}^i$ and $\boldsymbol{c}_{i,j}^j$ computed at the end of the execution satisfy

$$\boldsymbol{c}_{i,j}^i + \boldsymbol{c}_{i,j}^j \equiv_{k+s} \boldsymbol{a}^i \cdot \boldsymbol{b}^{j,i} \equiv_{k+s} \boldsymbol{a}^i \cdot \boldsymbol{\delta}_b^{j,i} + \boldsymbol{a}^i \cdot b^j.$$

Similarly, if i is corrupt and j is honest, then P_i uses $\boldsymbol{a}^{i,j}$ and P_j uses b^j, so the vectors $\boldsymbol{c}^i_{i,j}$ and $\boldsymbol{c}^j_{i,j}$ computed at the end of the execution satisfy

$$\boldsymbol{c}^i_{i,j} + \boldsymbol{c}^j_{i,j} \equiv_{k+s} \boldsymbol{a}^{i,j} \cdot b^j \equiv_{k+s} \boldsymbol{\delta}^{i,j}_a \cdot b^j + \boldsymbol{a}^i \cdot b^j.$$

Now, if $\boldsymbol{c}^i$ is the vector obtained by party P_i at the end of the multiplication, then we have that

$$\boldsymbol{c} \equiv_{k+s} \boldsymbol{a} \cdot b + \underbrace{\sum_{i \notin A} \boldsymbol{a}^i * \boldsymbol{\delta}^i_b}_{\boldsymbol{e}_a} + \underbrace{\sum_{j \notin A} \boldsymbol{\delta}^j_a \cdot b^j}_{\boldsymbol{e}_b}$$

where $\boldsymbol{a} = \sum_{i=1}^n \boldsymbol{a}^i$, $b = \sum_{i=1}^n b^i$, $\boldsymbol{\delta}^i_b = \sum_{j \in A} \boldsymbol{\delta}^{j,i}_b$ and $\boldsymbol{\delta}^j_a = \sum_{i \in A} \boldsymbol{\delta}^{i,j}_a$, and all congruences are considered component-wise. Notice that each entry in $\boldsymbol{\delta}^j_a$ is the sum of at most n bits and therefore it is upper bounded strictly by n, since we assume that $n \ll 2^{k+s}$ we can consider the sum $\boldsymbol{a} = \sum_{i=1}^n \boldsymbol{a}^i$ (without the modulus).

Assume all parties (including corrupt ones) take the right linear combination in the combine phase (every adversarial misbehavior will result in an additive error term that only depends on values that the adversary has, and this term will be absorbed by the error term in the authentication phase). Therefore, after the combination and authentication phases the parties obtain values $[b]$, $[a]$, $[c]$, $[\hat{a}]$, $[\hat{c}]$ where $b, a, c, \hat{a}, \hat{c} \in \mathbb{Z}_{2^{k+s}}$ satisfy

$$c \equiv_{k+s} a \cdot b + e_a + e_b + e_{Auth}$$
$$\hat{c} \equiv_{k+s} \hat{a} \cdot b + \hat{e}_a + \hat{e}_b + \hat{e}_{Auth}$$

and

$$\begin{aligned} c &\equiv_{k+s} \sum_{h=1}^{\tau} r_h \cdot \boldsymbol{c}[h], & \hat{c} &= \sum_{h=1}^{\tau} \hat{r}_h \cdot \boldsymbol{c}[h] \\ a &\equiv_{k+s} \sum_{h=1}^{\tau} r_h \cdot \boldsymbol{a}[h], & \hat{a} &\equiv_{k+s} \sum_{h=1}^{\tau} \hat{r}_h \cdot \boldsymbol{a}[h] \\ e_a &\equiv_{k+s} \sum_{h=1}^{\tau} r_h \cdot \boldsymbol{e}_a[h], & \hat{e}_a &\equiv_{k+s} \sum_{h=1}^{\tau} \hat{r}_h \cdot \boldsymbol{e}_a[h] \\ e_b &\equiv_{k+s} \sum_{h=1}^{\tau} r_h \cdot \boldsymbol{e}_b[h], & \hat{e}_b &\equiv_{k+s} \sum_{h=1}^{\tau} \hat{r}_h \cdot \boldsymbol{e}_b[h]. \end{aligned}$$

We prove the following two claims can be proven using the same techniques as in the single and batch MAC checking protocols from Section 3, and Lemma 1.

