Committed MPC

Maliciously Secure Multiparty Computation from Homomorphic Commitments

Tore K. Frederiksen[1], Benny Pinkas[2], and Avishay Yanai[2]

[1] Security Lab, Alexandra Institute, Denmark⋆
[2] Department of Computer Science, Bar-Ilan University, Israel⋆⋆
tore.frederiksen@alexandra.dk, benny@pinkas.net, ay.yanay@gmail.com

Abstract. We present a new multiparty computation protocol secure against a static and malicious dishonest majority. Unlike most previous protocols that were based on working on MAC-ed secret shares, our approach is based on computations on homomorphic commitments to secret shares. Specifically we show how to realize MPC using any additively-homomorphic commitment scheme, even if such a scheme is an interactive two-party protocol.
Our new approach enables us to do arithmetic computation over arbitrary finite fields. In addition, since our protocol computes over committed values, it can be readily composed within larger protocols, and can also be used for efficiently implementing committing OT or committed OT. This is done in two steps, each of independent interest:

1. Black-box extension of any (possibly interactive) two-party additively homomorphic commitment scheme to an additively homomorphic multiparty commitment scheme, only using coin-tossing and a "weak" equality evaluation functionality.
2. Realizing multiplication of multiparty commitments based on a lightweight preprocessing approach.

Finally we show how to use the fully homomorphic commitments to compute any functionality securely in the presence of a malicious adversary corrupting any number of parties.

1 Introduction

Secure computation (MPC) is the area of cryptography concerned with mutually distrusting parties who wish to compute some function f on private input from each of the parties, yielding some private output to each of the parties. If we consider p parties, $P_1, \ldots, P_p$ where party P_i has input x_i the parties then wish to learn their respective output y_i. We can thus describe the function to compute as $f(x_1, x_2, \ldots, x_p) = (y_1, y_2, \ldots, y_p)$. It was shown in the 80's how to realize this, even against a malicious adversary taking control over a majority of the parties [23]. With feasibility in place, much research has been carried out trying to make MPC as efficient as possible. One specific approach to efficient MPC, which has gained a lot of traction is based on secret sharing [23,5,3]: Each party secretly shares his or her input with the other parties. The parties then parse f as an arithmetic circuit, consisting of multiplication and addition gates. In a collaborative manner, based on the shares, they then compute the circuit, to achieve shares of the output which they can then open.

1.1 Our Contributions

Using the secret sharing approach opens up the possibility of malicious parties using "inconsistent" shares in the collaborative computation. To combat this, most protocols add a MAC on the true value shared between the parties. If someone cheats it is then possible to detect this when verifying the MAC [14,16,32].

In this paper we take a different approach to ensure correctness: We have each party commit to its shares towards the other parties using an additively homomorphic commitment. We then have the collaborative computation take place on the commitments instead of the pure shares. Thus, if some party tries to change its share during the protocol then the other parties will notice when the commitments are opened in the end (as the opening will be invalid).

By taking this path, we can present the following contributions:

⋆ Majority of work done while at Bar-Ilan University, Israel.

⋆⋆ All authors were supported by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minsters Office. Tore has also received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731583.

1. An efficient and black-box reduction from random multiparty homomorphic commitments, to two-party additively homomorphic commitments.
2. Using these multiparty commitments we present a new secret-sharing based MPC protocol with security against a majority of malicious adversaries. Leveraging the commitments, our approach does not use any MAC scheme and does not rely on a random oracle or any specific number theoretic assumptions.
3. The new protocol has several advantages over previous protocols in the same model. In particular our protocol works over fields of arbitrary characteristic, independent of the security parameter. Furthermore, since our protocol computes over committed values it can easily be composed inside larger protocols. For example, it can be used for computing committed OT in a very natural and efficient way.
4. We suggest an efficient realization of our protocol, which only relies on a PRG, coin-tossing and OT[3]. We give a detailed comparison of our scheme with other dishonest majority, secret-sharing based MPC schemes, showing that the efficiency of our scheme is comparable, and in several cases preferable, over state-of-the-art.

1.2 High Level Idea

We depart from any (possibly interactive) two-party additively homomorphic commitment scheme. To achieve the most efficient result, without relying on a random oracle or specific number theoretic assumptions, we consider the scheme of [18], since has been shown to be highly efficient in practice [34,35]. This scheme, along with others [11,10,9] works on commitments to vectors of m elements over some field $\mathbb{F}$. For this reason we also present our results in this setting. Thus any of these schemes could be used.

The first part of our protocol constructs a large batch of commitments to random values. The actual value in such a commitment is unknown to any party, instead, each party holds an additive share of it. This is done by having each party pick a random message and commit to it towards every other party, using the two-party additively homomorphic commitment scheme. The resulted multiparty commitment is the sum of all the messages the parties committed to, which is uniformly random if there is at least one honest party. We must ensure that a party commits to the same message towards all other parties, to this end the parties agree on a few (random) linear combinations over the commitments, which are then opened and being checked.

Based on these random additively shared commitments, the parties execute a preprocessing stage to construct random multiplication triples. This is done in a manner similar to MASCOT [28], yet a bit different, since our scheme supports computation over arbitrary small fields and MASCOT requires a field of size exponential in the security parameter. More specifically the Gilboa protocol [22] for multiplication of additively shared values is used to compute the product of two shares of the commitments between each pair of parties. However, this is not maliciously secure as the result might be incorrect and a few bits of information on the honest parties' shares might be leaked. To ensure correctness cut-and-choose and sacrificing steps are executed. First, a few triples are opened and checked for correctness. This ensures that not all triples are incorrectly constructed. Next, the remaining triples are mapped into buckets, where some triples are sacrificed to check correctness of another triple. At this point all the triples are correct except with negligible probability. Finally, since the above process grants the adversary the ability to leak some bits from the honest parties shares, the parties engage in a combining step, where triples are randomly "added" together to ensure that the result will contain at least one fully random triple.

As the underlying two-party commitments are for *vectors* of messages, our protocol immediately features single-instruction multiple-data (SIMD), which allows multiple simultaneously executions of the same computation (over different inputs). However, when performing only a single execution we would like to use only one element out of the vector and save the rest of the elements for a later use. We do so by preprocessing reorganization pairs, following the same approach presented in MiniMAC [16,12,15], which allows to perform a linear transformation over a committed vector.

With the preprocessing done, the online phase of our protocol proceeds like previous secret-sharing based MPC schemes such as [14,28,16]. That is, the parties use their share of the random commitments to give input to the protocol. Addition is then carried out locally and the random multiplication triples are used to interactively realize multiplication gates.

[3] OT can be efficiently realized using an OT extension, without the usage of a random oracle, but rather a type of correlation robustness, as described in [2].

Efficiency. In table Table 1 we count the amount of OTs, two-party commitments and coin tossing operations required in the different commands of our protocol (specifically, in the **Rand**, **Input**, **ReOrg**, **Add** and **Mult** commands).

The complexities describe what is needed to construct a vector of m elements in the underlying field in the amortized sense. When using the commitment scheme of [18] it must hold that $m \geq s/\lceil \log_2(|\mathbb{F}|) \rceil$ where s is the statistical security parameter.

	Rand, Input	ReOrg	Add	Mult
OTs	0	0	0	$27m \log(\lvert\mathbb{F}\rvert) p(p-1)$
Two-party Commitments	$p(p-1)$	$3p(p-1)$	0	$81p(p-1)$
Random coins	$\log(\lvert\mathbb{F}\rvert)$	$4\log(\lvert\mathbb{F}\rvert)$	0	$108\log(\lvert\mathbb{F}\rvert)$

Table 1. Amortized complexity of each instruction of our protocol (Rand,Input,ReOrg,Add and Mult), when constructing a batch of 2^{20} multiplication triples, *each* with m independent components among p parties. The quadratic complexity of the number of two-party commitments reflects the fact that our protocol is constructed from any two-party commitment scheme in a black-box manner, and so each party independently commits to all other party for every share it posses.

1.3 Related Work

Comparison to SPDZ and TinyOT. In general having the parties commit to their shares allows us to construct a secret-sharing based MPC protocol ala SPDZ [14,28], but without the need of shared amd specific information theoretic MACs. This gives us several advantages over the SPDZ approach:

- We get a light preprocessing stage of multiplication triples as we can base this on commitments to random values, which are later adjusted to reflect a multiplication. Since the random values are additively homomorphic and committed, this limits the adversary's possible attack vector. In particular we do not need an authentication step.
- Using the commitment scheme of [18] we get the possibility of committing to messages in any field $\mathbb{F}$ among p parties, using communication of only $O(\log(|\mathbb{F}|) \cdot p^2)$ bits, amortized. This is also the case when $\mathbb{F}$ is the binary field[4] or of different characteristic than 2. In comparison, SPDZ requires the underlying field to be of size $\Omega(2^s)$ where s is a statistical security parameter.
- The TinyOT protocol [32,29,7] on the other hand only works over GF(2) and requires a MAC of $\tilde{O}(s)$ bits on each secret bit. Giving larger overhead than in SPDZ, MiniMAC and our protocol and limiting its use-case to evaluation of boolean circuits.

Comparison to MiniMAC. The MiniMAC protocol [16] uses an error correcting code over a vector of data elements. It can be used for secure computation over small fields without adding long MACs to each data element. However, unfortunately the authors of [16] did not describe how to realize the preprocessing needed. Neither did the follow up works [12,15]. The only efficient[5] preprocessing protocol for MiniMAC that we know of is the one presented in [19] based on OT extension. However this protocols has it quirks:

- It only works over fields of characteristic 2.
- The ideal functionality described is different from the ones in [16,12,15]. Furthermore, it is non-standard in the sense that the corruption that an adversary can apply to the shares of honest parties can be based on the inputs of the honest parties.
- There is no proof that this ideal functionality works in the online phase of MiniMAC.
- There seems to be a bug in one of the steps of the preprocessing of multiplication triples. We discuss this in further detail in Appendix H.

[4] This requires a commitment to be to a vector of messages in $\mathbb{F}$.

[5] I.e. one that does not use a generic MPC protocol to do the preprocessing.

OT Extensions. All the recent realizations of the preprocessing phase on secret shared protocols such as SPDZ, MiniMAC and TinyOT are implemented using OT. The same goes for our protocol. Not too long ago this would have not been a practically efficient choice since OT generally requires public key operations. However, the seminal work of Beaver [4] showed that it was possible to extend a few OTs, using only symmetric cryptography, to achieve a practically unbounded amount of OTs. Unfortunately Beaver's protocol was not practically efficient, but much research has been carried out since then [24,32,1,2,27], culminating with a maliciously secure OT extension where a one-out-of-two OT of 128 bit messages with $s = 64$ can be done, in the amortized sense, in $0.3\mu s$ [27].

Commitment Extensions. Using additive homomorphic commitments for practical MPC is a path which would also not have been possible even just a few years ago. However, much study has undergone in the area of "commitment extension" in the recent years. All such constructions that we know of require a few OTs in a preprocessing phase and then construction and opening of commitments can be done using cheap symmetric or information theoretic primitives. The work on such extensions started in [21] and independently in [11]. A series of follow-up work [10,18,34,9,6] presented several improvements, both asymptotically and practically. Of these works [34] is of particular interest since it presents an implementation (based on the scheme of [18]) and showed that committing and opening 128 bit messages with $s = 40$ can be done in less than $0.5\mu s$ and $0.2\mu s$ respectively, in the amortized sense for a batch of 500,000 commitments. [6]

It should be noted that Damgård *et al.* [11] also achieved both additively and multiplicative homomorphic commitments. They use this to get an MPC protocol cast in the client/server setting. We take some inspiration from their work, but note that their setting and protocols are quite different from ours in that they use verifiable secret sharing to achieve the multiplicative property and so their scheme is based on threshold security, meaning they get security against a constant fraction of servers in a client/server protocol.

Relation to [13]. The protocol by Damgård and Orlandi also considers a maliciously secure secret-sharing based MPC in the dishonest majority setting. Like us, their protocol is based on additively homomorphic commitments where each party is committed to its share to thwart malicious behavior. However, unlike ours, their protocol only works over large arithmetic fields and uses a very different approach. Specifically they use the cut-and-choose paradigm along with packed secret sharing in order to construct multiplication triples. Furthermore, to get random commitments in the multiparty setting, they require usage of public-key encryption for each commitment. Thus, the amount of public-key operations they require is linear in the amount of multiplication gates in the circuit to compute. In our protocol it is possible to limit the amount of public-key operations to be asymptotic in the security parameter, as we only require public-key primitives to bootstrap the OT extension.

Other Approaches to MPC. Other approaches to maliciously secure MPC in the dishonest majority setting exist. For example Yao's garbled circuit [36,30,31], where the parties first construct an encrypted Boolean circuit and then evaluate it locally. Another approach is "MPC-in-the-head" [25,26] which efficiently combines any protocol in the malicious honest majority settings and any protocol in the semi-honest dishonest majority settings into a protocol secure in the malicious dishonest majority settings.

1.4 Paper Outline

We start with some preliminaries in Section 2 where we define our notation, variables names and ideal functionalities. We continue in Section 3 with a description of how to achieve a multiparty additively homomorphic commitment scheme from any (possibly interactive) two-party homomorphic commitment scheme. In Section 4 we describe how to use the multiparty commitment scheme to preprocess multiplication triples and in general realize an offline phase for a secret sharing based MPC protocol. Afterwards, in Section 5 we describe how to realize such an MPC scheme. We compare the efficiency of our protocol to previous constructions in Section Section 6 and finally we consider possible applications based on our protocol in Section 7.

[6] Note that this specific implementation unfortunately uses a code which does not have the properties our scheme require. Specifically its product-code has too low minimum distance.

2 Preliminaries

2.1 Parameters and Notation

Throughout the paper we use "negligible probability in s" to refer to a probability $o(1/poly(s))$ where $poly(s)$ indicates some polynomial in $s \in \mathbb{N}$. Similarly we use "overwhelming probability in s" to denote a probability $1 - o(1/poly(s))$, where s is the statistical security parameter.

There are $p \in \mathbb{N}$ parties $P_1, \ldots, P_p$ participating in the protocol. The notation $[k]$ refers to the set $\{1, \ldots, k\}$. We let vector variables be expressed with **bold** face. We use square brackets to select a specific element of a vector, that is, $\mathbf{x}[\ell] \in \mathbb{F}$ is the ℓ'th element of the vector $\mathbf{x} \in \mathbb{F}^m$ for some $m \geq \ell$. We assume that vectors are column vectors and use $\|$ to denote concatenation of rows, that is, $\mathbf{x}\|\mathbf{y}$ with $\mathbf{x}, \mathbf{y} \in \mathbb{F}^m$ is a $m \times 2$ matrix. We use $* : \mathbb{F}^m \times \mathbb{F}^m \to \mathbb{F}^m$ to denote component-wise multiplication and $\cdot : \mathbb{F} \times \mathbb{F}^m \to \mathbb{F}^m$ to denote a scalar multiplication. We will sometimes abuse notation slightly and consider $\mathbb{F}$ as a set of elements and thus use $\mathbb{F}\backslash\{0\}$ to denote the elements of $\mathbb{F}$, excluding the additive neutral element 0.

If S is a set we assume that there exists an arbitrary, but globally known deterministic ordering in such a set and let $S[i] = S_i$ denote the ith element under such an ordering. In general we always assume that sets are stored as a list under such an ordering. When needed we use $(a, b, ...)$ to denote a list of elements in a specific order. Letting A and B be two sets s.t. $|A| = |B|$ we then abuse notation by letting $\{(a, b)\} \in (A, B)$ denote $\{(A[i], B[i])\}_{i\in[|A|]}$. I.e. a and b denote the i'th element in A, respectively B.

All proof and descriptions of protocols are done using the Universally Composable framework [8].

2.2 Ideal Functionalities

We list the ideal UC-functionalities we need for our protocol. Note that we use the standard functionalities for Coin Tossing, Secure Equality Check, Oblivious Transfer and Multiparty Computation.

We need a coin-tossing functionality that allows all parties to agree on uniformly random elements in a field. For this purpose we describe a general, maliciously secure coin-tossing functionality in Fig. 1.

Functionality $\mathcal{F}_{\mathsf{CT}}$: Interacts with $P_1, \ldots, P_p$ and an adversary $\mathcal{A}$.

- Upon receiving $(\mathtt{toss}, n, \mathbb{F})$ from all parties, where $\mathbb{F}$ is a description of some field $\mathbb{F}$ and n an integer, leak $(\mathtt{toss}, n, \mathbb{F})$ to $\mathcal{A}$. Then sample n uniformly random elements $x_1, \ldots, x_n \in_R \mathbb{F}$ and send $(\mathtt{random}, x_1, \ldots, x_n)$ to $\mathcal{A}$.
- If $\mathcal{A}$ returns the message $(\mathtt{deliver})$ then send the message delivered to $\mathcal{A}$ to all parties, otherwise if $\mathcal{A}$ returns the message $(\mathtt{abort})$ then output $\mathtt{abort}$ to all parties.

Fig. 1. Ideal Functionality $\mathcal{F}_{\mathsf{CT}}$

Furthermore we need to be able to securely evaluate equality of values. This functionality is described in Fig. 2. Notice that this functionality allows the adversary to learn the honest parties' inputs *after* it supplies its own. Furthermore, we allow the adversary to learn the result of the equality check before any honest parties, which gives him the chance to abort the protocol. Thus this function should only be used on data that is not private. The functionality can for example be implemented using a commitment scheme where each party commits to its input towards every other party. Once all parties have committed, the parties open the commitments and each party locally evaluates if everything is equal.

We also require a standard 1-out-of-2 functionality denoted by $\mathcal{F}_{\mathsf{OT}}$ as described in Fig. 3.

Finally, a fully fledged MPC functionality, very similar to the one described in previous works such as SPDZ and MiniMAC, is described in Fig. 4. Note that the functionality maintains the dictionary id that maps indices to values stored by the functionality. The expression $\mathsf{id}[k] = \perp$ means that no value is stored by the functionality at index k in that dictionary. Also note that the functionality is described as operating over *vectors* from $\mathbb{F}^m$ rather than over elements from $\mathbb{F}$. This is because our protocol allows up to m simultaneous secure computations of the same function (on different inputs) at the price of a single computation, thus, every operation (such as input, random, add, multiply)

Functionality $\mathcal{F}_{\mathsf{EQ}}$: Interacts with $P_1, \ldots, P_p$ and an adversary $\mathcal{A}$. It proceeds as follows:

Equality: Upon receiving $\left(\texttt{equal}, i, \mathbf{x}^i\right)$ from party P_i for all $i \in [p]$ where $\mathbf{x}^i \in \mathbb{F}^m$ (if P_i is corrupted then $\mathbf{x}^i$ is selected by $\mathcal{A}$), proceed as follows: If $\mathbf{x}^1 = \mathbf{x}^2 = \ldots = \mathbf{x}^p$ then send $(\texttt{equal}, \texttt{accept})$ to $\mathcal{A}$, otherwise send $(\texttt{equal}, \mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^p, \texttt{reject})$ to $\mathcal{A}$. Proceed as follows:

- Upon receiving $\left(\texttt{equal}, i, \mathbf{x}^i\right)$ from party P_i for all $i \in p$ (if P_i is corrupted then $\mathbf{x}^i$ is selected by $\mathcal{A}$) if $\mathbf{x}^1 = \mathbf{x}^2 = \ldots = \mathbf{x}^p$ then send $(\texttt{equal}, \texttt{accept})$ to $\mathcal{A}$, otherwise send $(\texttt{equal}, \mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^p, \texttt{reject})$ to $\mathcal{A}$.
- If $\mathcal{A}$ returns $(\texttt{deliver})$ and $\mathbf{x}^1 = \mathbf{x}^2 = \ldots = \mathbf{x}^p$ then send the message $(\texttt{equal}, \texttt{accept})$ to all parties. If instead $\mathbf{x}^i \neq \mathbf{x}^j$ for some $i, j \in [p]$, then send $(\texttt{equal}, \mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^p, \texttt{reject})$ to all parties.
- If $\mathcal{A}$ instead returns $(\texttt{abort})$ then output $\texttt{abort}$ to all parties.

Fig. 2. Ideal Functionality $\mathcal{F}_{\mathsf{EQ}}$

Functionality $\mathcal{F}_{\mathsf{OT}}$: Interacts with a sender P_i, a receiver P_j and an adversary $\mathcal{A}$ and proceeds as follows:

Sender Input: Upon receiving $(\texttt{transfer}, x_0, x_1)$ from P_i where $x_0, x_1 \in \{0,1\}^*$ leak $(\texttt{transfer})$ to $\mathcal{A}$.
Receiver Input: Upon receiving $(\texttt{receive}, b)$ from P_j where $b \in \{0,1\}$ leak $(\texttt{receive})$ to $\mathcal{A}$. If a message of the form $(\texttt{transfer}, x_0, x_1)$ has been received from P_i then output $(\texttt{deliver}, x_b)$ to P_j and $(\texttt{deliver}, \perp)$ to P_i.

Fig. 3. Ideal Functionality $\mathcal{F}_{\mathsf{OT}}$

are done in a component wise manner to a vector from $\mathbb{F}^m$. As we describe later, it is indeed possible to perform a single secure computation when needed.

Functionality $\mathcal{F}_{\mathsf{MPC}\text{-}\mathbb{F}^m}$: Interacts with $P_1, \ldots, P_p$ and an adversary $\mathcal{A}$.

Init: Upon receiving $(\texttt{init})$ from all parties forward this message to $\mathcal{A}$. Initialize an empty dictionary id.
Input: Upon receiving $(\texttt{Input}, i, k, \mathbf{x})$ from P_i where $\mathbf{x} \in \mathbb{F}^m$ and $(\texttt{Input}, i, k)$ from all other parties, set $\mathsf{id}[k] = \mathbf{x}$ and output $(\texttt{Input}, i, k)$ to all parties and $\mathcal{A}$.
Rand: Upon receiving $(\texttt{random}, k)$ from all parties, pick a random $\mathbf{x} \in \mathbb{F}^m$ and set $\mathsf{id}[k] = \mathbf{x}$. Output $(\texttt{random}, k)$ to all parties and $\mathcal{A}$.
Add: Upon receiving $(\texttt{add}, x, y, z)$ from all parties where $\mathsf{id}[x], \mathsf{id}[y] \neq \perp$, set $\mathsf{id}[z] = \mathsf{id}[x] + \mathsf{id}[y]$ and output $(\texttt{add}, x, y, z)$ to all parties and $\mathcal{A}$.
Public Add: Upon receiving $(\texttt{add}, \mathbf{x}, y, z)$ from all parties where $\mathbf{x} \in \mathbb{F}^m$ and $\mathsf{id}[y] \neq \perp$, set $\mathsf{id}[z] = \mathbf{x} + \mathsf{id}[y]$ and output $(\texttt{add}, \mathbf{x}, y, z)$.
Multiply: Upon receiving $(\texttt{mult}, x, y, z)$ from all parties where $\mathsf{id}[x], \mathsf{id}[y] \neq \perp$, set $\mathsf{id}[z] = \mathsf{id}[x] * \mathsf{id}[y]$ and output $(\texttt{mult}, x, y, z)$ to all parties and $\mathcal{A}$.
Public Multiply: Upon receiving $(\texttt{mult}, \mathbf{x}, y, z)$ from all parties where $\mathbf{x} \in \mathbb{F}^m$ and $\mathsf{id}[y] \neq \perp$, set $\mathsf{id}[z] = \mathbf{x} * \mathsf{id}[y]$ and output $(\texttt{mult}, x, y, z)$ to all parties and $\mathcal{A}$.
Output: Upon receiving $(\texttt{Output}, k)$ from all parties where $\mathsf{id}[k] \neq \perp$ then output $(k, \mathsf{id}[k])$ to $\mathcal{A}$. If $\mathcal{A}$ returns $(\texttt{deliver})$ then output $(k, \mathsf{id}[k])$ to all parties, otherwise, if $\mathcal{A}$ returns $(\texttt{abort})$ then output $\texttt{abort}$ to all parties.

Fig. 4. Ideal Functionality $\mathcal{F}_{\mathsf{MPC}\text{-}\mathbb{F}^m}$

Dependencies between functionalities and protocols. We illustrate the dependencies between the ideal functionalities just presented and our protocols in Fig. 5. We see that $\mathcal{F}_{\mathsf{CT}}$ and $\mathcal{F}_{\mathsf{EQ}}$, along with a two-party commitments scheme, $\mathcal{F}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$ (presented in the next section) are used to realize our multiparty commitment scheme in protocol $\Pi_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$. Functionalities $\mathcal{F}_{\mathsf{CT}}$ and $\mathcal{F}_{\mathsf{EQ}}$ are again used, along with $\mathcal{F}_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$ and $\mathcal{F}_{\mathsf{OT}}$ to realize the augmented homomorphic commitment scheme $\Pi_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$. $\Pi_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ constructs all the preprocessed material, in particular multiplication triples, needed in order to realize the fully fledged MPC protocol $\Pi_{\mathsf{MPC}\text{-}\mathbb{F}^m}$.

2.3 Arithmetic Oblivious Transfer

Generally speaking, arithmetic oblivious transfer allows two parties P_i and P_j to obtain an additive shares of the multiplication of two elements $x, y \in \mathbb{F}$, where P_i privately holds x and P_j privately holds y.

A protocol for achieving this in the semi-honest settings is presented in [22] and used in MASCOT [28]. Let $\ell = \lceil \log \mathbb{F} \rceil$ be the number of bits required to represent elements from the field $\mathbb{F}$, then the protocol goes by having the parties run in ℓ (possibly parallel) rounds, each of which invokes an instance of the general oblivious transfer functionality ($\mathcal{F}_{\mathsf{OT}}$). This is described by procedure ArithmeticOT in Fig. 6.

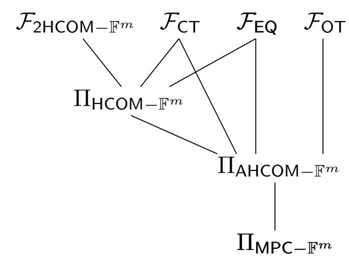

Fig. 5. Outline of functionalities and protocols.

Procedure ArithmeticOT(x, y):

For $q = 1$ to $\ell = \lceil \log \mathbb{F} \rceil$, the parties P_i and P_j do as follows:

1. Party P_j (the sender) chooses a uniformly random $r_q \in \mathbb{F}$ and set the two ℓ-bit strings s_q^0, s_q^1 where $s_q^0 = r_q$ and $s_q^1 = y + r_q$.
2. Party P_j invokes $\mathcal{F}_{\mathsf{OT}}$ with the message $(\texttt{transfer}, s_q^0, s_q^1)$.
3. Party P_i (as the receiver) invokes $\mathcal{F}_{\mathsf{OT}}$ with the bit $x_q \in \{0, 1\}$ with the message $(\texttt{receive}, x_q)$ (note that x_q corresponds to the bit decomposition of x).
4. $\mathcal{F}_{\mathsf{OT}}$ returns $(\texttt{deliver}, s_q^{x_q})$ to P_i and $(\texttt{deliver}, \perp)$ to P_j.

Finally, party P_i outputs $z^i = \sum_{q \in [\ell]} s_q^{x_q} \cdot 2^{q-1}$ and P_j outputs $z^j = \sum_{q \in [\ell]} -r_q \cdot 2^{q-1}$.

Fig. 6. Procedure ArithmeticOT

Correctness. Note that P_i outputs

$$\begin{aligned} z^i &= \sum_{q \in [\ell]} s_q^{x_q} \cdot 2^{q-1} = \sum_{q \in [\ell]} (x_q \cdot y + r_q) \cdot 2^{q-1} = \sum_{q \in [\ell]} x_q \cdot y \cdot 2^{q-1} + \sum_{q \in [\ell]} r_q \cdot 2^{q-1} \\ &= x \cdot y + \sum_{q \in [\ell]} r_q \cdot 2^{q-1} = x \cdot y - z^j \end{aligned}$$

and thus we have $z^i + z^j = x \cdot y$. The second equality holds because $s_q^{x_q}$ equals $y + r_q$ if $x_q = 1$ and equals r_q if $x_q = 0$.

The use of arithmetic OT to construct multiplication triples. In our protocol we use the above procedure to multiply two elements $\mathbf{x}, \mathbf{y} \in \mathbb{F}^m$ such that one party privately holds $\mathbf{x}$ and the other party privately holds $\mathbf{y}$. Specifically, we can do this using m invocations of the ArithmeticOT procedure, thus, to multiply elements from $\mathbb{F}^m$ we make a total of $m \log(\lceil |\mathbb{F}| \rceil)$ calls to the `transfer` command of the $\mathcal{F}_{\mathsf{CT}}$ functionality.

Malicious behavior. In the above procedure party the sending party, P_j, may guess bits of x in the following manner: To guess that the q'th bit is 1 (i.e. $x_q = 1$) P_j calls $(\texttt{transfer}, s_q^0, s_q^1)$ with $s_q^0 = 0$ (rather than $s_q^0 = r_q$) and $s_q^1 = y + r_q$ (as required). Then, if $x_q = 0$ then P_i adds $s_q^0 \cdot 2^{q-1} = 0$ when computing z^i, while P_j decreases $r_q \cdot 2^{q-1}$ as required. On the other hand, if $x_q = 1$ then P_i adds $s_1 \cdot 2^{q-1} = (y + r_q) \cdot 2^{q-1}$ when computing z^i and P_j decreases $r_q \cdot 2^{q-1}$ as required. Now, notice that if $x_q = 0$ then the results of the procedure are z^i and z^j such that $z^i + z^j \neq xy$ while if $x_q = 1$ then $z^i + z^j = xy$. Thus, if z^i and z^j are used later on in the protocol then P_j may learn x_q by inspecting if the protocol aborts or not. If it aborts before the parties decided their inputs then nothing is learned by P_j, however, if the protocol aborts afterwards then this reveals x_q to P_j. Furthermore, it is also possible for the sender P_j to input "incorrect" value for both s_q^0 and s_q^1 such that the receiver P_i ends up with specific and incorrect result.

Note that the receiving party, P_i, may mount a similar attack as well trying to learn y: The sender sets $s_q^0 = r_q$ and $s_q^1 = r_q + y^j$ and let the receiver's q-th bit be x_q^i. Now, if the receiver inputs $1 - x_q^i$ (instead of x_q^i) to the q-th OT then the output of the arithmetic OT would be correct iff $y^j = 0$. That is, the sender may guess whether $y^j = 0$ or not and can also know that its guess was correct if the protocol does not abort when the values are used later on.

Notice that the sender's and receiver's attacks are quite different: The sender may guess the value of each *bit* of the receiver and guesses correctly with probability $1/2$ for every guess while the sender may guess that the sender's value is zero and may succeed with probability $1/|\mathbb{F}|$ (since the shares are uniformly random).

We treat these malicious behaviors in the protocol, specifically, in the *combining* step in Section 4.2.

3 Homomorphic Commitments

In this section we present the functionalities for two-party and multiparty homomorphic commitment schemes, however, we present a realization only to the multiparty case since it uses a two-party homomorphic commitment scheme in a black-box manner and so it is not bound to any specific realization.

For completeness and concreteness of the efficiency analysis we do present a realization to the two-party homomorphic commitment scheme in Appendix A.

3.1 Two-Party Homomorphic Commitments

Functionality $\mathcal{F}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$ is held between two parties P_i and P_j, in which P_i commits to some value $\mathbf{x} \in \mathbb{F}^m$ toward party P_j, who eventually holds the commitment information, denoted $[\mathbf{x}]^{i,j}$. In addition, by committing to some value $\mathbf{x}$ party P_i holds the opening information, denoted $\langle\mathbf{x}\rangle^{i,j}$, such that having P_i send $\langle\mathbf{x}\rangle^{i,j}$ to P_j is equivalent to issuing the command **Open** on $\mathbf{x}$ by which P_j learns $\mathbf{x}$.

The functionality works in a batch manner, that is, P_i commits to γ (random) values at once using the **Commit** command. These γ random values are considered as "raw-commitments" since they have not been processes yet. The sender turns the commitment from "raw" to "actual" by issuing either **Input** or **Rand** commands on it: The **Input** command modifies the committed value to the sender's choice and the **Rand** command keeps the same value of the commitment (which is random). In both cases the commitment is considered as a "actual" and is not "raw" anymore. Actual commitments can then be combined using the **Linear Combination** command to construct a new actual-commitment.

To keep track of the commitments the functionality uses two dictionaries: raw and actual. Both map from identifiers to committed values such that the mapping returns $\perp$ if no mapping exists for the identifier. We stress that a commitment is either raw or actual, but not both. That means that either raw or $\mathtt{actual}$, or both return $\perp$ for every identifier. To issue the **Commit** command, the committer is instructed to choose a set I of γ freshly new identifiers, this is simply a set of identifiers I such that for every $k \in I$ raw and actual return $\perp$. The functionality is formally described in Fig. 7.

Functionality $\mathcal{F}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$: Interacts with two parties P_i and P_j and the adversary $\mathcal{A}$.

Init: Upon receiving $(\mathtt{init})$ from both parties set $\mathsf{raw} = \mathsf{actual} = \emptyset$ and forward the message to $\mathcal{A}$.

Commit: Upon receiving $(\mathtt{commit}, I)$ from P_i where I is a set of γ freshly new identifiers, send the message $(\mathtt{commit}, I)$ to $\mathcal{A}$. If $\mathcal{A}$ sends back $(\mathtt{no\text{-}corrupt})$ proceed as follows: For each $k \in I$ sample $\mathbf{x}_k \in_R \mathbb{F}^m$ and store $\mathsf{raw}[k] = \mathbf{x}_k$. Finally send $(\mathtt{committed}, \{(k, \mathbf{x}_k)\}_{k\in I})$ to P_i and $(\mathtt{committed}, I)$ to P_j and $\mathcal{A}$. If P_i is corrupted and $\mathcal{A}$ instead sends back $(\mathtt{corrupt}, \{(k, \bar{\mathbf{x}}_k)\}_{k\in I})$ proceed as above, but instead of sampling the values at random, use the values $\{(k, \bar{\mathbf{x}}_k)\}_{k\in I}$.

Input: Upon receiving a message $(\mathtt{Input}, k, \mathbf{y})$ from P_i if $\mathsf{raw}[k] \neq \perp$ then store $\mathsf{raw}[k] = \perp$ and $\mathsf{actual}[k] = \mathbf{y}$. Then send $(\mathtt{Input}, k)$ to P_j and $\mathcal{A}$.

Rand: Upon receiving a message $(\mathtt{random}, k)$ from P_i if $\mathsf{raw}[k] = \mathbf{x}_k \neq \perp$ then store $\mathsf{raw}[k] = \perp$ and $\mathsf{actual}[k] = \mathbf{x}_k$. Then send $(\mathtt{random}, k)$ to P_j and $\mathcal{A}$.

Linear Combination: Upon receiving $(\mathtt{linear}, (\{(k, \boldsymbol{\alpha}_k)\}_{k\in K}, \boldsymbol{\beta}, k')$ for $\boldsymbol{\alpha}_k, \boldsymbol{\beta} \in \mathbb{F}^m$ from P_i if $\mathsf{actual}[k] = \mathbf{x}_k \neq \perp$ for every $k \in K$ and $\mathtt{raw}[k'] = \mathsf{actual}[k'] = \perp$ then store $\mathsf{actual}[k'] = \boldsymbol{\beta} + \sum_{k\in K} \boldsymbol{\alpha}_k * \mathbf{x}_k$, and forward the message to P_j and $\mathcal{A}$.

Open: Upon receiving a message $(\mathtt{open}, k)$ from P_i, if $\mathsf{actual}[k] = \mathbf{x}_k \neq \perp$ then send $(\mathtt{opened}, \mathbf{x}_k)$ to P_j and $\mathcal{A}$.

Fig. 7. Ideal Functionality $\mathcal{F}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$

To simplify readability of our protocol we may use shorthands to the functionality's commands invocations as follows: Let $[\mathbf{x}_k]^{i,j}$ and $[\mathbf{x}_{k'}]^{i,j}$ be two actual-commitments issued by party P_i toward party P_j (i.e. the committed values are stored in $\texttt{actual}[k]$ and $\texttt{actual}[k']$ respectively). The **Linear Combination** command of Fig. 7 allows to compute the following operations which will be used in our protocol. The operations are defined over $[\mathbf{x}_k]^{i,j}$ and $[\mathbf{x}_{k'}]^{i,j}$ and result with the actual-commitment $[\mathbf{x}_{k''}]^{i,j}$:

- **Addition.** (Equivalent to the command $(\texttt{linear}, \{(k, \mathbf{1}), (k', \mathbf{1})\}, \mathbf{0}, k'')$.)

$$[\mathbf{x}_k]^{i,j} + [\mathbf{x}_{k'}]^{i,j} = [\mathbf{x}_k + \mathbf{x}_{k'}]^{i,j} = [\mathbf{x}_{k''}]^{i,j} \quad \text{and} \quad \langle \mathbf{x}_k \rangle^{i,j} + \langle \mathbf{x}_{k'} \rangle^{i,j} = \langle \mathbf{x}_k + \mathbf{x}_{k'} \rangle^{i,j} = \langle \mathbf{x}_{k''} \rangle^{i,j}$$

- **Constant Addition.** (Equivalent to the command $(\texttt{linear}, \{(k, \mathbf{1})\}, \boldsymbol{\beta}, k'')$.)

$$\boldsymbol{\beta} + [\mathbf{x}_k]^{i,j} = [\boldsymbol{\beta} + \mathbf{x}_k]^{i,j} = [\mathbf{x}_{k''}]^{i,j} \quad \text{and} \quad \boldsymbol{\beta} + \langle \mathbf{x}_k \rangle^{i,j} = \langle \boldsymbol{\beta} + \mathbf{x}_k \rangle^{i,j} = \langle \mathbf{x}_{k''} \rangle^{i,j}$$

- **Constant Multiplication.** (Equivalent to the command $(\texttt{linear}, \{(k, \boldsymbol{\alpha})\}, \mathbf{0}, k'')$.)

$$\boldsymbol{\alpha} * [\mathbf{x}_k]^{i,j} = [\boldsymbol{\alpha} * \mathbf{x}_k]^{i,j} = [\mathbf{x}_{k''}]^{i,j} \quad \text{and} \quad \boldsymbol{\alpha} * \langle \mathbf{x}_k \rangle^{i,j} = \langle \boldsymbol{\alpha} * \mathbf{x}_k \rangle^{i,j} = \langle \mathbf{x}_{k''} \rangle^{i,j}$$

Realization of these operations depends on the underlying two-party commitment scheme. In Appendix A we describe how addition of commitments and scalar multiplication are supported with the scheme of [18], where we also show how to extend this to enable a componentwise multiplication of an actual-commitment with a public vector from $\mathbb{F}^m$ as well (this is delayed to the appendix as it follows the same approach used in MiniMAC [16]). In the following we assume that public vector componentwise multiplication is supported in the two-party scheme.