Claim 4 *If the sacrifice step passes, then it holds that* $e := e_a + e_b + e_{Auth} \equiv_k 0$ *and* $\hat{e} := \hat{e}_a + \hat{e}_b + \hat{e}_{Auth} \equiv_k 0$ *with probability at least* $1 - 2^{-s}$.

Claim 5 *Suppose that the sacrificing step passes, then all the errors* $\{\boldsymbol{\delta}^i_a[h]\}_{h, i \notin A}$ *are zero except with probability at most* $2^{-k+\log(n \cdot (k+1-\log n))}$

The previous claim allows us to conclude that $e_b = \hat{e}_b \equiv_{k+s} 0$, except with negligible probability. Now we would like to claim that the value $\rho \in \mathbb{Z}_{2^{k+s}}$ opened in the sacrifice step is indistinguishable from the one opened in the real execution. Since in the ideal execution the simulator opens a uniform value, what we actually need to show is that in a real execution ρ looks (close to) uniform. Given that $\rho = t \cdot a - \hat{a} \mod 2^k$, this can be accomplished by showing that $\hat{a}$ looks uniform to the environment. In order to see that $\hat{a} \equiv_{k+s} \sum_{h=1}^{\tau} \hat{r}_h \cdot \boldsymbol{a}[h] \equiv_{k+s} \sum_{i=1}^{n} \left(\sum_{h=1}^{\tau} \hat{r}_h \cdot \boldsymbol{a}^i[h] \right)$ is uniformly distributed it suffices to show that at least for one $i_0 \notin A$ it holds that $\hat{a}^{i_0}$ looks uniform to the environment, where $\hat{a}^i = \sum_{h=1}^{\tau} \hat{r}_h \cdot \boldsymbol{a}^i[h] \mod 2^{k+s}$, and that all these values are actually independent. This can be shown using the Leftover Hash Lemma by giving a good lower bound on the min-entropy of $\boldsymbol{a}^{i_0}$. We proceed with the details below.

Using Claim 4 and Claim 5, we have that whenever the sacrifice step passes it holds that

$$-e_{Auth} \equiv_k e_a \equiv_k \sum_{h=1}^{\tau} r_h \cdot \boldsymbol{e}_a[h] \equiv_k \sum_{h=1}^{\tau} r_h \sum_{i \notin A} \boldsymbol{a}^i[h] \cdot \boldsymbol{\delta}_b^i[h].$$

and

$$-\hat{e}_{Auth} \equiv_k \hat{e}_a \equiv_k \sum_{h=1}^{\tau} \hat{r}_h \cdot \boldsymbol{e}_a[h] \equiv_k \sum_{h=1}^{\tau} \hat{r}_h \sum_{i \notin A} \boldsymbol{a}^i[h] \cdot \boldsymbol{\delta}_b^i[h].$$