3.2 Multiparty Homomorphic Commitments

Functionality $\mathcal{F}_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$, presented in Fig. 8, is a generalization of $\mathcal{F}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$ to the multiparty setting where the commands **Init, Commit, Input, Rand, Open** and **Linear Combination** have the same purpose as before. The additional command **Partial Open** allows the parties to open a commitment to a single party only (in contrast to **Open** that opens a commitment to *all* parties). As before, the functionality maintains the dictionaries raw and actual to keep track on the raw and actual commitments. The major change in the multiparty setting is that *all* parties take the role of both the committer and receiver (i.e. P_i and P_j from the two-party setting). For every commitment stored by the functionality (either raw or actual), both the commitment information and the opening information are secret shared between $P_1, \ldots, P_p$ using a full-threshold secret sharing scheme.

Functionality $\mathcal{F}_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$: Interacts with parties $P_1, \ldots, P_p$ and an adversary $\mathcal{A}$, who may cause the functionality to abort at any time:

Init: Upon receiving $(\texttt{init})$ from all parties, forward the message to $\mathcal{A}$ and initialize empty dictionaries raw and actual.

Commit: Upon receiving $(\texttt{commit}, I)$ where I is a set of γ freshly new identifiers, for every $k \in I$ store $\mathsf{raw}[k] = \top$ and send $(\texttt{commit}, I)$ to all parties and $\mathcal{A}$.

Input: Upon receiving a message $(\texttt{Input}, i, k, \mathbf{y})$ from P_i and $(\texttt{Input}, i, k)$ from all other parties, if $\mathsf{raw}[k] \neq \bot$ then assign $\mathsf{raw}[k] = \bot$, assign $\mathsf{actual}[k] = \mathbf{y}$ and send $(\texttt{Input}, i, k)$ to all parties and $\mathcal{A}$.

Rand: Upon receiving a message $(\texttt{random}, k)$ from all parties, if $\mathsf{raw}[k] \neq \bot$ then pick a random $\mathbf{x}_k \in_R \mathbb{F}^m$, assign $\mathsf{raw}[k] = \mathbf{x}_k$ and send $(\texttt{random}, k)$ to all parties and $\mathcal{A}$.

Linear Combination: Upon receiving a message $(\texttt{linear}, \{(k, \boldsymbol{\alpha}_k)\}_{k \in K}, \boldsymbol{\beta}, k')$ for $\boldsymbol{\alpha}_k, \boldsymbol{\beta} \in \mathbb{F}^m$ from all parties, if $\mathsf{actual}[k] = \mathbf{x}_k \neq \bot$ for all $k \in K$ and $\texttt{raw}[k'] = \texttt{actual}[k'] = \bot$ then store $\mathsf{actual}[k'] = \boldsymbol{\beta} + \sum_{k \in K} \boldsymbol{\alpha}_k * \mathbf{x}_k$ and send $(\texttt{linear}, \{(k, \boldsymbol{\alpha}_k)\}_{k \in K}, \boldsymbol{\beta}, k')$ to all parties and $\mathcal{A}$.

Open: Upon receiving a message (open, k) from all parties, if $\mathsf{actual}[k] = \mathbf{x}_k \neq \bot$ then send $(\mathsf{opened}, k, \mathbf{x}_k)$ to $\mathcal{A}$. $\mathcal{A}$ may then abort the protocol, otherwise send $(\mathsf{opened}, k, \mathbf{x}_k)$ to the honest parties.

Partial Open: Upon receiving a message (open, i, k) from all parties, if $\mathsf{actual}[k] = \mathbf{x}_k \neq \bot$ then send $(\mathsf{opened}, i, k, \mathbf{x}_k)$ to party P_i and (opened, i, k) to all other parties and $\mathcal{A}$.

Fig. 8. Ideal Functionality $\mathcal{F}_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$

3.3 Realizing $\mathcal{F}_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$ in the $(\mathcal{F}_{\mathsf{EQ}}, \mathcal{F}_{\mathsf{CT}}, \mathcal{F}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m})$-hybrid Model

Let us first fix the notation for the multiparty homomorphic commitments: We use $[\![\mathbf{x}]\!]$ to denote a (multiparty) commitment to the message $\mathbf{x}$. As mentioned above, both the message $\mathbf{x}$ and the commitment to it $[\![\mathbf{x}]\!]$ are secret shared between the parties, that is, party P_i holds $\mathbf{x}^i$ and $[\![\mathbf{x}]\!]^i$ such that $\mathbf{x} = \sum_{i\in[p]} \mathbf{x}^i$ and $[\![x]\!]^i$ is composed of the information described in the following. By issuing the **Commit** command, party P_i sends $[\mathbf{x}^i]^{i,j}$ for every $j \neq i$ (by invoking the **Commit** command from $\mathcal{F}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$). Thus, party P_i holds the opening information for all instances of the commitments to $\mathbf{x}^i$ toward all other parties, that is, it holds $\left\{\langle \mathbf{x}^i \rangle^{i,j}\right\}_{j\in[p]\smallsetminus\{i\}}$. In addition, P_i holds the commitment information received from all other parties, $\mathbf{x}^j$ (for $j \neq i$), that is, it holds $\left\{[\mathbf{x}^j]^{j,i}\right\}_{j\in[p]\smallsetminus\{i\}}$. All that information that P_i holds with regard to the value $\mathbf{x}$ is denoted by $[\![\mathbf{x}]\!]^i$, which can be seen as a share to the multiparty commitment $[\![\mathbf{x}]\!]$.

In protocol $\Pi_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$ (from Fig. 9) each party has a local copy of the raw and actual dictionaries described above, that is, party P_i maintains raw^i and actual^i. In the protocol, P_i may be required to store $[\![\mathbf{x}]\!]^i$ (i.e. its share of $[\![\mathbf{x}]\!]$) in a dictionary (either raw^i or actual^i) under some identifier k, in such case P_i actually assigns $\mathsf{raw}^i[k] = \left\{[\mathbf{x}^j]^{j,i}, \langle \mathbf{x}^i \rangle^{i,j}\right\}_{j\in[p]\smallsetminus\{i\}}$ which may also be written as $\mathsf{raw}^i[k] = [\![\mathbf{x}]\!]^i$.

In the following we explain the main techniques used to implement the instructions of functionality $\mathcal{F}_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$ (we skip the instructions that are straightforward):

Linear operations. From the linearity of the underlying two-party homomorphic commitment functionality it follows that performing linear combinations over a multiparty commitments can be done locally by every party. We describe the notation in the natural way as follows: Given multiparty commitments $[\![\mathbf{x}]\!]$ and $[\![\mathbf{y}]\!]$ and a constant public vector $\mathbf{c} \in \mathbb{F}^m$:

- **Addition.** For every party P_i:

$$\begin{aligned}
[\![\mathbf{x}]\!]^i + [\![\mathbf{y}]\!]^i &= \left\{[\mathbf{x}^j]^{j,i}, \langle \mathbf{x}^i \rangle^{i,j}\right\}_{j\in[p]\smallsetminus i} + \left\{[\mathbf{y}^j]^{j,i}, \langle \mathbf{y}^i \rangle^{i,j}\right\}_{j\in[p]\smallsetminus i} \\
&= \left\{[\mathbf{x}^j]^{j,i} + [\mathbf{y}^j]^{j,i}, \langle \mathbf{x}^i \rangle^{i,j} + \langle \mathbf{y}^i \rangle^{i,j}\right\}_{j\in[p]\smallsetminus i} \\
&= \left\{[\mathbf{x}^j + \mathbf{y}^j]^{j,i}, \langle \mathbf{x}^i + \mathbf{y}^i \rangle^{i,j}\right\}_{j\in[p]\smallsetminus i} = [\![\mathbf{x} + \mathbf{y}]\!]^i
\end{aligned}$$

- **Constant addition.** The parties obtain $[\![\boldsymbol{\beta} + \mathbf{x}]\!]$ by having P_1 perform $\mathbf{x}^i = \mathbf{x}^i + \boldsymbol{\beta}$, then, party P_1 computes:

$$\boldsymbol{\beta} + [\![\mathbf{x}]\!]^i = \boldsymbol{\beta} + \left\{[\mathbf{x}^j]^{j,i}, \langle \mathbf{x}^i \rangle^{i,j}\right\}_{j\in[2,p]} = \left\{[\mathbf{x}^j]^{j,i}, \boldsymbol{\beta} + \langle \mathbf{x}^i \rangle^{i,j}\right\}_{j\in[2,p]} = [\![\boldsymbol{\beta} + \mathbf{x}]\!]^i$$

  and all other parties P_j compute:

$$\begin{aligned}
\boldsymbol{\beta} + [\![\mathbf{x}]\!]^j &= \boldsymbol{\beta} + \left\{[\mathbf{x}^i]^{i,j}, \langle \mathbf{x}^j \rangle^{j,i}\right\}_{j\in[p]\smallsetminus j} = \left\{[\mathbf{x}^i]^{i,j}, \langle \mathbf{x}^j \rangle^{j,i}\right\}_{j\in[2,p]\smallsetminus j} \cup \left\{[\boldsymbol{\beta} + \mathbf{x}^1]^{1,j}, \langle \mathbf{x}^j \rangle^{j,1}\right\} \\
&= [\![\boldsymbol{\beta} + \mathbf{x}]\!]^j
\end{aligned}$$

- **Constant multiplication.** For every party P_i:

$$\boldsymbol{\alpha} * [\![\mathbf{x}]\!]^i = \boldsymbol{\alpha} * \left\{[\mathbf{x}^j]^{j,i}, \langle \mathbf{x}^i \rangle^{i,j}\right\}_{j\in[p]\smallsetminus i} = \left\{\boldsymbol{\alpha} * [\mathbf{x}^j]^{j,i}, \boldsymbol{\alpha} * \langle \mathbf{x}^i \rangle^{i,j}\right\}_{j\in[p]\smallsetminus i} = [\![\boldsymbol{\alpha} * \mathbf{x}]\!]^i$$

Notice that public addition is carried out by only adding the constant $\boldsymbol{\beta}$ to *one* commitment (we arbitrarily chose P_1's commitment). This is so, since the true value committed to in a multiparty commitment is additively shared between p parties. Thus, if $\boldsymbol{\beta}$ was added to each share, then what would actually be committed to would be $p \cdot \boldsymbol{\beta}$! On the other hand, for public multiplication we need to multiply the constant $\boldsymbol{\alpha}$ with *each* commitment, so that the sum of the shares will all be multiplied with $\boldsymbol{\alpha}$.

Commit. As the parties produce a batch of commitments rather than a single one at a time, assume the parties wish to produce γ commitments, each party picks $\gamma + s$ uniformly random messages from $\mathbb{F}^m$. Each party commit to each of these $\gamma + s$ messages towards each other party using different instances of the **Commit** command from $\mathcal{F}_{\text{2HCOM-}\mathbb{F}^m}$, and thus different randomness.

Note that a malicious party might use the two-party commitment scheme to commit to different messages toward different parties, which leads to an incorrect multiparty commitment. To thwart this, we have the parties execute random linear combination checks as done for batch-opening of commitments in [18]: The parties invoke the coin-tossing protocol to agree on a $s \times \gamma$ matrix, $\mathbf{R}$ with elements in $\mathbb{F}$. In the following we denote the element in the qth row of the kth column of $\mathbf{R}$ by $\mathbf{R}_{q,k}$. Every party computes s random linear combinations of the opening information that it holds toward every other party. Similarly, every party computes s combinations of the commitments that it obtained from every other party. The coefficients of the qth combination are determined by the q'th row $\mathbf{R}$ and the qth vector from the s "extra" committed messages added to the combination. That is, let the $\gamma + s$ messages committed by party P_i toward P_j be $\mathbf{x}_1^{i,j}, \ldots, \mathbf{x}_{\gamma+s}^{i,j}$ and see that the qth combination computed by P_j is $\left(\sum_{k \in \gamma} \mathbf{R}_{q,k} \cdot [\mathbf{x}_k^{i,j}]\right) + [\mathbf{x}_{\gamma+q}^{i,j}]$ and the combination computed by P_i is $\left(\sum_{k \in \gamma} \mathbf{R}_{q,k} \cdot \langle \mathbf{x}_k^{i,j} \rangle\right) + \langle \mathbf{x}_{\gamma+q}^{i,j} \rangle$. Then P_i open the result to P_j, who checks that it is correct. If P_i was honest it committed to the same values towards all parties and so $\mathbf{x}_k^i = \mathbf{x}_k^{i,j} = \mathbf{x}_k^{i,j'}$ for all $k \in [\gamma + s]$ and $j \neq j' \in [p] \setminus \{i\}$. Likewise for the other parties, so if everyone is honest they all obtain the same result from the opening of the combination. Thus, a secure equality check would be correct in this case. However, if P_i cheated, and committed to different values toward different parties than this is detected with overwhelming probability, since the parties perform s such combinations.

Input. Each party does a partial opening (see below) of a raw, unused commitment towards the party that is supposed to give input. Based on the opened message the inputting party computes a *correction value*. That is, if the raw commitment, before issuing the input command, is a shared commitment to the value $\mathbf{x}$ and the inputting party wish to input $\mathbf{y}$, then it computes the value $\epsilon = \mathbf{y} - \mathbf{x}$ and sends this value to all parties. All parties then add $[\![\mathbf{x}]\!] + \epsilon$ to the dictionary actual and remove it from the dictionary raw. Since the party giving input is the only one who knows the value $\mathbf{x}$, and it is random, this does not leak.

We prove the following theorem in Appendix B.

Theorem 3.1. *Protocol $\Pi_{\text{HCOM-}\mathbb{F}^m}$ (of Fig. 9) UC-securely realizes functionality $\mathcal{F}_{\text{HCOM-}\mathbb{F}^m}$ (of Fig. 8) in the $\mathcal{F}_{\text{2HCOM-}\mathbb{F}^m}$, $\mathcal{F}_{\text{CT}}$, and $\mathcal{F}_{\text{EQ}}$-hybrid model, against a static and malicious adversary corrupting any majority of the parties.*

4 Committed Multiparty Computation

4.1 Augmented Commitments

In the malicious, dishonest majority setting, our protocol, as other protocols, works in the offline-online model. The offline phase consists of constructing sufficiently many multiplication triples which are later used in the online phase to carry out a secure multiplications over committed, secret shared values[7]. To this end, we augment functionality $\mathcal{F}_{\text{HCOM-}\mathbb{F}^m}$ with the instruction **Mult** that uses the multiparty raw-commitments that were created by the **Commit** instruction of Fig. 8 and produces multiplication triples of the form $([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{z}]\!])$ such that $\mathbf{x} * \mathbf{y} = \mathbf{z}$. Note that a single multiplication triple is actually three multiparty commitments to values from $\mathbb{F}^m$ such that $\mathbf{z}$ is the result of a componentwise multiplication of $\mathbf{x}$ and $\mathbf{y}$. That actually means that $\mathbf{z}_q = \mathbf{x}_q \cdot \mathbf{y}_q$ for every $q \in [m]$. Hence, this features the ability to securely evaluate up to m instances of the circuit at the same cost of evaluation of a single instance (i.e. in case the parties want to evaluate some circuit m times but with different inputs each time) where all m instances are being evaluated simultaneously. If the parties wish to evaluate only $m' < m$ instances of the circuit, say $m' = 1$, they do so by using only the values stored in the first component of the vectors, while ignoring the rest of the components. However, using a multiplication triple wastes *all* components of $\mathbf{x}, \mathbf{y}$ and $\mathbf{z}$ even if the parties wish to use only their first component. To avoid such a loss we augment $\mathcal{F}_{\text{HCOM-}\mathbb{F}^m}$ with the instruction **ReOrg**. The **ReOrg** instruction preprocesses *reorganization pairs* which are used to compute a linear operator over a multiparty commitment. For example this enable the parties to "copy" the first component to another, new, multiparty commitment, such that all

[7] Typically a secure addition can be carried out locally by each party.

Protocol $\Pi_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$. Interacts between p parties.

Init: On input (init) from all parties each pair of parties P_i and P_j invoke the command (init) of functionality $\mathcal{F}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$ to initialize an instances denoted by $\mathcal{F}^{i,j}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$.

Commit: To obtain a multiparty commitment to γ random values from $\mathbb{F}^m$:

1. The parties agree on a set I' of $\gamma + s$ freshly new identifiers.
2. Every party P_i engages in $\mathcal{F}^{i,j}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$ for all $j \neq i$ by sending the message $(\mathtt{commit}, I')$ and receiving the message $\left(\mathtt{committed}, \left\{\left(k, \mathbf{x}^{i,j}_k\right)\right\}_{k\in I'}\right)$. As a result, P_j receives the message $(\mathtt{committed}, I')$ from $\mathcal{F}^{j,i}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$ for all $j \neq i$.
3. Every party P_i chooses $\mathbf{x}^i_k \in_R \mathbb{F}^m$ for every $k \in I'$. This is the value that is going to be committed from P_i toward all other parties.
4. Every party P_i engages in $\mathcal{F}^{i,j}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$ for all $j \neq i$ by sending the message $\left(\mathtt{Input}, k, \mathbf{x}^i_k\right)$ and receives back $[\![\mathbf{x}]\!]^i = \left\{[\mathbf{x}^j_k]^{j,i}, \langle \mathbf{x}^i_k\rangle^{i,j}\right\}_{j\in[p]\setminus\{i\}}$ for every $k \in I'$.
5. The parties agree on sets I and S such that $|I| = \gamma$, $|S| = s$, $I \cap S = \emptyset$ and $I \cup S = I'$.
6. The parties issue the command $(\mathtt{toss}, \mathbb{F}^{s\times\gamma})$ to functionality $\mathcal{F}_{\mathsf{CT}}$, by which they receive $(\mathtt{random}, \mathbf{R})$ where $\mathbf{R} \in \mathbb{F}^{s\times\gamma}$. We denote the element of the qth row of the kth column by $\mathbf{R}_{q,k}$.
7. For every $q \in S$ every party P_i computes $\langle \mathbf{s}^i_q\rangle^{i,j} = \langle \mathbf{x}^i_q\rangle^{i,j} + \sum_{k\in I} \mathbf{R}_{q,k} \cdot \langle \mathbf{x}^i_k\rangle^{i,j} = \langle \mathbf{x}^i_q + \sum_{k\in I} \mathbf{R}_{q,k} \cdot \mathbf{x}^i_k\rangle^{i,j}$ and sends $\langle \mathbf{s}^i_q\rangle^{i,j}$ to P_j. P_j then computes $[\mathbf{s}^i_q]^{i,j} = [\mathbf{x}^i_q]^{i,j} + \sum_{k\in I} \mathbf{R}_{q,k} \cdot [\mathbf{x}^i_k]^{i,j} = [\mathbf{x}^i_q + \sum_{k\in I} \mathbf{R}_{q,k} \cdot \mathbf{x}^i_k]^{i,j}$ and reveals $\mathbf{s}^i_q$.
8. For every $q \in I$ every party P_i computes $\mathbf{c}^i_q = \sum_{j\in[p]} \mathbf{s}^j_q$ and inputs $(\mathtt{equal}, i, \mathbf{c}^i_q)$ to $\mathcal{F}_{\mathsf{EQ}}$. If $\mathcal{F}_{\mathsf{EQ}}$ responds with $\mathtt{abort}$ or $(\mathtt{equal}, \mathbf{s}^1_q, \ldots, \mathbf{s}^p_q, \mathtt{reject})$ in any of these calls then abort, otherwise output $(\mathtt{committed}, I)$. For every $k \in I$ store $\mathsf{raw}^i[k] = [\![\mathbf{x}_k]\!]^i$.

Input: Upon input $(\mathtt{Input}, i, k, \mathbf{y})$ from party i and $(\mathtt{Input}, i, k)$ from all other parties:

1. Party P_j (for $j \neq i$) aborts if $\mathsf{raw}^j[k] = \perp$. Otherwise P_j sends $\langle \mathbf{x}^j_k\rangle^{j,i}$ to P_i (using $(\mathtt{open}, k)$), who learns $\mathbf{x}^j_k$.
2. Party P_i computes $\mathbf{x}_k = \sum_{j\in[p]} \mathbf{x}^j_k$ and broadcasts $\boldsymbol{\epsilon}_k = \mathbf{y} - \mathbf{x}_k$ to all other parties.
3. Party P_i updates $[\![\mathbf{x}_k]\!]^i$ by setting the opening values to $\langle \mathbf{x}^i + \epsilon_k\rangle^{i,j} = \langle \mathbf{x}^i_k\rangle^{i,j} + \epsilon_k$ for all $j \in [p]$. Similarly, party P_j (for $j \neq i$) updates $[\![\mathbf{x}_k]\!]^j$ by setting the ith commitment information to be $[\mathbf{x}^i_k + \epsilon_k]^{i,j} = [\mathbf{x}^i_k]^{i,j} + \epsilon_k$.
4. Party P_j (for all $j \in [p]$) assigns $\mathsf{raw}^j[k] = \perp$ and $\mathsf{actual}^j = [\![\mathbf{x}_k]\!]^j$.

Rand: The parties agree on an arbitrary k such that $\mathsf{raw}^i[k] = [\![\mathbf{x}_k]\!]^i \neq \perp$ for all $i \in [p]$, set $\mathsf{raw}^i[k] = \perp$ and $\mathsf{actual}^i[k] = [\![\mathbf{x}_k]\!]^i$.

Linear Combination: The parties agree on a set of indices K and the public vectors $\{\boldsymbol{\alpha}_k\}_{k\in K}$ such that $\mathsf{actual}[k] \neq \perp$ and $\boldsymbol{\alpha}_k \in \mathbb{F}^m$ for every $k \in K$. In addition, the parties agree on a public vector $\boldsymbol{\beta} \in \mathbb{F}^m$ and an index k' such that $\mathsf{raw}[k'] = \mathsf{actual}[k'] = \perp$. Finally, every party P_i stores $\mathsf{actual}[k'] = \boldsymbol{\beta} + \sum_{k\in K} \boldsymbol{\alpha}_k * [\![\mathbf{x}_k]\!]^i$.

Open: To open $[\![\mathbf{x}_k]\!]$, every party P_i sends $\langle \mathbf{x}^i_k\rangle^{i,j}$ to P_j for all $j \neq i$. Then, party P_i obtains $\mathbf{x}^j_k$ from the commitment and the opening information $[\mathbf{x}^j_k]^{j,i}$ and $\langle \mathbf{x}^j_k\rangle^{j,i}$ respectively. Finally P_i computes $\mathbf{x}_k = \sum_{j\in[p]} \mathbf{x}^j_k$.

Partial Open: To open $[\![\mathbf{x}_k]\!]$ to party P_i, every party P_j (for $j \neq i$) sends $\langle \mathbf{x}^j_k\rangle^{j,i}$ to P_i. Then, party P_i obtains $\mathbf{x}^j_k$ from the commitment and the opening information $[\mathbf{x}^j_k]^{j,i}$ and $\langle \mathbf{x}^j_k\rangle^{j,i}$ respectively. Finally P_i computes $\mathbf{x}_k = \sum_{j\in[p]} \mathbf{x}^j_k$.

Fig. 9. Protocol $\Pi_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$

components of the new multiparty commitment are equal to the first component of the old one. For instance, the linear operator $\phi \in \mathbb{F}^{m\times m}$ such that its first column is all 1 and all other columns are all 0, transforms the vector $\mathbf{x}$ to $\mathbf{x}' = \mathbf{x}_1, \ldots, \mathbf{x}_1$ (m times). Applying ϕ to $\mathbf{y}$ and $\mathbf{z}$ as well results in a new multiplication triple $(\mathbf{x}', \mathbf{y}', \mathbf{z}')$ where only the first component of $(\mathbf{x}, \mathbf{y}, \mathbf{z})$ got used (rather than all their m components). We note that the construction of reorganization pairs are done in a batch for *each* function ϕ resulting in the additive destruction of s extra raw commitments (i.e. an additive overhead). In the **ReOrg** command, described in Fig. 10, the linear operator ϕ is applied to L raw commitments in a batch manner. The inputs to ϕ are the messages stored by the functionality under identifiers from the set X and the outputs override the messages stored by the functionality under identifiers from the set Y. The messages stored under identifiers from the set R are being destroyed (this reflects the additive overhead of that command).

Adding instructions **Mult** and **ReOrg** to the $\mathcal{F}_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$ functionality, we get the augmented functionality $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ formally presented in Fig. 10.

Realizing $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ The protocol $\Pi_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ is formally presented in Fig. 12 and Fig. 13. In the following we describe the techniques used in $\Pi_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ and show the analysis that implies the number of multiplication triples

Functionality $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$. Interacts with p parties and an adversary $\mathcal{A}$:

Init: As in Fig. 8, in addition, initialize two additional empty dictionaries ReOrg and mult.
Commit, Input, Rand, Linear Combination, Open, Partial Open: As in Fig. 8.
Mult: Upon receiving a message $(\mathtt{mult}, K)$ from all parties where $\mathsf{raw}[k] \neq \perp$ for every $k \in K$, partition K into four sets X, Y, Z, R where $|X| = |Y| = |Z| = T$ for some T and $|R| = 3(\tau_1 + \tau_1((\tau_2)^2 - 1) \cdot T)$. (The values of τ_1 and τ_2 are explained below). For every $q = 1, \dots, T$:

1. Set $\mathsf{mult}[q] = (X_q, Y_q, Z_q)$.
2. Sample $\mathbf{x}, \mathbf{y} \in_R \mathbb{F}^m$ and set $\mathsf{actual}[X_q] = \mathbf{x}$, $\mathsf{actual}[Y_q] = \mathbf{y}$ and $\mathsf{actual}[Z_q] = \mathbf{x} * \mathbf{y}$.

Finally, for every $k \in K$ set $\mathsf{raw}[k] = \perp$. Output $\left(\mathtt{mult}, (X_q, Y_q, Z_q)_{q\in[T]}\right)$ to all parties and $\mathcal{A}$.
ReOrg: Upon receiving a message $(\mathtt{reOrg}, \phi, K)$ from all parties where $\phi \in \mathbb{F}^{m\times m}$ is a linear operator and $\mathsf{raw}[k] \neq \perp$ for every $k \in K$, do as follow: Partition K into three sets X, Y, R where $|X| = |Y| = T$ for some T and $|R| = s$. For every $q \in [T]$ set $\mathsf{ReOrg}[q] = (X_q, Y_q)$, $\mathsf{actual}[X_q] = \mathbf{x}_{X_q}$ and $\mathsf{actual}[Y_q] = \phi(\mathbf{x}_{X_q})$. Finally, for every $k \in K$ set $\mathsf{raw}[k] = \perp$. Output $\left(\mathtt{reOrg}, \{(X_q, Y_q)\}_{q\in[T]}\right)$ to all parties and $\mathcal{A}$.

Fig. 10. Ideal Functionality $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$

that should be constructed in one batch for the protocol to be secure. Specifically, in Section 4.2 we describe how to implement the **Mult** command and in Section 4.3 we describe how to implement the **ReOrg** command.

4.2 Generating Multiplication Triples

Secure multiplication in our online phase, similar to previous works in the field, is performed using multiplication triples (AKA Beaver triples). In our work a multiplication triple is of the form $(\llbracket \mathbf{x} \rrbracket, \llbracket \mathbf{y} \rrbracket, \llbracket \mathbf{z} \rrbracket)$ where $\llbracket \mathbf{x} \rrbracket, \llbracket \mathbf{y} \rrbracket$ and $\llbracket \mathbf{z} \rrbracket$ are multiparty commitments of messages $\mathbf{x}, \mathbf{y}$ and $\mathbf{z}$ respectively as defined in Section 3.3 and $\mathbf{z} = \mathbf{x} * \mathbf{y}$. The construction of triples is done in a batch and consists of four parts briefly described below (and further explained and analyzed soon afterward):

1. **Construction.** Using the arithmetic OT procedure formalized in Section 2.3 the parties first *construct* multiplication triples that may be "malformed" and "leaky" in case of a malicious adversary. Here malformed means that they are incorrect, i.e. $\mathbf{x} * \mathbf{y} \neq \mathbf{z}$ and "leaky" means that the adversary has tried to guess the value of the share of an honest party (the term is further explained below).
2. **Cut-and-Choose** The parties select τ_1 triples at random which they check for correctness. If any of these triples are malformed then they abort. Otherwise, when mapping the remaining triples into buckets, with overwhelming probability all buckets will contain at least one correct triple.
3. **Sacrificing.** The remaining triples (from the cut-and-choose) are mapped to buckets, τ_1 triples in each bucket such that at least one of the triples is correct. Each bucket is then tested to check its correctness where by this check only a single multiplication is being output while the other $\tau_1 - 1$ are being discarded. This step guarantees that either the output triple is correct or a malformed triple is detected, in which case the protocol aborts.
4. **Combining.** As some of the triples may be "leaky" this allows the adversary to carry a selective attack, that is, to probe whether its guess was correct or not. If the guess is affected by the input of an honest party then it means that the adversary learns that input. Thus, as the name suggests, the goal of this step is to produce a non-leaky triple by combining τ_2 triples, which are the result of the sacrificing step (and thus are guaranteed to be correct), where at least one of the τ_2 is non-leaky. As we will see later, this condition is satisfied with overwhelming probability.

Construction. The triples are generated in a batch, that is, sufficiently many triples are generated at once. However, the construction of each triple is independent of the others. Thus, we proceed by describing how to generate a single triple. The parties select three raw-commitments, denoted $\llbracket \mathbf{x} \rrbracket, \llbracket \mathbf{y} \rrbracket, \llbracket \mathbf{z}' \rrbracket$, that were generated by $\mathcal{F}_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$. The goal of this step is to change $\llbracket \mathbf{z}' \rrbracket$ to $\llbracket \mathbf{z} \rrbracket$ such that $\llbracket \mathbf{z} \rrbracket = \llbracket \mathbf{x} * \mathbf{y} \rrbracket$.

Recall that for a message $\mathbf{x}$ that is committed to by all parties, we have that each party P_i knows $\mathbf{x}^i$ such that $\mathbf{x} = \sum_{i\in[p]} \mathbf{x}^i$. Thus, the product $\mathbf{x} * \mathbf{y}$ equals $\left(\sum_{i\in[p]} \mathbf{x}^i\right) * \left(\sum_{i\in[p]} \mathbf{y}^j\right) = \sum_{i\in[p]} \mathbf{x}^i * (\sum_{j\in[p]} \mathbf{y}^j)$. In order to have each party P_i hold the value $\mathbf{z}^i$ such that $\sum_{i\in[p]} \mathbf{z}^i = \mathbf{x} * \mathbf{y}$ we let party P_i use the arithmetic OT procedure (as describe in Section 2.3) to have a share of the multiplication $\mathbf{x}^i * \mathbf{y}^j$ for every $j \in [p]$ where P_i inputs $\mathbf{x}^i$ and P_j inputs $\mathbf{y}^j$. After P_i multiplied its share $\mathbf{x}^i$ with all other parties' shares $\mathbf{y}^j$ the sum of all the shares is $\mathbf{x}^i * (\sum_{j\in[p]} \mathbf{y}^j)$ (assuming honest behavior). If all parties do the same, then every party ends up holding a share of $\mathbf{x} * \mathbf{y}$ as required. Remember that we want P_i to hold a share to $[\![\mathbf{x} * \mathbf{y}]\!]$ and not just a share to $\mathbf{x} * \mathbf{y}$ (i.e. we want all shares to be committed). To this end, every party broadcasts the difference $\mathbf{t}$ between the new share and the old share, that is, P_i broadcasts $\mathbf{t}^i = \mathbf{z}^i - \mathbf{z}'^i$, then, the parties perform a constant addition to the old commitments, that is, they compute $[\![\mathbf{z}]\!] = [\![\mathbf{z}']\!] + (\sum_{i\in[p]} \mathbf{t}^i)$.

Discussion. As described above, party P_i (for $i \in [p]$) participates in $p-1$ instantiations of the arithmetic OT functionality with every other party P_j (for $j \neq i$). The arithmetic OT functionality is of the form $(\mathbf{x}^i, (\mathbf{y}^j, \mathbf{r}^j)) \mapsto (\mathbf{x}^i * \mathbf{y}^j + \mathbf{r}^j, \perp)$, that is, P_i inputs its share $\mathbf{x}^i$ of $\mathbf{x}$, party P_j inputs its share $\mathbf{y}^j$ of $\mathbf{y}$ and a random value $\mathbf{r}^j$. The functionality outputs $\mathbf{x}^i * \mathbf{y}^j + \mathbf{r}^j$ to P_i and nothing to P_j. Then, to get a sharing of $\mathbf{x}^i * \mathbf{y}^j$ we instruct P_i to store $\mathbf{x}^i * \mathbf{y}^j + \mathbf{r}^j$ and P_j to store $-\mathbf{r}^j$ (see Section 2.3). Even if this arithmetic OT subprotocol is maliciously secure, it will only give semi-honest security in our setting when composed with the rest of the scheme. Specifically, there are two possible attacks that might be carried out by a malicious adversary:

1. Party P_j may input $\tilde{\mathbf{y}}^j \neq \mathbf{y}^j$ such that $\mathbf{e} = \tilde{\mathbf{y}}^j - \mathbf{y}^j$, in the instantiation of the arithmetic OT with every other P_i, where $\mathbf{y}^j$ is the value it is committed to. This results with the parties obtaining a committed share of the triple $([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{x} * (\mathbf{y} + \mathbf{e})]\!])$. We call such a triple a "malformed" triple.
2. In the arithmetic OT procedure party P_j may impact the output of P_i such that P_i obtains $\mathbf{x}^i * \mathbf{y}^j + \mathbf{r}^j$ only if the k'th value of $\mathbf{x}^i$ is equal to some value "guessed" by P_j, otherwise P_i obtains some garbage $\mathbf{x}^i * \tilde{\mathbf{y}}^i \in \mathbb{F}^m$. A similar attack can be carried out by P_i on $\mathbf{y}^j$ when computing over a "small" field (see the description of the malicious behavior in Section 2.3). In both cases, the parties obtain committed shares of the triple $([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{x} * \mathbf{y}]\!])$ only if the malicious party made a correct guess on an honest party's share, and an incorrect triple otherwise. Thus, when using this triple later on, the malicious party learns if it guessed correctly depending on whether the honest parties abort, thus, it is vulnerable to a "selective attack". We call such a triple "leaky", since it might leak privates bits from the input of an honest party.

We take three countermeasures (described in the next items) to produce correct and non-leaky triples:

1. In the *Cut-and-Choose* step we verify that a few (τ_1) randomly selected triples have been constructed correctly. This is done, by having each party open his committed shares associated with these triples and all parties verifying that the triples has been constructed according to the protocol. This step is required to ensure that not all triples were malformed as a preliminary for the sacrificing step (below) in which the triples are mapped to buckets. When working over $\mathbb{F} = \mathrm{GF}(2)$, this step is strictly needed to eliminate the case that all triples are malformed. For other fields, this step improves the amount of triples to be constructed in the batch.
2. In the *Sacrificing* step we make sure that a triple is correct (i.e. not malformed) by "sacrificing" $\tau_1 - 1$ other triples which are being used as a "one-time-pads" of the correct triple. As we treat a bunch of triples at once, the probability of an incorrect triple to pass this step without being detected is negligible in s (analysis is presented below). Having the parties committed (in the construction step) to $\tau_1 \cdot T$ triples, by the end of this step there will be T *correct* triples.
3. In the *Combining* step we partition the constructed (correct but possibly leaky) triples into buckets of τ_2 triples each, and show that for a sufficiently big number of triples that are the outcome of the sacrificing step, the probability that there exist a bucket in which all triples are leaky in a specific component is negligible in s. We show how to combine the τ_2 triples in a bucket and produce a new triple which is non-leaky. This is done twice, first to remove leakage on the $\mathbf{x}$ component and second to remove leakage on the $\mathbf{y}$ component.

Cut-and-Choose. The parties use $\mathcal{F}_{\mathsf{CT}}$ to randomly pick τ_1 triples to check. Note that τ_1 is the bucket-size used in *Sacrificing* below and in practice could be as low as 3 or 4. It was shown in [20] that when partitioning the triples into buckets of size τ_1 (where many of them may be malformed) then by sampling and checking only τ_1 triples, the probability that there exist a bucket full of malformed triples is negligible. Formally:

Corollary 4.1 (Corollary 6.4 in [20]). *Let $N = \tau_1 + \tau_1(\tau_2)^2 \cdot T$ be the number of constructed triples where $s \leq \log_2\left(\frac{(N\cdot\tau_1+\tau_1)!}{N\cdot\tau_1!\cdot(N\cdot\tau_1)!}\right)$, then, by opening τ_1 triples it holds that every bucket contains at least one correct triple with overwhelming probability.*

Hence, it is sufficient to open (and discard) τ_1 triples out of the triples from the Construction step and hand the remaining to the Sacrificing step below.