Intuitively, the only information that the adversary has about the honest party's shares is that the sacrifice step passed, which in turn implies that the above equation holds. Ideally, the fact that this relation holds should not reveal so much information about $\{\boldsymbol{a}^i\}_{i \notin A}$ to the adversary. Indeed, this will be the case, which will be seen when we bound by below the entropy of this random variable. To this end, let $m = n - |A|$ be the number of honest parties and let $S \subseteq \mathbb{Z}_2^{m \cdot \tau}$ be the set of all possible honest shares $(\boldsymbol{a}^i)_{i \notin A}$ for which the sacrifice step would pass. Notice that in particular, these shares satisfy the equations above and therefore they are completely determined by the errors that are introduced by the adversary. Moreover, since the shares $(\boldsymbol{a}^i)_{i \notin A}$ are uniformly distributed in S, the min-entropy of these shares is $\log |S|$. Additionally, the vectors in $(\boldsymbol{a}^i)_{i \notin A}$ are independent one from each other, hence there is at least one honest party P_{i_0} such that the min entropy of $\boldsymbol{a}^{i_0}$ is at least $\frac{\log |S|}{m}$. In the following we show that $a^{i_0} = \sum_{h=1}^{\tau} r_h \cdot \boldsymbol{a}^{i_0}[h] \mod 2^{k+s}$ and $\hat{a}^{i_0} = \sum_{h=1}^{\tau} \hat{r}_h \cdot \boldsymbol{a}^{i_0}[h] \mod 2^{k+s}$ look random to the environment.

Let β be the probability of passing the sacrifice step, i.e. $\beta = \frac{|S|}{2^{m\tau}} = 2^{-c}$ where $c = m\tau - \log |S|$. We get that

$$H_\infty \left(\boldsymbol{a}^{i_0} \right) \geq \frac{\log |S|}{m} = \tau - \frac{c}{m} \geq \tau - c.$$

Now consider the function $h_{\boldsymbol{r},\hat{\boldsymbol{r}}} : (\mathbb{Z}_2)^\tau \to (\mathbb{Z}_{2^{k+s}})^2$ given by

$$h_{\boldsymbol{r},\hat{\boldsymbol{r}}}(\boldsymbol{a}) = \left(\sum_{h=1}^{\tau} \boldsymbol{r}[h] \cdot \boldsymbol{a}[h] \mod 2^{k+s}, \quad \sum_{h=1}^{\tau} \hat{\boldsymbol{r}}[h] \cdot \boldsymbol{a}[h] \mod 2^{k+s} \right),$$

We claim that this family of functions is 2−universal. Let $\boldsymbol{a}, \boldsymbol{a}' \in (\mathbb{Z}_2)^\tau$ such that $\boldsymbol{a} \neq \boldsymbol{a}'$, say $\boldsymbol{a}[h_0] \not\equiv_{k+s} \boldsymbol{a}'[h_0]$. If $h_{\boldsymbol{r},\hat{\boldsymbol{r}}}(\boldsymbol{a}) = h_{\boldsymbol{r},\hat{\boldsymbol{r}}}(\boldsymbol{a}')$ then $\sum_{h=1}^{\tau} \boldsymbol{r}[h] \cdot (\boldsymbol{a}[h] - \boldsymbol{a}'[h]) \equiv_{k+s} 0$ and $\sum_{h=1}^{\tau} \hat{\boldsymbol{r}}[h] \cdot (\boldsymbol{a}[h] - \boldsymbol{a}'[h]) \equiv_{k+s} 0$. Given that $\boldsymbol{a}$ and $\boldsymbol{a}'$ are vectors of bits, we have that $\boldsymbol{a}[h_0] - \boldsymbol{a}'[h_0] = \pm 1$, so we can solve for $\boldsymbol{r}[h_0]$ and $\hat{\boldsymbol{r}}[h_0]$ in the equations above. Therefore, these equations hold with probability at most $\frac{1}{2^{k+s}} \cdot \frac{1}{2^{k+s}} = \frac{1}{2^{2(k+s)}}$ over the choice of $(\boldsymbol{r}, \hat{\boldsymbol{r}})$, and hence the family is 2-universal.

According to the Leftover Hash Lemma, even if the adversary knows $\boldsymbol{r}$ and $\hat{\boldsymbol{r}}$, the statistical distance between $h_{\boldsymbol{r},\hat{\boldsymbol{r}}}(X)$ and the uniform distribution in $(\mathbb{Z}_{2^{k+s}})^2$ is at most $2^{-\kappa}$, provided that $H_\infty(X) \geq 2\kappa + 2(k+s)$. This is satisfied if we take $\kappa = \frac{1}{2} \cdot (\tau - c - 2 \cdot (k+s))$.