Sacrificing. In the following we describe how to produce $(\tau_2)^2 \cdot T$ correct triples out of $\tau_1 \cdot (\tau_2)^2 \cdot T$ that were remained from the cut-and-choose step, and analyze what should T and τ_1 be in order to have all produced $(\tau_2)^2 \cdot T$ triples correct with overwhelming probability. We have the $(\tau_2)^2 \cdot T$ triples be uniformly assigned to buckets where each bucket contains τ_1 triples, denoted $\{t_k\}_{k\in[\tau_1]}$. For simplicity, in the following we assume that $\tau_1 = 3$. For every bucket, the parties apply the procedure $\mathsf{CorrectnessTest}$ (see Fig. 11) to triples t_1 and t_2. If the procedure returns successfully (i.e. the parties do not abort) they run the procedure again, this time with triples t_1 and t_3. Finally, if the procedure returns successfully from the second invocation as well then the withs consider t_1 as a correct triple, otherwise they abort the protocol. We note that this procedure is similar to the one used in [14] and other works.

Procedure $\mathsf{CorrectnessTest}(t_1, t_2)$**.**
Given the two triples $t_1 = ([\![\mathbf{a}]\!], [\![\mathbf{b}]\!], [\![\mathbf{c}]\!])$ and $t_2 = ([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{z}]\!])$ the parties do as follows:

1. Invoke $\mathcal{F}_{\mathsf{CT}}$ with the command $(\mathtt{toss}, 1, \mathbb{F}\setminus\{0\})$ to produce a uniformly random scalar $r \in_R \mathbb{F}\setminus\{0\}$.
2. Locally compute $[\![\boldsymbol{\epsilon}]\!] = r \cdot [\![\mathbf{x}]\!] - [\![\mathbf{a}]\!]$ and $[\![\boldsymbol{\rho}]\!] = [\![\mathbf{y}]\!] - [\![\mathbf{b}]\!]$ and publicly open $\boldsymbol{\epsilon}$ and $\boldsymbol{\rho}$, both in $\mathbb{F}^m$.
3. Locally compute $[\![\mathbf{e}]\!] = r \cdot [\![\mathbf{z}]\!] - [\![\mathbf{c}]\!] - \boldsymbol{\epsilon} * [\![\mathbf{b}]\!] - \boldsymbol{\rho} * [\![\mathbf{a}]\!] - \boldsymbol{\rho} * \boldsymbol{\epsilon}$ and publicly open $\mathbf{e} \in \mathbb{F}^m$.
4. If $\mathbf{e} \neq \mathbf{0}$ then abort. Otherwise output t_1.

Fig. 11. Procedure $\mathsf{CorrectnessTest}(t_1, t_2)$

Security. The correctness and security is explained in [14]. However, for completeness we prove the following lemma in Appendix C, which states that after the sacrificing step all produced triples are correct with overwhelming probability:

Lemma 4.2. *When $2^{-s} \leq \frac{(|\mathbb{F}|-1)^{1-\tau_1}\cdot(\tau_2)^2\cdot T\cdot(\tau_1\cdot(\tau_2)^2\cdot T)!\cdot\tau_1!}{(\tau_1\cdot(\tau_2)^2\cdot T+\tau_1)!}$ all the $(\tau_2)^2 \cdot T$ triples that are produced by the sacrificing step are correct except with probability at most 2^{-s}.*

Combining. The goal of this step is to produce T non-leaky triples out of the $(\tau_2)^2 \cdot T$ triples remained from the sacrificing step above. We do this in two sub-steps: First to remove the leakage (with regard to the arithmetic OT) of the sender and then to remove the leakage from the receiver. In each of the sub-steps we map the triples to buckets of size τ_2 and produce a single non-leaky triple out of it. In the following we first show how to produce one triple from each bucket with the apriori knowledge that at least one of the triples in the bucket is non-leaky (but we do not know which one is it) and later we show how to obtain such buckets. Denote the set of τ_2 triples by $\{([\![\mathbf{x}_k]\!], [\![\mathbf{y}_k]\!], [\![\mathbf{z}_k]\!])\}_{k\in[\tau_2]}$. We produce the triple $([\![(\mathbf{x}')]\!], [\![\mathbf{y}']\!], ([\![\mathbf{z})']\!])$ out of that set in the following way: The parties compute

$$[\![\mathbf{x}']\!] = \left[\!\!\left[\sum_{k\in[\tau_2]} \mathbf{x}_k\right]\!\!\right] \quad \text{and} \quad [\![\mathbf{y}']\!] = [\![\mathbf{y}_1]\!] \quad \text{and} \quad [\![\mathbf{z}']\!] = \left[\!\!\left[\left(\sum_{k\in[\tau_2]} \mathbf{x}_k\right) * \mathbf{y}_1\right]\!\!\right]$$

which constitute the triple $([\![\mathbf{x}']\!], [\![\mathbf{y}']\!], [\![\mathbf{z}']\!])$. It is easy to see that $[\![\mathbf{x}']\!]$ can be computed locally since it requires additions and constant multiplications only. Furthermore, $\mathbf{x}'$ is completely hidden since at least one of $\mathbf{x}_1, \ldots, \mathbf{x}_k$ was not leaked (and it is guaranteed from the construction step that it is chosen uniformly at random from $\mathbb{F}^m$). However, notice that $[\![\mathbf{z}']\!]$ cannot be computed locally, since it is required to multiply two multiparty commitments $\left[\!\!\left[\left(\sum_{k\in[\tau_2]} \mathbf{x}_k\right)\right]\!\!\right]$ and $[\![\mathbf{y}_1]\!]$. Thus, to obtain $[\![\mathbf{z}']\!]$ the parties first compute $[\![\boldsymbol{\epsilon}_k]\!] = [\![\mathbf{y}_1 - \mathbf{y}_k]\!]$ and open $\boldsymbol{\epsilon}_k$ for every $k = 2, \ldots, \tau_2$. Then compute $[\![\mathbf{z}']\!] = \left[\!\!\left[\mathbf{z}_1 + \sum_{k=2}^{\tau_2} \boldsymbol{\epsilon}_k * \mathbf{x}_k + \mathbf{z}_k\right]\!\!\right]$ by a local computation only.

We prove the following lemma in Appendix D:

Lemma 4.3. *Having a batch of at least* $\sqrt[\tau_2-1]{\frac{(s\cdot e)^{\tau_2}\cdot 2^s}{\tau_2}}$ *triples as input to a combining step, every bucket of* τ_2 *triples contains at least one non-leaky triple with overwhelming probability in* s *in the component that has been combined on.*

For instance, when $\mathbb{F} = \mathrm{GF}(2)$ having $s = 40$, $\tau_1 = 3$ $\tau_2 = 4$ it is required to construct $T \approx 8.4 \cdot 10^5$ correct and non-leaky triples in a batch. Instead, having $\tau_2 = 3$ means that $\approx 2.29 \cdot 10^8$ triples are required.

Working Over Non-binary Fields. When $\mathbb{F}$ is a field with odd characteristic then there is a gap between the maximal field element and the maximal value that is possible to choose which can fit in the same number of bits. For instance, when working over $\mathbb{F}_{11}$ then the maximal element possible is $10_{10} = 0101_2$ while the maximal value possible to fit in 4 bits is $15_{10} = 1111_2$, i.e. there is a gap of 5 elements. That means that an adversary could input a value that is not in the field and might harm the security.

We observe that the only place where this type of attack matters is in the ArithmeticOT procedure, since in all other steps the values that the adversary inputs percolate to the underlying homomorphic commitment scheme. In the following we analyze this case: To multiply x^i and y^j with $x^i, y^j \in \mathbb{F}_{\mathcal{P}}$ and $\mathcal{P}$ prime the parties P_i and P_j participate in a protocol of $\lceil \log \mathcal{P} \rceil$ steps. In the q-th step, where $q \in [\lceil \log \mathcal{P} \rceil]$, party P_i inputs x^i_q and P_2 inputs $s^0_q = r_q$ and $s^1_q = r_q + y^j$ to the $\mathcal{F}_{\mathsf{OT}}$ functionality. The functionality outputs $s^{x^i_q}$ to P_1 which updates the sum of the result. In the end of this process the parties hold shares to the multiplication $z = x^i \cdot y^j$.

We first examine the cases in which either s^0_q or s^1_q are not in the prime field, i.e. they belong to the gap $\mathsf{gap} = [2^{\lceil \log \mathcal{P} \rceil}] \smallsetminus \mathbb{F}_{\mathcal{P}}$. We first note that if both of them are in gap then this is certainly detected by P_1 (since P_1 receives one of them as the $\mathcal{F}_{\mathsf{OT}}$'s output). If only one of s^0_q, s^1_q is in gap then one of two cases occurs:

1. If the value that P_1 received from $\mathcal{F}_{\mathsf{OT}}$ is in gap then it is detected immediately as before (since P_1 clearly sees that the value is not in $\mathbb{F}_{\mathcal{P}}$) and can abort. Since this is the preprocessing phase it is independent of any secret input.
2. If the value that P_1 received from $\mathcal{F}_{\mathsf{OT}}$ is in $\mathbb{F}_{\mathcal{P}}$ but the other value is not, then it is guaranteed that the value P_1 obtains is a correct share. That the dishonest P_2 obtains a share in the gap is actually the same case as if P_2 adds an incorrect value to the sum s.t. it lands in the gap. This leads to two cases
   (a) If the incorrect value is $s^0_q \neq r_q$ then this is equivalent to add $s^0_q \mod \mathcal{P}$, which leads to an incorrect share of z. This case is detected in the sacrificing step.
   (b) If the incorrect value is $s^1_q \neq r_q + y^j$ then this is equivalent to add $s^1_q \mod \mathcal{P}$. As above, this leads to an incorrect share of z which is being detected in the sacrificing step.

The last case is when either r_q or y^j (or both) are not in $\mathbb{F}_{\mathcal{P}}$ but the sum s^1_q does. Then this is equivalent to choosing $y^j \in \mathbb{F}_{\mathcal{P}}$ and $r'_q = s^1_q - y^j \mod \mathcal{P}$ such that the value that P_2 adds to its sum is incorrect (since it is different than r'_q), and thus, this is being detected in the sacrificing step as before.

Similarly, consider a corrupted receiver who organizes its bits of x^j to represent an element in gap. We observe that this is equivalent to a receiver who inputs an incorrect value (value that is not committed before) for the following reason: The adversary knows nothing about the sender's (honest party) share y^j, let the value that P_i inputs be $\tilde{x}^i$, thus the ArithmeticOT procedure outputs shares to $\tilde{x}^i y^j \mod \mathcal{P} = (\tilde{x}^i \mod \mathcal{P})(y^j \mod \mathcal{P})$. Now, if $\tilde{x}^i \mod \mathcal{P} = 0$ (i.e. $\tilde{x}^i = \mathcal{P}$) then this is detected by the sacrificing procedure (since $0 \in \mathbb{F}_{\mathcal{P}}$ is not in the field). Otherwise, if $\tilde{x}^i \mod \mathcal{P} \neq 0$ then the result $\tilde{x}^i y^j \mod \mathcal{P}$ is a random element in the field $\mathbb{F}_{\mathcal{P}}$ and the same analysis from the proof of Lemma 4.2 follows.

Finally we make the observation that the math still work out in case we use an extension field and not a plain prime-field. Basically using the ArithmeticOT procedure we can still multiply with one bit at a time. The parties simply multiply with the appropriate constants in the extension field (and thus do any necessary polynomial reduction), instead of simply a two-power.

We prove the following theorem in Appendix E.

Theorem 4.4. *The method* ***Mult*** *in* $\Pi_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ *(Fig. 13) UC-securely implements the method* ***Mult*** *in functionality* $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ *(Fig. 10) in the* $\mathcal{F}_{\mathsf{OT}}$*-,* $\mathcal{F}_{\mathsf{EQ}}$*- and* $\mathcal{F}_{\mathsf{CT}}$*-hybrid model against a static and malicious adversary corrupting a majority of the parties.*

Protocol $\Pi_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$. Describes the implementation of $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ in the $\mathcal{F}_{\mathsf{OT}}, \mathcal{F}_{\mathsf{EQ}}\text{-}\ \mathcal{F}_{\mathsf{CT}}$-hybrid model. The protocol is an interaction between p parties, if $\mathcal{F}_{\mathsf{OT}}, \mathcal{F}_{\mathsf{EQ}}$ or $\mathcal{F}_{\mathsf{CT}}$ outputs `abort` at any point, so does this protocol. The parties begin the protocol with an empty dictionary ReOrg.

Init, Commit, Input, Rand, Linear Combination, Open, Partial Open:
Do exactly as in protocol $\Pi_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$ in Fig. 9.

ReOrg: The parties wish to construct reorganization pairs based on the linear function ϕ using the raw commitments with identifiers set K where $|K| = 2v + 2s$ for some v. If $\mathsf{raw}^i[k] \neq \perp$ for each $k \in K$ and $i \in [p]$ then partition K into the sets X, Y, A, B where $|X| = |Y| = v$ and $|A| = |B| = s$ and proceed as follows:

1. For each of the v pairs $\{(x, y)\} \in (X, Y)$ each party i broadcasts the value $\boldsymbol{\epsilon}^i_{x,y} = \phi(\mathbf{x}^i_x) - \mathbf{x}^i_y$.
2. For each of the s pairs $\{(a, b)\} \in (A, B)$ each party i broadcasts the value $\boldsymbol{\epsilon}^i_{a,b} = \phi(\mathbf{x}^i_a) - \mathbf{x}^i_b$.
3. For every pair $(x, y) \in (X, Y)$ and every pair $(a, b) \in (A, B)$ the parties pick freshly new indexes y' and b' and compute $\left[\!\left[\mathbf{x}_{y'}\right]\!\right] = \left[\!\left[\mathbf{x}_y\right]\!\right] + \sum_{j\in[P]} \boldsymbol{\epsilon}^j_{x,y}$ and $[\![\mathbf{x}_{b'}]\!] = [\![\mathbf{x}_b]\!] + \sum_{j\in[P]} \boldsymbol{\epsilon}^j_{a,b}$. Meaning that $\left[\!\left[\mathbf{x}_{y'}\right]\!\right] = [\![\phi(\mathbf{x}_x)]\!]$ and $[\![\mathbf{x}_{b'}]\!] = [\![\phi(\mathbf{x}_a)]\!]$. Let Y' be the set of y' and likewise let B' be the set of b'.
4. All parties input $(\mathtt{toss}, s \cdot v, \mathbb{F})$ to $\mathcal{F}_{\mathsf{CT}}$ and thus learn $(\mathtt{random}, \mathbf{R})$ (when viewing the output as a matrix $\mathbf{R} \in \mathbb{F}^{s\times v}$).
5. The parties now compute and open the linear combination for each $q \in [s]$, letting $\mathbf{R}_{q,k}$ denote the element in the q'th row of the k'th column of $\mathbf{R}$:
$$\left[\!\left[\mathbf{s}_q\right]\!\right] = \left[\!\left[\mathbf{x}_{A_q}\right]\!\right] + \sum_{k\in[v]} \mathbf{R}_{q,k} \cdot \left[\!\left[\mathbf{x}_{X_k}\right]\!\right] \quad \text{and} \quad \left[\!\left[\bar{\mathbf{s}}_q\right]\!\right] = \left[\!\left[\mathbf{x}_{B'_q}\right]\!\right] + \sum_{k\in[v]} \mathbf{R}_{q,k} \cdot \left[\!\left[\mathbf{x}_{Y'_k}\right]\!\right]$$
6. Each party now verifies that $\phi(\mathbf{s}_q) = \bar{\mathbf{s}}_q$. If not, they abort.
7. The parties set $\mathsf{ReOrg}^i[q] = (X_q, Y'_q)$, $\mathsf{actual}^i[X_q] = \left[\!\left[\mathbf{x}_{X_q}\right]\!\right]^i$, $\mathsf{actual}^i[Y'_q] = \left[\!\left[\phi(\mathbf{x}_{X_q})\right]\!\right]^i$ for every $q \in [v]$ and $\mathsf{raw}^i[k] = \perp$ for every $k \in K$. Output $(\mathtt{reOrg}, (X, Y'))$ to all parties.

Fig. 12. Protocol $\Pi_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ - Part 1

4.3 Reorganization of Components of a Commitment

The parties might want to move elements of $\mathbb{F}$ around or duplicate elements of $\mathbb{F}$ within a message. In general we might want to apply a linear function ϕ to a vector in $\mathbb{F}^m$ resulting in another vector in $\mathbb{F}^m$. To do so, they need to preprocess pairs of the form $([\![\mathbf{x}]\!], [\![\phi(\mathbf{x})]\!])$ where $\mathbf{x}$ is random. This is done by first having a pair of random commitments $([\![\mathbf{x}]\!], [\![\mathbf{y}]\!])$ (as the output of the **Commit** instruction of $\mathcal{F}_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$), then, party P_i broadcasts $\boldsymbol{\epsilon}^i = \phi(\mathbf{x}^i) - \mathbf{y}^i$ (i.e. by first applying ϕ on its own share). Note that from linearity of ϕ it follows that $\sum_{i\in[p]} \phi(\mathbf{x}^i) = \phi(\sum_{i\in[p]} \mathbf{x}^i) = \phi(\mathbf{x})$, thus $\sum_{i\in[p]} \boldsymbol{\epsilon}^i = \sum_{i\in[p]} \phi(\mathbf{x}^i) - \mathbf{y}^i = \phi(\mathbf{x}) - \mathbf{y}$. Then, the parties compute $[\![\mathbf{y}']\!] = [\![\mathbf{y}]\!] + \sum_{i\in[p]} \boldsymbol{\epsilon}^i = [\![\mathbf{y}]\!] + \phi(\mathbf{x}) - \mathbf{y} = \phi(\mathbf{x})$. For security reasons this is done simultaneously for a batch of $v + s$ pairs. Finally, the parties complete s random linear combination tests over the batch by producing a uniformly random matrix $\mathbf{R} \in \mathbb{F}^{s\times v}$ (using $\mathcal{F}_{\mathsf{CT}}$). Let $\mathbf{R}_{q,k}$ be the element in the qth row and kth column of $\mathbf{R}$. To perform the test, divide the $v + s$ pairs into two sets A, B of v and s pairs respectively. For each pair $\left(\left[\!\left[\mathbf{z}_q\right]\!\right], \left[\!\left[\mathbf{z}_{q'}\right]\!\right]\right)$ in B for $q \in s$ compute and open

$$\left[\!\left[\mathbf{s}_q\right]\!\right] = \left[\!\left[\mathbf{z}_q\right]\!\right] + \sum_{k\in[v]} \mathbf{R}_{q,k} \cdot [\![\mathbf{x}_k]\!] \quad \text{and} \quad \left[\!\left[\bar{\mathbf{s}}_q\right]\!\right] = \left[\!\left[\mathbf{z}_{q'}\right]\!\right] + \sum_{k\in[v]} \mathbf{R}_{q,k} \cdot [\![\mathbf{y}_k]\!]$$

Each party now verifies that $\phi\left(\mathbf{s}_q\right) = \bar{\mathbf{s}}_q$. If this is so, they accept. Otherwise they abort.