Finally, ignoring the event in which the check passes with some non-zero errors, which happens with negligible probability, the distinguishing advantage of $\mathcal{Z}$ is the multiplication between the probability of passing the sacrifice step and the probability of distinguishing the output distribution from random, given that the check passed. This is equal to

$$\beta \cdot 2^{-\kappa} = 2^{-c} \cdot 2^{-\frac{1}{2} \cdot (\tau - c - 2 \cdot (k+s))} = 2^{-\frac{\tau - 2 \cdot (k+s)}{2} - \frac{c}{2}}.$$

Since we want this probability to be bounded by 2^{-s} for any c, we take τ so that $s \leq \frac{\tau - 2 \cdot (k+s)}{2}$, which is equivalent to $\tau \geq 4s + 2k$. □

7 Efficiency Analysis

We now turn to estimating the efficiency of our preprocessing protocol, focusing on the triple generation phase since this is likely to be the bottleneck in most applications. We emphasise that the costs presented here, compared with those of previous protocols, do not take into account the benefits to applications from working over $\mathbb{Z}_{2^k}$ instead of a finite field with arithmetic modulo a prime. Supporting natural arithmetic modulo 2^k offers advantages on several levels: it simplifies implementations by avoiding the need for modular arithmetic, it reduces the complexity of compiling existing programs into arithmetic circuits, and we believe that it will also be beneficial in performing operations such as secure comparison and bit decomposition of shared values more efficiently than standard techniques using arithmetic modulo p.

Cost of the preprocessing. When authenticating a secret-shared value $x \in \mathbb{Z}_{2^k}$, the main cost is running the vector OLEs, which have inputs over $\mathbb{Z}_{2^k}$ and outputs over $\mathbb{Z}_{2^{k+s}}$, when the MAC key $\alpha \in \mathbb{Z}_{2^s}$. Each vector OLE requires s correlated OTs on messages over $\mathbb{Z}_{2^\ell}$, where $\ell = \max(k+s, 2s)$, which gives an amortized cost of $s \cdot \ell$ bits for each component of the vector OLE. We ignore the cost of the consistency check, since this is independent of the number of values being authenticated.

To generate a triple, we need τ random OTs on strings of length $k+s$ bits, which cost $k+s$ bits of communication each using [13], followed by $\tau \cdot (k+s)$

bits to send the $d^{j,i}$ values. The parties then authenticate 5 values in $\mathbb{Z}_{2^{k+s}}$, which requires generating MACs modulo $\mathbb{Z}_{2^{k+2s}}$ for security. Generating these MACs costs $5 \cdot s \cdot (k+2s) \cdot n(n-1)$ bits of communication using Π_{Auth} based on correlated OT, since the vector OLEs are performed with $\ell = k + 2s$. The costs of $\mathcal{F}_{\mathsf{Rand}}$ and the sacrifice check are negligible compared to this, since the MAC check can be performed in a batch when producing many triples at once. This gives a total cost estimate of $5s(k+2s) + 2\tau(k+s)$ bits per triple. Setting $\tau = 4s + 2k$ (to give failure probability 2^{-s}) this becomes $2(k+2s)(9s+4k)$.