Based on this we state the following theorem, which we prove in Appendix F.

Theorem 4.5. *The method **ReOrg** in $\Pi_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ of Fig. 12 UC-securely implements the method **ReOrg** in functionality $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ of figure Fig. 10 in the $\mathcal{F}_{\mathsf{OT}}$-, $\mathcal{F}_{\mathsf{EQ}}$- and $\mathcal{F}_{\mathsf{CT}}$-hybrid model against a static and malicious adversary corrupting a majority of the parties.*

5 Protocol for Multiparty Computation

In Fig. 14 we show how to realize a fully fledged arithmetic MPC protocol secure against a static and malicious adversary, with the possibility of corrupting a majority of the parties. This protocol is very similar to the one used in MiniMAC [16] and thus we will not dwell on its details.

Protocol $\Pi_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$. Describes the implementation of $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ in the $\mathcal{F}_{\mathsf{OT}}, \mathcal{F}_{\mathsf{EQ}}\text{-} \mathcal{F}_{\mathsf{CT}}$-hybrid model. The protocol is an interaction between p parties, if $\mathcal{F}_{\mathsf{OT}}, \mathcal{F}_{\mathsf{EQ}}$ or $\mathcal{F}_{\mathsf{CT}}$ output abort at any point, so does this protocol. The parties begin the protocol with an empty dictionary mult.

Mult: Upon receiving a message $(\mathtt{mult}, K)$ from all parties where $\mathsf{raw}[k] \neq \perp$ for every $k \in K$, let $|K| = 3(\tau_1 + \tau_1 \cdot (\tau_2)^2 \cdot T)$, assign the commitments into $\tau_1 + \tau_1 \cdot (\tau_2)^2 \cdot T$ triples denoted by $[\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{z}]\!]$.

1. **Construction.** For each of the $\tau_1 + \tau_1 \cdot (\tau_2)^2 \cdot T$ triples denoted by $[\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{z}]\!]$ do as follows:
   (a) Party P_i (for every $i \in [p]$) executes the arithmetic OT procedure $\mathsf{ArithmeticOT}(\mathbf{x}^i, \mathbf{y}^j)$ of Fig. 6 together with every party $P_j \neq P_i$ where P_i inputs $\mathbf{x}^i$ and P_j inputs $\mathbf{y}^j$. Let $\mathbf{s}^i_{i\leftarrow j}$ be the output for P_i and $\mathbf{s}^j_{i\leftarrow j}$ be the output for P_j.
   (b) Every party P_i computes $\mathbf{s}^i = \mathbf{x}^i * \mathbf{y}^i + \sum_{j\neq i} \mathbf{s}^i_{i\leftarrow j} + \sum_{j\neq i} \mathbf{s}^i_{j\leftarrow i}$ and broadcasts $\mathbf{t}^i = \mathbf{s}^i - \mathbf{z}^i$.
   (c) All parties compute and store $[\![\mathbf{z}]\!] = [\![\mathbf{z}]\!] + \sum_{i\in[p]} \mathbf{t}^i = [\![\mathbf{x} * \mathbf{y}]\!]$
2. **Cut-and-Choose.** Assign τ_1 randomly picked triples, out of the $\tau_1 + (\tau_2)^2 \cdot T$ triples constructed above, into a bucket using $\mathcal{F}_{\mathsf{CT}}$. For each triple in this bucket, $([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{z}]\!])$, proceed as follows:
   (a) The parties publicly open $[\![\mathbf{x}]\!], [\![\mathbf{y}]\!]$ and $[\![\mathbf{z}]\!]$.
   (b) Every party locally verifies if $\mathbf{x} * \mathbf{y} \stackrel{?}{=} \mathbf{z}$. If this is the case they discard the triple $([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{z}]\!])$, otherwise they abort.
3. **Sacrificing.** Let $\tau_1 \cdot (\tau_2)^2 T$ be the number of triples remaining, where each triple is of the form $([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{z}]\!])$. The parties do as follows:
   (a) Assign the triples uniformly into τ_1 buckets where each bucket contains exactly τ_1 triples, denoted $t_1, \ldots, t_{\tau_1}$ (the uniform assignment done via the use of the coin tossing functionality $\mathcal{F}_{\mathsf{CT}}$).
   (b) Run $\mathsf{CorrectnessTest}(t_1, t_k)$ for $k \in \{2, \ldots, \tau_1\}$ (see Fig. 11) where k is the raw-commitment ID of $[\![\mathbf{x}]\!]$. Note that according to the procedure, if a malformed triple is detected then the parties abort.
   (c) Consider t_1 as a correct triple.
4. **Combining.** Let $(\tau_2)^2 \cdot T$ be the number of correct triples produced by the above step.
   (a) **Combine on x:** The parties assign the triples uniformly into $\tau_2 T$ buckets of τ_2 triples each (as before, this is done using $\mathcal{F}_{\mathsf{CT}}$). For every bucket, denote the triples it contain by $\{([\![\mathbf{x}_k]\!], [\![\mathbf{y}_k]\!], [\![\mathbf{z}_k]\!])\}_{k\in[\tau_2]}$ the parties do as follows:
      i. Compute $[\![\mathbf{x}']\!] = \left[\!\!\left[\sum_{k\in[\tau_2]} \mathbf{x}_k\right]\!\!\right]$ and $[\![\mathbf{y}']\!] = [\![\mathbf{y}_1]\!]$
      ii. Compute $[\![\boldsymbol{\epsilon}_k]\!] = [\![\mathbf{y}_1 - \mathbf{y}_k]\!]$ and open $\boldsymbol{\epsilon}_k$ for every $k = \{2, \ldots, \tau_2\}$.
      iii. Compute $[\![\mathbf{z}']\!] = \left[\!\!\left[\mathbf{z}_1 + \sum_{k=2}^{\tau_2} \boldsymbol{\epsilon}_k * \mathbf{x}_k + \mathbf{z}_k\right]\!\!\right] = [\![\mathbf{x}']\!] * [\![\mathbf{y}']\!]$.
   (b) **Combine on y:** The parties assign the triples uniformly into T buckets of τ_2 triples each (as before, this is done using $\mathcal{F}_{\mathsf{CT}}$). For every bucket, denote the triples it contain by $\{([\![\mathbf{x}_k]\!], [\![\mathbf{y}_k]\!], [\![\mathbf{z}_k]\!])\}_{k\in[\tau_2]}$ the parties do as follows:
      i. Compute $[\![\mathbf{y}']\!] = \left[\!\!\left[\sum_{k\in[\tau_2]} \mathbf{y}_k\right]\!\!\right]$ and $[\![\mathbf{x}']\!] = [\![\mathbf{x}_1]\!]$
      ii. Compute $[\![\boldsymbol{\epsilon}_k]\!] = [\![\mathbf{x}_1 - \mathbf{x}_k]\!]$ and open $\boldsymbol{\epsilon}_k$ for every $k = \{2, \ldots, \tau_2\}$.
      iii. Compute $[\![\mathbf{z}']\!] = \left[\!\!\left[\mathbf{z}_1 + \sum_{k=2}^{\tau_2} \boldsymbol{\epsilon}_k * \mathbf{y}_k + \mathbf{z}_k\right]\!\!\right] = [\![\mathbf{x}']\!] * [\![\mathbf{y}']\!]$.

Fig. 13. Protocol $\Pi_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ - Part 2

We prove the following theorem in Appendix G:

Theorem 5.1. *The protocol in Fig. 14 UC-securely implements the functionality $\mathcal{F}_{\mathsf{MPC}\text{-}\mathbb{F}^m}$ of figure Fig. 10 in the $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$-hybrid model against a static and malicious adversary corrupting a majority of the parties.*

6 Efficiency

6.1 Practical Optimizations

Several significant optimizations can be applied to our protocol. We chose to describe the optimizations here rather than earlier for the ease of presentation. In the following we present each of the optimizations and sketch out its security.

Using less storage. As we mentioned before, the two-party homomorphic commitment scheme of [18] (described in Appendix A) can be used as an implementation of functionality $\mathcal{F}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$. Briefly, in this two party commitment

Init: The parties invoke $(\texttt{init})$ followed by $(\texttt{commit}, I)$ on $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ to get a sufficient amount of raw commitments. Next the parties call $(\texttt{mult}, \cdot)$ and $(\texttt{reOrg}, \phi, \cdot)$ to get a sufficient amount of multiplication triples, $([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{z}]\!])$ and reorganization pairs $([\![\mathbf{x}]\!], [\![\phi(\mathbf{x})]\!])$.

Input: To share P_i's input $\mathbf{y} \in \mathbb{F}^m$, party P_i calls $(\texttt{Input}, i, k, \mathbf{y})$ on $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ with k being the identifier of a raw commitment. All other parties P_j call $(\texttt{Input}, j, k)$. The parties obtain commitment $[\![\mathbf{y}_k]\!]$.

Rand: All parties call $(\texttt{random}, k)$ on $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ with k being an identifier of a raw commitment. The parties obtain commitment $[\![\mathbf{x}_k]\!]$.

Public Add: To add together a public value $\mathbf{y}$ and a commitment, $[\![\mathbf{x}]\!]$, the parties simply compute $\mathbf{y} + [\![\mathbf{x}]\!] = [\![\mathbf{y} + \mathbf{x}]\!]$ using the **Linear** command on $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$.

Add: To add two commitments together, $[\![\mathbf{x}]\!]$ and $[\![\mathbf{y}]\!]$ the parties simply compute $[\![\mathbf{x}]\!] + [\![\mathbf{y}]\!] = [\![\mathbf{x} + \mathbf{y}]\!]$ using the **Linear** command on $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$.

Public Multiply: To multiply together a public value $\mathbf{y}$ and a commitment, $[\![\mathbf{x}]\!]$, the parties simply compute $\mathbf{y} * [\![\mathbf{x}]\!] = [\![\mathbf{y} * \mathbf{x}]\!]$ using the **Linear** command on $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$.

Multiply: To multiply together two commitments, $[\![\mathbf{x}]\!]$ and $[\![\mathbf{y}]\!]$, the parties select a preprocessed multiplication triple $([\![\mathbf{a}]\!], [\![\mathbf{b}]\!], [\![\mathbf{c}]\!])$ and proceed as follows:

1. The parties open $\epsilon = [\![\mathbf{x}]\!] - [\![\mathbf{a}]\!]$ and $\rho = [\![\mathbf{y}]\!] - [\![\mathbf{b}]\!]$ using the commands **Linear** and **Open** on $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$.
2. The parties compute $[\![\mathbf{z}]\!] = [\![\mathbf{x} * \mathbf{y}]\!] = [\![\mathbf{c}]\!] + \epsilon * [\![\mathbf{b}]\!] + \rho * [\![\mathbf{a}]\!] + \epsilon * \rho$ using the command **Linear** on $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$.

Reorganize: To apply a linear operator ϕ to commitment $[\![\mathbf{x}]\!]$ the parties select a preprocessed reorganization pair $([\![\mathbf{a}]\!], [\![\phi\mathbf{a}]\!])$. They then proceed as follows:

1. The parties open $\epsilon = [\![\mathbf{x}]\!] - [\![\mathbf{a}]\!]$ using the commands **Linear** and **Open** on $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$.
2. The parties then compute $[\![\phi(\mathbf{x})]\!] = [\![\phi(\mathbf{a})]\!] + \phi(\epsilon)$ using the commands **Linear** on $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$.

Output: The parties open the value $[\![\mathbf{x}]\!]$ that should be output of the computation using the command **Open** on $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$.

Fig. 14. Protocol UC-realizing $\mathcal{F}_{\mathsf{MPC}\text{-}\mathbb{F}^m}$ in the $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ model.

scheme the committer holds a set of $2m$ vectors from $\mathbb{F}^\gamma$, namely the vectors $\bar{\mathbf{s}}_1^0, \bar{\mathbf{s}}_1^1, \ldots, \bar{\mathbf{s}}_m^0, \bar{\mathbf{s}}_m^1$ whereas the receiver choose a set of m bits $b_1, \ldots, b_m$, denoted as "its choice of watch bits" and obtains the m vectors $\bar{\mathbf{s}}_1^{b_1}, \ldots, \bar{\mathbf{s}}_m^{b_m}$, denoted as "the watchbits".

Recall that in our multiparty homomorphic commitment scheme party P_i participates as a receiver in $p-1$ instances of two-party commitment scheme with all other parties. This means that P_i needs to remember its choice of watchbits for every other party and this accordingly for every linear operation that is performed over the commitments. For instance, let $[\![\mathbf{x}]\!], [\![\mathbf{y}]\!]$ be two multiparty commitments between three parties, then party P_1 stores $[\![\mathbf{x}]\!]^1 = \left\{\left\{[\mathbf{x}^2]^{2,1}, [\mathbf{x}^2]^{3,1}\right\}, \left\{\langle\mathbf{x}^1\rangle^{1,2}, \langle\mathbf{x}^1\rangle^{1,3}\right\}\right\}$. To perform the operation $[\![\mathbf{x}]\!] + [\![\mathbf{y}]\!]$ then P_1 end up with

$$[\![\mathbf{x} + \mathbf{y}]\!]^1 = \left\{\left\{[\mathbf{x}^2]^{2,1} + [\mathbf{y}^2]^{2,1}, [\mathbf{x}^2]^{3,1} + [\mathbf{y}^2]^{3,1}\right\},\ \left\{\langle\mathbf{x}^1\rangle^{1,2} + \langle\mathbf{y}^1\rangle^{1,2}, \langle\mathbf{x}^1\rangle^{1,3} + \langle\mathbf{y}^1\rangle^{1,3}\right\}\right\}$$

To make it more efficient, P_i can choose the bits $b_1, \ldots, b_m$ only once and use them in *all* instances of two-party commitments. This makes the process of linear operations over commitments simpler and does not requires from P_1 to store the commitments for $p-1$ parties. Applying the optimization to the above example, we have that P_1 stores only a single value for the commitment part, that is, now P_1 needs to store

$$[\![\mathbf{x} + \mathbf{y}]\!]^1 = \left\{[\mathbf{x}^2]^{2,1} + [\mathbf{y}^2]^{2,1} + [\mathbf{x}^2]^{3,1} + [\mathbf{y}^2]^{3,1},\ \left\{\langle\mathbf{x}^1\rangle^{1,2} + \langle\mathbf{y}^1\rangle^{1,2}, \langle\mathbf{x}^1\rangle^{1,3} + \langle\mathbf{y}^1\rangle^{1,3}\right\}\right\}$$

Security follows from the underlying commitment scheme, since what we now do is simply equivalent to storing a sum of commitments in a single instance of the two-party scheme.

In a bit more detail, we see that since $\mathcal{F}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$ is UC-secure, it is secure under composition. Furthermore, considering the worst case where only a single party is honest and all other parties are malicious and colluding we then notice that the above optimization is equivalent to executing $p-1$ instances of the $\mathcal{F}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$, but where the same watchbits are chosen by the honest party. We see that this is almost the same as calling **Commit** p times. The only exception is that the seeds of the committing party, P_i, of the calls to $\mathcal{F}_{\mathsf{OT}}$ are different in our optimized protocol. Thus it is equivalent to the adversary being able to select p potentially different seeds to the calls to **Commit**. However, the output of the PRG calls are indistinguishable from random in both cases and so the distributions in both cases are indistinguishable assuming p is polynomial in the security parameter.

Optimized CorrecnessTest. Recall that in the sacrificing step of protocol $\Pi_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ (see Fig. 13) the parties perform two openings of commitments for every bucket (the opening is described as part of the CorrecnessTest in Fig. 11). That is, beginning the step with $\tau_1 \cdot (\tau_2)^2 \cdot T$ triples (which are assigned to $(\tau_2)^2 \cdot T$ buckets) leads to the opening of $(\tau_1 - 1) \cdot (\tau_2)^2 \cdot T$ triples.

Since we require that the results of all of these openings be $\mathbf{0}$, then any linear combination over these opening would be $\mathbf{0}$ as well if they are correct. On the other hand, if one or more of the openings are not zero the result of a linear combination over the openings might be $\mathbf{0}$ with probability $\frac{1}{|\mathbb{F}|}$. Thus, agreeing on a s random linear combinations over the openings would detect an incorrect triple with overwhelming probability.

Optimized opening. In the online phase of our protocol, for every multiplication gate the parties need to open some random commitments using the **Open** command. The implementation of the **Open** command requires interaction between every pair of parties, thus, the communication complexity is $\Omega(T \cdot p^2)$ where T is the number of multiplication gates in the circuit. Following the same idea as used in SPDZ and MiniMAC, we note that we can reduce the communication complexity for every gate to $O(p)$ in the following way, to perform a "partial opening" of a commitment $[\![\mathbf{x}]\!]$:

1. Every party P_i sends its share $\mathbf{x}^i$ to P_1
2. P_1 computes $\mathbf{x} = \sum_{j \in [p]} \mathbf{x}^j$ and sends back $\mathbf{x}$ to everyone.

This incurs a communication complexity of $O(p)$ rather than $O(p^2)$. In the end of the evaluation of the circuit, the parties perform s random linear combinations over the commitment values that were "partially opened" earlier. Then, they open the results of the linear combinations using the **Open** command. If one of the opened results with a wrong value (i.e. that does not conform with the result of the linear combination of the values sent from P_1 in the partial opening) then the parties abort.

Using this optimization leads to a communication complexity of $\Omega(T \cdot p + s \cdot p^2)$. Security follows by the same arguments as used in SPDZ and MiniMAC. Particularly before opening the output nothing gets leaked during the execution of the gates in the protocol and since the adversary does not know the random linear combinations he cannot send manipulated values that pass this check.

Optimizing for large fields. If the field we compute in contains at least 2^s elements, then the construction of multiplication triples becomes much lighter. First see that in this case it is sufficient to only have two triples per bucket for sacrificing. This is because the adversary's success probability of getting an incorrect triple through the CorrectnessTest in Fig. 11 is less than $|\mathbb{F}|^{-1} \leq 2^{-s}$. Next we see that it is possible to eliminate the combining step on the $\mathbf{y}$ components of the triples. This follows since the party inputting x into the ArithmeticOT procedure in Fig. 6 can now only succeed in a selective failure attack on the honest party's input y if he manages to guess y. To see this notice that if the adversary changes the q'th bit of his input x then the result of the computation will be different from the correct result with a factor $y \cdot 2^{q-1}$. But since y is in a field of at least 2^s elements then $y \cdot 2^{i-1} = 0$ with probability at most 2^{-s} and thus its cheating attempt will be caught in the CorrectnessTest with overwhelming probability. Furthermore the combining on $\mathbf{x}$ is now also overly conservative in the bucket size τ_2. To see this notice that the adversary only gets to learn at most $s - 1$ bits in total over all triples. This means that it cannot fully learn the value of a component of $\mathbf{x}$ for all triples in the bucket (since it is at least s bits long), which is what our proof, bounding his success probability assumes. Instead we can now bound its success probability by considering a different attack vectors and using the Leftover Hash Lemma to compute the maximum amount of leakage it can learn when combining less than τ_2 triples in a bucket as done in [28]. However, we leave the details of this as future work. To conclude, even when using the very conservative bound on bucket size, we get that it now takes only $6m \log(|\mathbb{F}|)$ OTs, amortized, when constructing 2^{21} triples instead of $27m \log(|\mathbb{F}|)$ when $s = 40$.

6.2 Efficiency Comparison

The computationally heavy parts in our protocol are the usage of oblivious transfers and the use of the underlying homomorphic two-party commitments. Both of these are rather efficient in practice having the state-of-the-art constructions of Keller *et al.* ([27] for OT) and of Frederiksen *et al.* ([18], for two-party homomorphic commitments). It

Scheme	Finite Field	Rand, Input	Schur, ReOrg	Mult	
		COTe	COTe	COTe	OT
[19]	$\mathbb{F}_{2^c}$ for $c \geq 1$	$m\log(\lvert\mathbb{F}\rvert)$	$m\log(\lvert\mathbb{F}\rvert)$	$24m\log(\lvert\mathbb{F}\rvert)$	$12m\log(\lvert\mathbb{F}\rvert) + 6s$
[28]	$\mathbb{F}_{2^c}$ for $c \geq 2s$	$m\log(\lvert\mathbb{F}\rvert)$	-	$5m\log(\lvert\mathbb{F}\rvert)$	$3m\log(\lvert\mathbb{F}\rvert)$
[7]	$\mathbb{F}_2$	$m\log(\lvert\mathbb{F}\rvert)$	-	$12m\log(\lvert\mathbb{F}\rvert)$	$3m\log(\lvert\mathbb{F}\rvert)$
This work	Any	0	0	0	$27m\log(\lvert\mathbb{F}\rvert)$
This work*	$\mathbb{F}_{2^c}$ for $c \geq s$	0	0	0	$6m\log(\lvert\mathbb{F}\rvert)$

Table 2. Comparison of the overhead of OTs needed, in the amortized sense. All values should be multiplied with $p(p-1)$ to get the true number of needed OTs. We differentiate between regular OTs and the more efficient correlated random OT with error (COTe) [28]. We assume that the computational security parameter $\kappa \geq m\log(|\mathbb{F}|)$ some complexities increase. $\mathbb{F}$ =GF(2). For [7,28] $m = 1$ is possible. We assume at least 2^{21} triples are generated which gives the smallest numbers to the protocols. *) Using optimization 4. in 6.1, requiring $|\mathbb{F}| \geq 2^s$.

should be noted that if one wish to use a binary field, or another small field, then it is necessary to use a code based on algebraic geometry internally if using the commitment scheme of Frederiksen *et al.* [18]. These are however not as efficient to compute as, for example, the BCH code used in the implementation of [18] done in [34].

Notice that the amount of OTs our protocol require is a factor of $O(m\log(|\mathbb{F}|))$ greater than the amount of commitments it require. Therefore, in Table 2 we try to compare our protocol with [19], [28] and [7] purely based on the amount of OTs needed. This gives a fair estimation on the efficiency of our protocol compared to the current state-of-the-art protocols for the same settings (static, malicious majority in the secret sharing approach).

Furthermore, we note that both [28] and [19] (which is used as the underlying preprocessing phase for MiniMAC) require a factor of between $O(m)$ and $O(m^2)$ more coin tosses than our protocol. The reason for this is that in our protocol it is sufficient to perform the random linear combinations using a random *scalar* from $\mathbb{F}$ (i.e. scalar multiplication) whereas [28] and [19] requires a componentwise multiplication using a random *vector* from $\mathbb{F}^m$. Note that in the comparison in Table 2 we adjusted the complexity of [19] to fit what is needed to securely fix the issue regarding the sacrificing, which we present in Appendix H.

7 Applications

Practically all maliciously secure MPC protocols require some form of commitments. Some, e.g. the LEGO family of protocols [33,17,18,34], also require these commitments to be additively homomorphic. Our MPC protocol works directly on such commitments, we believe it makes it possible to use our protocol as a component in a greater scheme with small overhead, as all private values are already committed to. Below we consider one such specific case; when constructing committed OT from a general MPC protocol.

7.1 Bit Committed OT

The bit-OT two-party functionality $(b, x_0, x_1) \mapsto (x_b, \perp)$ can be realized using a secure evaluation of a circuit containing a single AND gate and two XOR gates: Let b denote the choice bit and x_0, x_1 the bit messages, then $x_b = b \wedge (x_0 \oplus x_1) \oplus x_0$.

We notice that all shares in our protocol are based on two-party commitments. This means that constructing a circuit similar to the description above will compute OT, based on shares which are committed to. Thus we can efficiently realize an OT functionality working on commitments. Basically we use $\mathbb{F}$ =GF(2) and compute a circuit with one layer of AND gates computing the functionality above. In the end we only open towards the receiver. At any later point in time it is possible for the sender to open the commitments to x_0 and x_1, no matter what the receiver chose. The sender can also open b towards the receiver. However we notice that we generally need to open m committed OTs at a time (since we have m components in a message). However, if this is not possible in the given application we can use reorganization pairs to open only specific OTs, by simply branching each output message (consisting of m components) into m output messages each of which only opening a single component, and thus only a single actual OT.

Furthermore, since we are in the two-party setting, and because of the specific topology of the circuit we do not need to have each multiparty commitment be the sum of commitments between each pair of parties. Instead the receiving party simply commits to b towards the sending party using a two-party commitment. Similarly the sending party commits to x_0 and x_1 towards the receiving party using a two-party commitment. Now, when they construct a multiplication triple they only need to do one OT per committed OT they construct; the receiver inputting his b and the receiver inputting $x_0 \oplus x_1$. Since the sender should not learn anything computed by the circuit the parties do no need to complete the arithmetic OT in other direction.

In this setting we have $\mathbb{F}$ =GF(2) (hence $m \geq s$), $p = 2$ and 1 multiplication gate when constructing a batch of m committed OTs. Plugging these into the equations in Table 1 we see that the amortized cost for a single committed-OT is 36 regular string OTs of κ bits and $108/m \leq 108/s \leq 3$ (for $s = 40$) commitments for batches of m committed-OTs.

It is also possible to achieve committed OT using other MPC protocols, in particular the TinyOT protocols [32,7] have a notion of committed OT as part of its internal construction. However our construction is quite different.

Acknowledgment The authors would like to thank Carsten Baum and Yehuda Lindell for useful discussions along Peter Scholl and Marcel Keller for valuable feedback and discussions in relation to their SPDZ and MiniMAC preprocessing papers.

References

1. G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *ACM CCS*, pages 535–548, 2013.
2. G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer extensions with security for malicious adversaries. In *EUROCRYPT*, pages 673–701, 2015.
3. D. Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, pages 420–432, 1991.
4. D. Beaver. Correlated pseudorandomness and the complexity of private computations. In *STOC*, pages 479–488, 1996.
5. M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, pages 1–10, 1988.
6. L. T. A. N. Brandão. Very-efficient simulatable flipping of many coins into a well - (and a new universally-composable commitment scheme). In C. Cheng, K. Chung, G. Persiano, and B. Yang, editors, *PKC*, volume 9615 of *Lecture Notes in Computer Science*, pages 297–326. Springer, 2016.
7. 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. *IACR Cryptology ePrint Archive*, 2015:472, 2015.
8. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.
9. I. Cascudo, I. Damgård, B. David, N. Döttling, and J. B. Nielsen. Rate-1, linear time and additively homomorphic UC commitments. In *CRYPTO*, pages 179–207, 2016.
10. I. Cascudo, I. Damgård, B. M. David, I. Giacomelli, J. B. Nielsen, and R. Trifiletti. Additively homomorphic UC commitments with optimal amortized overhead. In *PKC*, pages 495–515, 2015.
11. I. Damgård, B. M. David, I. Giacomelli, and J. B. Nielsen. Compact VSS and efficient homomorphic UC commitments. In *ASIACRYPT*, pages 213–232, 2014.
12. I. Damgård, R. Lauritsen, and T. Toft. An empirical study and some improvements of the minimac protocol for secure computation. In *SCN*, pages 398–415, 2014.
13. I. Damgård and C. Orlandi. Multiparty computation for dishonest majority: From passive to active security at low cost. In *CRYPTO*, pages 558–576, 2010.
14. I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, pages 643–662, 2012.
15. I. Damgård and R. W. Zakarias. Fast oblivious AES A dedicated application of the minimac protocol. In *AFRICACRYPT*, pages 245–264, 2016.
16. I. Damgård and S. Zakarias. Constant-overhead secure computation of boolean circuits using preprocessing. In *TCC*, pages 621–641, 2013.
17. T. K. Frederiksen, T. P. Jakobsen, J. B. Nielsen, P. S. Nordholt, and C. Orlandi. Minilego: Efficient secure two-party computation from general assumptions. In *EUROCRYPT*, pages 537–556, 2013.
18. T. K. Frederiksen, T. P. Jakobsen, J. B. Nielsen, and R. Trifiletti. On the complexity of additively homomorphic UC commitments. In *TCC*, pages 542–565, 2016.
19. T. K. Frederiksen, M. Keller, E. Orsini, and P. Scholl. A unified approach to MPC with preprocessing using OT. In *ASIACRYPT*, pages 711–735, 2015.
20. J. Furukawa, Y. Lindell, A. Nof, and O. Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *EUROCRYPT*, pages 225–255, 2017.

21. J. A. Garay, Y. Ishai, R. Kumaresan, and H. Wee. On the complexity of UC commitments. In *EUROCRYPT*, pages 677–694, 2014.
22. N. Gilboa. Two party RSA key generation. In *CRYPTO*, pages 116–129, 1999.
23. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *STOC*, pages 218–229, 1987.
24. Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *CRYPTO*, pages 145–161, 2003.
25. Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Zero-knowledge from secure multiparty computation. In *STOC*, pages 21–30, 2007.
26. Y. Ishai, M. Prabhakaran, and A. Sahai. Secure arithmetic computation with no honest majority. In *TCC*, pages 294–314, 2009.
27. M. Keller, E. Orsini, and P. Scholl. Actively secure OT extension with optimal overhead. In *CRYPTO*, pages 724–741, 2015.
28. M. Keller, E. Orsini, and P. Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *ACM CCS*, pages 830–842, 2016.
29. E. Larraia, E. Orsini, and N. P. Smart. Dishonest majority multi-party computation for binary circuits. In *CRYPTO*, pages 495–512, 2014.
30. Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *EUROCRYPT*, pages 52–78, 2007.
31. Y. Lindell, B. Pinkas, N. P. Smart, and A. Yanai. Efficient constant round multi-party computation combining BMR and SPDZ. In *CRYPTO*, pages 319–338, 2015.
32. J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A new approach to practical active-secure two-party computation. In *CRYPTO*, pages 681–700, 2012.
33. J. B. Nielsen and C. Orlandi. LEGO for two-party secure computation. In *TCC*, pages 368–386, 2009.
34. J. B. Nielsen, T. Schneider, and R. Trifiletti. Constant round maliciously secure 2PC with function-independent preprocessing using LEGO. In *NDSS*, 2017.
35. P. Rindal and R. Trifiletti. Splitcommit: Implementing and analyzing homomorphic UC commitments. *IACR Cryptology ePrint Archive*, 2017:407, 2017.
36. A. C.-C. Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167, 1986.

A Two-Party Additively Homomorphic Commitments of [18]

For completeness we overview the two-party additively homomorphic commitment scheme of [18]. Furthermore, we show how to extend it to allow multiplication of public vectors rather than just public scalar values. The protocol is formally presented in Fig. 15, Fig. 16 and Fig. 17.

We point out that the ideal functionality $\mathcal{F}_{\text{2HCOM-}\mathbb{F}^m}$ we described in Fig. 7 is slightly different from the functionality described in [18] and implemented by the protocol in Fig. 15, Fig. 16 and Fig. 17. Disregarding the methods **Pair** and **Public Multiplication** the difference is purely based on meta-data and is there solely to make the usage of $\mathcal{F}_{\text{2HCOM-}\mathbb{F}^m}$ simpler and the presentation of our main results easier. Specifically the difference is that the functionality implemented by $\Pi_{\text{HCOM-}\mathbb{F}^m}$ and described in [18] allows openings of linear combinations instead of constructing linear combinations internally, which can then be opened later. We denote the actual functionality implemented in Fig. 15, Fig. 16 and Fig. 17 by $\mathcal{F}'_{\text{2HCOM-}\mathbb{F}^m}$. This functionality is like $\mathcal{F}_{\text{2HCOM-}\mathbb{F}^m}$ when removing **Rand** and linear **Linear Combination** and using the following **Open** and **Input** commands:

Open: Upon receiving a message $(\texttt{open}, (\{(k, \alpha_k)\}_{k\in K}))$ from P_i if $\mathsf{raw}[k] \neq \perp$ and $\alpha_k \in \mathbb{F}$ for every $k \in K$ then send $\texttt{opened}, (\{(k, \alpha_k)\}_{k\in K}), \sum_{k\in K} \alpha_k \cdot \mathbf{x}_k$ to P_j and $\mathcal{S}$

Input: Upon receiving a message $(\texttt{Input}, k, \mathbf{y})$ from P_i, if $\mathsf{raw}[k] \neq \perp$ then set $\mathsf{raw}[k] = \mathbf{y}$ and output $(\texttt{Input}, k)$ to P_j and $\mathcal{S}$.

We see that $\mathcal{F}_{\text{2HCOM-}\mathbb{F}^m}$ can (almost) be implemented in the $\mathcal{F}'_{\text{2HCOM-}\mathbb{F}^m}$ hybrid model by simply storing all **Linear Combination**, **Input** and **Rand** and then internally construct the actual dictionary and simply open the correct linear combinations when receiving an open command, potentially adding a public constant β. This works without issues since only public info, such as indexes and confirmation of the command is sent to parties when issuing **Rand** and **Linear Combination**. Regarding **Input** the only difference is that $\mathcal{F}'_{\text{2HCOM-}\mathbb{F}^m}$ only keeps track of a single structure, raw instead of both raw and actual. This can clearly be perfectly simulated by simply keeping track of actual internally.

The only remaining discrepancy between the functionalities is public multiplication of a constant vector $\boldsymbol{\alpha} \in \mathbb{F}^m$, rather than a scalar. We discuss how to handle this in the following by the additional methods **Pair** and **Public Multiplication**. To do so we us first fix some notations regarding linear codes.

Codes. In our construction we use a systematic linear error correction code $C = [n, m, d]$ over $\mathbb{F}$, that is, a code with dimension m, length n and minimum distance d, where messages are from $\mathbb{F}^m$. We assume that C cyclic is a MDS code[8], that is, it holds that $m + d = n + 1$. We require that $m \cdot \lfloor \log_2(|\mathbb{F}|) \rfloor \geq s$.

Let $C(x)$ denote the encoding of a vector $\mathbf{x}$ as a codeword in a linear code C. The *Schur transform* of C (as described in [16]), denoted C^*, is a linear $[n, m^*, d^*]$ code, defined as the span of the set of vectors $\{\mathbf{x} * \mathbf{y} \mid \mathbf{x}, \mathbf{y} \in C\}$. It holds that $m^* \geq m$ and $d^* \leq d$, but we require that $d^* \geq (s + \log(v))/\log_2(|\mathbb{F}|)$. It should be noted that for small fields such as the binary field, an algebraic geometry code is needed in order to ensure the required distance in the Schur transform. We use the operator $\pi_{m,n} : \mathbb{F}^l \rightarrow \mathbb{F}^{n-m}$ to denote the projection of components from position m to n in some vector of l elements, to a new vector of $n - m$ elements. When we just wish to retrieve the first elements of a vector we use $\pi_m : \mathbb{F}^l \rightarrow \mathbb{F}^m$ to denote the projection of the first m elements into a new vector.

Linear Operations. Following the notation introduced in Section 3.1 we show how the linear operations are reflected when using the base commitment scheme in Fig. 15 and Fig. 16. The computation of these linear operations in $\mathcal{F}_{\text{HCOM-}\mathbb{F}^m}$ is thus done for each pair of parties, using the underlying two-party commitment scheme.

Addition

$$[\mathbf{x}_k] + [\mathbf{x}_{k'}] = [\mathbf{x}_k + \mathbf{x}_{k'}]$$

is equivalent to P_j computing

$$\mathbf{w}_k := \mathbf{w}_k + \mathbf{w}_{k'}$$

and

$$\langle \mathbf{x} \rangle_k + \langle \mathbf{x} \rangle_{k'} = \langle \mathbf{x} \rangle_k + \mathbf{x}_{k'}$$

is equivalent to P_i computing:

$$(\mathbf{t}_k^0, \mathbf{t}_k^1, \mathbf{c}_k^0) := (\mathbf{t}_k^0 + \mathbf{t}_{k'}^0, \mathbf{t}_k^1 + \mathbf{t}_{k'}^1, \mathbf{c}_k^0 + \mathbf{c}_{k'}^0)$$

[8] For concreteness one might just assume that C is a Reed-Solomon code.

Constant addition

$$\mathbf{y} + \langle \mathbf{x} \rangle = \langle \mathbf{y} + \mathbf{x} \rangle$$

is equivalent to P_i computing

$$(\mathbf{t}^0, \mathbf{t}^1, \mathbf{c}^0) := (\mathbf{t}^0, \mathbf{t}^1, \mathbf{c}^0, \mathbf{y})$$

and

$$\mathbf{y} + [\mathbf{x}] = [\mathbf{y} + \mathbf{x}]$$

is equivalent to P_j storing

$$\mathbf{w} := (\mathbf{w}, \mathbf{y})$$

The public vector $\mathbf{y}$ is added to the message after it has been opened verified.

Scalar multiplication

$$\alpha \cdot \langle \mathbf{x} \rangle = \langle \alpha \cdot \mathbf{x} \rangle$$

is equivalent to P_i computing

$$(\mathbf{t}^0, \mathbf{t}^1, \mathbf{c}^0) := (\alpha \cdot \mathbf{t}^0, \alpha \cdot \mathbf{t}^1, \alpha \cdot \mathbf{c}^0)$$

and

$$\alpha \cdot [\mathbf{x}] = [\alpha \cdot \mathbf{x}]$$

is equivalent to P_j computing

$$\mathbf{w} := \alpha \cdot \mathbf{w}$$

We notice that for constant addition we do not modify the commitment or verification bits, but simply say that the public message $\mathbf{y}$ should be added after opening $[\mathbf{x}]$. This may seem insecure since we open something else than the actual message. However, since $\mathbf{y}$ is already known to the receiver then it learns $\mathbf{x} + \mathbf{y}$ in any case and can isolate $\mathbf{x}$ on its own. However, there is an issue if we wish to use $\mathbf{x} + \mathbf{y}$ as input to another operation; if it is addition, we simply keep $\mathbf{y}$ "in the head" as part of the commitment resulting from the multiplication. In case of public multiplication of a scalar we simply must also multiply $\mathbf{y}$ with the public scalar and keep this in the head. The problems occur in case of multiplication with a message vector (encoded as a codeword) or another commitment, which we show how to handle below.