Comparison with MASCOT. Table 1 shows the estimated communication complexity of our protocol for two parties creating a triple in different rings. Note that like MASCOT [14] — the most practical OT-based protocol for actively secure, dishonest majority MPC over finite fields — we expect that communication will be the bottleneck, since the protocol has very simple computational costs. In the table we fix the computational security parameter to 128, and set the statistical security parameter to $s = 64$ in a 64 or 128-bit ring, or $s = 32$ in the 32-bit ring, giving the claimed security bounds (cf. Theorem 1 and Claim 5). Compared with MASCOT, our protocol needs around twice as much communication for 64 or 128-bit triples, with roughly the same level of statistical security. Over the integers modulo 2^{32}, the overhead reduces to around 50% more than MASCOT, although here the statistical security parameters of 26 and 32 bits may be too low for some applications. Note that many applications will not be possible with MASCOT or SPDZ over a 32-bit field, since here integer overflow (modulo p) occurs more easily, and emulating operations such as secure comparison and bit decomposition over a field requires working with a much larger modulus to avoid overflow. When working over $\mathbb{Z}_{2^{32}}$ instead, this should not be necessary.

These overheads for triple generation, compared with MASCOT, come from the fact that our protocol sometimes needs to work in larger rings to ensure security. For example, for the triple check to be secure, our protocol authenticates shares of triples modulo 2^{k+s}, even though the triples are only ever used modulo 2^k in the online phase. This means that when creating these MACs with the protocol from Section 5, we need to work over $\mathbb{Z}_{2^{k+2s}}$ to ensure security. We leave it to future work to try to avoid these costs and improve efficiency.

Comparison with SPDZ using homomorphic encryption. In very recent work [15], Keller, Pastro and Rotaru presented a new variant of the SPDZ protocol that improves upon the performance of MASCOT. In the two-party setting, they show that an optimized implementation of the original SPDZ [9] runs around twice as fast as MASCOT, and give a new variant that performs 6 times as fast in 64-bit fields; this would probably be around 12 times as fast as our protocol for 64-bit rings. The original SPDZ uses somewhat homomorphic encryption based on the ring-LWE assumption, while their newer variant uses additively homomorphic encryption, and the conjecture that ring-LWE based additively homomorphic encryption has "linear-only" homomorphism. It seems likely that both of these protocols could be adapted to generate triples over $\mathbb{Z}_{2^k}$

Protocol	Message space	Stat. security	Input cost (kbit)	Triple cost (kbit)
Ours	$\mathbb{Z}_{2^{32}}$	26	3.17	79.87
	$\mathbb{Z}_{2^{64}}$	57	12.48	319.49
	$\mathbb{Z}_{2^{128}}$	57	16.64	557.06
MASCOT	32-bit field	32	1.06	51.20
	64-bit field	64	4.16	139.26
	128-bit field	64	16.51	360.44

Table 1. Communication cost of our protocol and previous protocols for various rings and fields, and statistical security parameters

using our techniques. One challenge, however, is to adapt the ciphertext packing techniques used in SPDZ for messages over $\mathbb{F}_p$ to the case of $\mathbb{Z}_{2^k}$, to allow parallel homomorphic operations on ciphertexts; it was shown how this can be done in [10], but it's not clear how efficient this method is in practice.

Acknowledgements

This work has been supported by the European Research Council (ERC) under the European Unions's Horizon 2020 research and innovation programme under grant agreement No 669255 (MPCPRO); the European Union's Horizon 2020 research and innovation programme under grant agreement No 731583 (SODA); the European Union's Horizon 2020 research and innovation programme under grant agreement No 74079 (ALGSTRONGCRYPTO); and the Danish Independent Research Council under Grant-ID DFF–6108-00169 (FoCC).

References

1. Asharov, G., Lindell, Y., Schneider, T., and Zohner, M. More efficient oblivious transfer extensions with security for malicious adversaries. In *EUROCRYPT 2015, Part I* (Apr. 2015), E. Oswald and M. Fischlin, Eds., vol. 9056 of *LNCS*, Springer, Heidelberg, pp. 673–701.
2. Bendlin, R., Damgård, I., Orlandi, C., and Zakarias, S. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT 2011* (May 2011), K. G. Paterson, Ed., vol. 6632 of *LNCS*, Springer, Heidelberg, pp. 169–188.
3. Bogdanov, D., Laur, S., and Willemson, J. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS 2008* (Oct. 2008), S. Jajodia and J. López, Eds., vol. 5283 of *LNCS*, Springer, Heidelberg, pp. 192–206.
4. Canetti, R. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS* (Oct. 2001), IEEE Computer Society Press, pp. 136–145.
5. Cramer, R., Fehr, S., Ishai, Y., and Kushilevitz, E. Efficient multi-party computation over rings. In *EUROCRYPT 2003* (May 2003), E. Biham, Ed., vol. 2656 of *LNCS*, Springer, Heidelberg, pp. 596–613.
6. Damgård, I., Damgård, K., Nielsen, K., Nordholt, P. S., and Toft, T. Confidential benchmarking based on multiparty computation. In *FC 2016* (Feb. 2016), J. Grossklags and B. Preneel, Eds., vol. 9603 of *LNCS*, Springer, Heidelberg, pp. 169–187.
7. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., and Smart, N. P. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS 2013* (Sept. 2013), J. Crampton, S. Jajodia, and K. Mayes, Eds., vol. 8134 of *LNCS*, Springer, Heidelberg, pp. 1–18.
8. Damgård, I., Orlandi, C., and Simkin, M. Yet another compiler for active security or: Efficient MPC over arbitrary rings. In *CRYPTO 2018*
9. Damgård, I., Pastro, V., Smart, N. P., and Zakarias, S. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO 2012* (Aug. 2012), R. Safavi-Naini and R. Canetti, Eds., vol. 7417 of *LNCS*, Springer, Heidelberg, pp. 643–662.
10. Gentry, C., Halevi, S., and Smart, N. P. Better bootstrapping in fully homomorphic encryption. In *PKC 2012* (May 2012), M. Fischlin, J. Buchmann, and M. Manulis, Eds., vol. 7293 of *LNCS*, Springer, Heidelberg, pp. 1–16.
11. Goldreich, O., Micali, S., and Wigderson, A. How to play any mental game or A completeness theorem for protocols with honest majority. In *19th ACM STOC* (May 1987), A. Aho, Ed., ACM Press, pp. 218–229.
12. Impagliazzo, R., Levin, L. A., and Luby, M. Pseudo-random generation from one-way functions (extended abstracts). In *21st ACM STOC* (May 1989), ACM Press, pp. 12–24.
13. Keller, M., Orsini, E., and Scholl, P. Actively secure OT extension with optimal overhead. In *CRYPTO 2015, Part I* (Aug. 2015), R. Gennaro and M. J. B. Robshaw, Eds., vol. 9215 of *LNCS*, Springer, Heidelberg, pp. 724–741.
14. Keller, M., Orsini, E., and Scholl, P. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In *ACM CCS 16* (Oct. 2016), E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds., ACM Press, pp. 830–842.
15. Keller, M., Pastro, V., and Rotaru, D. Overdrive: Making SPDZ great again. In *EUROCRYPT* (2018), LNCS. https://eprint.iacr.org/2017/1230.

16. Nielsen, J. B., Nordholt, P. S., Orlandi, C., and Burra, S. S. A new approach to practical active-secure two-party computation. In *CRYPTO 2012* (Aug. 2012), R. Safavi-Naini and R. Canetti, Eds., vol. 7417 of *LNCS*, Springer, Heidelberg, pp. 681–700.

17. Nielsen, J. B., Schneider, T., and Trifiletti, R. Constant round maliciously secure 2pc with function-independent preprocessing using LEGO. In *24th NDSS Symposium* (2017), The Internet Society. http://eprint.iacr.org/2016/1069.

18. Pettai, M., and Laud, P. Automatic proofs of privacy of secure multi-party computation protocols against active adversaries. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015* (2015), IEEE, pp. 75–89.

19. Scholl, P. Extending oblivious transfer with low communication via key-homomorphic PRFs. In *Public-Key Cryptography (PKC)* (2018), Lecture Notes in Computer Science. https://eprint.iacr.org/2018/036.