A.1 Schur Pairs

Multiplying two codewords together results in a codeword in the Schur transform which has low minimum distance. Notice that this also happens even when we multiply a commitment with public message. We need to convert such a commitment to a commitment in the code C to be able to multiply again. To do so we need to processes commitments to a *Schur Pair* of a random message. Basically a Schur pair is a pair of commitments to the same random message where one is encoded using C and one is encoded using C^*, i.e. $(C(\mathbf{x}), C^*(\mathbf{x}))$ for a random message $\mathbf{x} \in \mathbb{F}^m$. Remember that we assume that C is a Maximum Distance Separable (MDS) and cyclic code. This means that the message space of C^* is at least of size $2m-1$. Thus we must fill the $m-1$ remaining message components when computing $C^*(\mathbf{x})$. In order to avoid leakage in the online phase these $m-1$ extra elements must also be random such that they can act as one-time padding for the product of two codewords in C in the online phase. Thus, to construct a commitment to $\mathbf{x}$ using C^* we require constructing a new instance of the commitment scheme using the $[n, m^*, d^*]$ code C^* instead of C. This is done by calling the **Commit** procedure, using the same seed OTs as we did when constructing the commitments in C. This is done to ensure that P_j gets the same choice of watchbits, $\mathbf{b}$. Because of this overhead we require the construction of Schur Pairs to be done in a batch. The idea is that we then have P_i adjust the value of the commitments in C^*, basically using the **Input** procedure, to ensure that these commit to the same values as the commitments done using C. Then to ensure correctness a linear combination procedure is executed that both C and C^* in the pair encodes the same message in the first m components. To ensure that not too much info is leaked we construct s extra commitments, both in C and C^* which will be used as padding in the linear combination check and discarded afterwards.

Notice that we unfortunately cannot just use C to encode the same message twice even though $C \subseteq C^*$. The reason being that in the online phase the elements in position m to m^* might leak info on the message if we do this.

Protocol between a sender P_i and a receiver P_j. We let $\mathcal{F}_{\mathsf{PRG}} : \{0,1\}^\kappa \to \mathbb{F}^{\mathrm{poly}(\kappa)}$ be a pseudorandom generator with arbitrary polynomial stretch.

Init:

1. On common input $(\mathtt{init}, m)$ we assume the parties agree on a linear code C in systematic form over $\mathbb{F}$ with parameters $[n, m, d]$ along with its Schur code, C^* with parameters $[n, m^*, d^*]$. The parties also initialize an internal set of unique identifiers $\mathrm{ID} = \emptyset$ and another initially empty set U.
2. For $l \in [n]$, P_i picks $\mathbf{r}_l^0, \mathbf{r}_l^1 \in_R \{0,1\}^\kappa$ and inputs $(\mathtt{transfer}, \mathbf{r}_l^0, \mathbf{r}_l^1)$ to $\mathcal{F}_{\mathsf{OT}}$ and P_j picks $b_l \in_R \{0,1\}$ and inputs $(\mathtt{receive}, b_l)$ to $\mathcal{F}_{\mathsf{OT}}$. The functionality replies with $(\mathtt{deliver}, \mathbf{r}_l^{b_l})$ to P_j and $(\mathtt{deliver}, \perp)$ to P_i.

Commit:

1. On common input $(\mathtt{commit}, \gamma)$, for $l \in [n]$, both parties use $\mathcal{F}_{\mathsf{PRG}}$ to extend the first n of their received seeds for $\mathcal{F}_{\mathsf{OT}}$ into vectors of length $\gamma + 2s$. These are denoted $\bar{\mathbf{s}}_l^0, \bar{\mathbf{s}}_l^1 \in \mathbb{F}^{\gamma+2s}$ where P_i knows both and P_j knows $\bar{\mathbf{s}}_{b_l}^0$. Next define the matrices $\mathbf{S}^0, \mathbf{S}^1 \in \mathbb{F}^{n\times(\gamma+2s)}$ such that for $l \in [n]$ the l'th row of $\mathbf{S}^b$ is $\bar{\mathbf{s}}_l^b$ for $b \in \{0,1\}$.
2. Pick a set $\mathcal{J}$ s.t. $\mathcal{J} \cap \mathrm{ID} = \emptyset$ and $|\mathcal{J}| = \gamma + 2s$. We assume w.l.o.g. that the elements of $\mathcal{J}$ are $[\gamma + 2s]$. For $k \in \mathcal{J}$ let the column vector of $\mathbf{S}^0, \mathbf{S}^1$ be $\mathbf{s}_k^0$, respectively $\mathbf{s}_k^1$. For $b \in \{0,1\}$, P_i lets $\mathbf{t}_k^b = \pi_m(\mathbf{s}_k^b)$ and lets $\mathbf{t}_k = \mathbf{t}_k^0 + \mathbf{t}_k^1$. Also P_j lets $\mathbf{w}_k = (w_k^1, w_k^2, \ldots, w_k^n)$ and $\mathbf{b} = (b_1, b_2, \ldots, b_n)$ where $w_k^l = \mathbf{s}_k^{b_l}[l]$ for $l \in [n]$.
3. For $\mathcal{J}$, P_i lets $\mathbf{c}_k^0 = \pi_{m,n}(\mathbf{s}_k^0)$ and $\mathbf{c}_k^1 = \pi_{m,n}(C(\mathbf{t}_k)) - \mathbf{c}_k^0$. It then computes the correction value $\bar{\mathbf{c}}_k = \mathbf{c}_k^1 - \pi_{m,n}(\mathbf{s}_k^1)$.
4. Finally P_i sends the set $\{\bar{\mathbf{c}}_k\}_{j\in\mathcal{J}}$ to P_j. For $l \in [n-m]$ if $b_{m+l} = 1$, P_j updates $w_k^{m+l} := \bar{\mathbf{c}}_k[l] + w_k^{m+l}$.

Consistency Check:

5. For each $q \in [2s]$ P_j samples $r_1^q, \ldots, r_\gamma^q \in_R \mathbb{F}$ and sends these to P_i.
6. P_i then computes

$$\tilde{\mathbf{t}}_q^0 = \mathbf{t}_{\gamma+q}^0 + \sum_{k=1}^{\gamma} r_k \cdot \mathbf{t}_k^0 \quad \tilde{\mathbf{t}}_q^1 = \mathbf{t}_{\gamma+q}^1 + \sum_{k=1}^{\gamma} r_k \cdot \mathbf{t}_k^1 \quad \tilde{\mathbf{c}}_q^0 = \mathbf{c}_{\gamma+q}^0 + \sum_{k=1}^{\gamma} r_k \cdot \mathbf{c}_k^0$$

   and sends $(\tilde{\mathbf{t}}_q^0, \tilde{\mathbf{t}}_q^1, \tilde{\mathbf{c}}_q^0)$ to P_j for each $q \in [2s]$.
7. For each $q \in [2s]$ P_j computes $\tilde{\mathbf{w}}_q = \mathbf{w}_{\gamma+q} + \sum_{k=1}^{\gamma} r_k \cdot \mathbf{w}_k$. It lets $\tilde{\mathbf{c}}_q = \pi_{m,n}(C(\tilde{\mathbf{t}}_q^0 + \tilde{\mathbf{t}}_q^1))$ and $\tilde{\mathbf{c}}_g^1 = \tilde{\mathbf{c}}_g - \tilde{\mathbf{c}}_g^0$. Finally for $u \in [m]$ and $v \in [n-m]$, P_j verifies that $\tilde{\mathbf{t}}_q^{b_u}[u] = \tilde{\mathbf{w}}_q[u]$ and $\tilde{\mathbf{c}}_q^{b_{m+v}}[v] = \tilde{\mathbf{w}}_q[m+v]$.
   If the above check fails P_j outputs abort and halts.

Output:

8. Both parties let $\mathrm{ID} = \mathrm{ID} \cup \mathcal{J} \setminus \{\gamma + g\}_{q\in[2s]}$ and $\mathrm{U} = \mathrm{U} \cup \mathcal{J} \setminus \{\gamma + q\}_{q\in[2s]}$. P_i now holds opening information $\{(\mathbf{t}_k^0, \mathbf{t}_k^1, \mathbf{c}_k^0)\}_{k\in\mathcal{J}}$ and P_j holds the verifying information $\{\mathbf{w}_k\}_{k\in\mathcal{J}\setminus\{\gamma+q\}_{q\in[2s]}}$. P_i outputs $(\mathtt{random}, \mathcal{J} \setminus \{\gamma+q\}_{q\in[2s]}, \{\mathbf{t}_k\}_{k\in\mathcal{J}\setminus\{\gamma+q\}_{q\in[2s]}})$ and P_j outputs $(\mathtt{random}, \mathcal{J} \setminus \{\gamma + q\}_{q\in[2s]})$.[9]

Fig. 15. Protocol UC-realizing $\mathcal{F}'_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$ in the $\mathcal{F}_{\mathsf{OT}}$-hybrid model – part 1.

Online Usage. To use the Schur pairs to facilitate multiplication of a public message vector some interaction is required. We describe the protocol for achieving this in Fig. 17. Basically if we wish to multiply the public constant vector $\boldsymbol{\alpha} \in \mathbb{F}^m$ with a commitment to $\mathbf{x}$, using a Schur Pair $(C(\mathbf{x}), C^*(\mathbf{x}\|\mathbf{x}'))$, we use the linearity of the code and the fact that the commitment consists of an additive secret sharing and let P_i compute $C(\boldsymbol{\alpha}) * C(\mathbf{y})$ by doing component wise multiplication of $C(\boldsymbol{\alpha})$ onto the shares committing to $\mathbf{y}$, resulting in an element in C^*. We then hide the result of this by subtracting $C^*(\mathbf{x}\|\mathbf{x}')$, using the fact that C^* has message length m^* and thus that $\mathbf{x}'$ will be used to hide any info on $\boldsymbol{\alpha} * \mathbf{y}$ which might otherwise be leaked by the last $m^* - m$ message components of $C^*(\boldsymbol{\alpha} * \mathbf{y})$. Thus P_i will open the message $\boldsymbol{\epsilon} = \boldsymbol{\alpha} * \mathbf{y} - \mathbf{x}$. P_j verifies that the opening is correct and then adjusts the commitment $C(\mathbf{x})$ by adding ϵ s.t. the values $\mathbf{x}$ from C and C^* cancel out and what remains is a commitment $\boldsymbol{\alpha} * \mathbf{y}$ using C.

Security. As we have augmented the protocol with the procedure **Pair** we need to prove that this augmentation is secure. To do so we first define the ideal functionality of the extra commands **Pair** command and then **Public Multiplication**:

Pair: On input $(\mathtt{pair}, K)$ from all parties where K is a set of size $\nu + s$ for some ν partition K into two sets X and R where $|X| = \nu$ and $|R| = s$. Store the tuple $(\mathtt{pair}, k)$ for each $k \in X$ and output $(\mathtt{pair}, X)$ to P_i and P_j.

Public Multiplication: On input $(\mathtt{mult}, k, \boldsymbol{\alpha})$ from P_i where $\mathtt{actual}[k] = \mathbf{x}_k \neq \perp$, some message $(\mathtt{pair}, k')$ is stored and $\boldsymbol{\alpha} \in \mathbb{F}^m$ then send $(\mathtt{mult}, k, \boldsymbol{\alpha})$ to P_j and set $\mathtt{actual}[k'] = \boldsymbol{\alpha} * \mathbf{x}_k$, $\mathtt{actual}[k] = \perp$ and delete $(\mathtt{pair}, k')$.

Input:

1. On input $(\texttt{Input}, k, \mathbf{y})$ from P_i let P_i compute $\tilde{\mathbf{y}}_k = \mathbf{y} - \mathbf{t}_k$ and send $(\texttt{chosen}, k, \tilde{\mathbf{y}}_k)$ to P_j. Else ignore the message.
2. P_j stores $(\texttt{chosen}, k, \tilde{\mathbf{y}}_k)$ and both parties set $\mathrm{U} = \mathrm{U}\backslash\{k\}$.

Open:

1. On input $(\texttt{open}, \{(k, \alpha_k)\}_{k\in K})$ where each $\alpha_k \in \mathbb{F}$ and for all $k \in K$, P_i holds $(\mathbf{t}_k^0, \mathbf{t}_k^1, \mathbf{c}_k^0)$ then it computes
$$\bar{\mathbf{t}}^0 = \sum_{k\in K} \alpha_k \cdot \mathbf{t}_k^0, \quad \bar{\mathbf{t}}^1 = \sum_{k\in K} \alpha_k \cdot \mathbf{t}_k^1, \quad \bar{\mathbf{c}}^0 = \sum_{k\in K} \alpha_k \cdot \mathbf{c}_k^0$$
and sends $(\texttt{opening}, \{(k, \alpha_k)\}_{k\in K}, (\mathbf{t}^0, \mathbf{t}^1, \mathbf{c}^0))$ to P_j. Else it ignores the input message.
2. Upon receiving the message $(\texttt{opening}, \{(k, \alpha_k)\}_{k\in K}, (\mathbf{t}^0, \mathbf{t}^1, \mathbf{c}^0))$ from P_i, if for all $k \in K$, P_j holds $\mathbf{w}_k$ it lets $\mathbf{t} = \mathbf{t}^0 + \mathbf{t}^1$ and computes $\mathbf{w} = \sum_{k\in K} \alpha_k \cdot \mathbf{w}_k$. It lets $\mathbf{c} = \pi_{m,n}(C(\mathbf{t}))$ and computes $\mathbf{c}^1 = \mathbf{c} - \mathbf{c}^0$.
Finally for $u \in [m]$ and $v \in [n-m]$, P_j verifies that
$$\mathbf{t}^{b_u}[u] = \mathbf{w}[u]\ ,\ \mathbf{c}^{b_{m+v}} = \mathbf{w}[m+v]\ .$$
3. If all checks are valid set $\mathrm{U} = \mathrm{U}\backslash C$ and output $(\texttt{opened}, \{(k, \alpha_k)\}_{k\in K}, \mathbf{t} + \sum_{k\in K} \alpha_k \cdot \tilde{\mathbf{y}}_k)$ where $K \subseteq \mathrm{ID}\backslash\mathrm{U}$ s.t. for each $k \in K$ P_j has stored a message $(\texttt{chosen}, k, \tilde{\mathbf{y}}_k)$. Else it aborts and halts.

Public Multiplication:

1. On input $(\texttt{mult}, \boldsymbol{\alpha}, k)$ where $\boldsymbol{\alpha} \in \mathbb{F}^m$, P_i holds $(\mathbf{t}_k^0, \mathbf{t}_k^1, \mathbf{c}_k^0)$, P_j holds $\mathbf{w}_k$ and has stored $(\texttt{chosen}, k, \tilde{\mathbf{y}}_k)$. Further, both parties has a messages $(\texttt{pair}, l)$ stored and thus P_i holds $((\mathbf{t}_l^0, \mathbf{t}_l^1, \mathbf{c}_l^0), (\mathbf{t}_l^{*0}, \mathbf{t}_l^{*1}, \mathbf{c}_l^{*0}), \bar{\mathbf{t}}_l)$ and P_j holds $(\mathbf{w}_l, \mathbf{w}_l^*, \bar{\mathbf{t}}_l)$. P_i computes
$$\begin{aligned}\hat{\mathbf{t}}_l^{*0} &= -\mathbf{t}_l^{*0} + \pi_{m^*}(C(\boldsymbol{\alpha})) * (\mathbf{t}_k^0 \| \pi_{m^*-m}(\mathbf{c}_k^0)),\\ \hat{\mathbf{t}}_l^{*1} &= -\mathbf{t}_l^{*1} + \pi_{m^*}(C(\boldsymbol{\alpha})) * (\mathbf{t}_k^1 \| \pi_{m^*-m}(\mathbf{c}_k^1)),\\ \hat{\mathbf{c}}_l^{*0} &= -\mathbf{c}_l^{*0} + \pi_{m^*,n}(C(\boldsymbol{\alpha})) * \pi_{m^*,n}(\mathbf{c}_k^0)\end{aligned}$$
and sends $(\hat{\mathbf{t}}_l^{*0}, \hat{\mathbf{t}}_l^{*1}, \hat{\mathbf{c}}_l^{*0})$ to P_j.
2. Upon receiving $(\hat{\mathbf{t}}_l^{*0}, \hat{\mathbf{t}}_l^{*1}, \hat{\mathbf{c}}_l^{*0})$ from P_i, P_j then defines $\hat{\mathbf{t}}_l^* = \hat{\mathbf{t}}_l^{*0} + \hat{\mathbf{t}}_l^{*1}$ and lets $\hat{\mathbf{c}}^* = \pi_{m^*,n}(C^*(\hat{\mathbf{t}}_l^*))$ and $\hat{\mathbf{c}}^{*1} = \hat{\mathbf{c}}^* - \hat{\mathbf{c}}^{*0}$.
3. P_j then verifies for $u \in [m^*]$ and $v \in [n-m^*]$ that $\hat{\mathbf{t}}_l^{*b_u}[u] = -\mathbf{w}_k^*[u]_l + C(\boldsymbol{\alpha})[u] \cdot \mathbf{w}_k[u]$ and $\hat{\mathbf{c}}^{*b_v}[v] = -\mathbf{w}^*[m^*+v] + C(\boldsymbol{\alpha})[m^*+v] \cdot \mathbf{w}_k[m^*+v]$.
4. P_j computes $\tilde{\mathbf{y}}_l = \boldsymbol{\alpha} * \tilde{\mathbf{y}}_k + \pi_m(\hat{\mathbf{t}}_l^*) - \bar{\mathbf{t}}_l$ and stores the message $(\texttt{chosen}, l, \tilde{\mathbf{y}}_l)$ and both parties delete $(\texttt{pair}, l)$.

Fig. 16. Protocol UC-realizing $\mathcal{F}'_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$ in the $\mathcal{F}_{\mathsf{OT}}$-hybrid model – part 2.

Theorem A.1. *The methods **Pair** and **Public Multiplication** in Fig. 16 and Fig. 17 UC-securely realizes the **Pair** and **Public Multiplication** functionalities described above against a static and malicious adversary.*

Proof. Since the two methods are simply extensions to the functionality $\mathcal{F}'_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$, and $\mathcal{F}'_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$ is realized exactly as in [18] we will piggyback a lot the security proof in this paper and assume the reader is very familiar with that proof.

We start by showing correctness. This is straight forward, but we write it out for completeness: **Pair:** We show that after steps 1-7 have been completed, if both parties were honest, then the three checks in step 8 should pass and it should also hold (in order to make the whole protocol work) that $\mathbf{t}_k^0 + \mathbf{t}_k^1 = \pi_m(\mathbf{t}_k^{*0} + \mathbf{t}_k^{*1}) + \bar{\mathbf{t}}_k^*$ for $k \in [v+s]$. For the first two parts of 8 we see that these follow directly from correctness of opening of linear commitments, since this is

Pair: Upon receiving a message $(\texttt{pair}, K)$ from all parties where $K \subseteq \mathrm{U}$ of size $\nu + s$ for some ν. Then partition this into the sets X, R where $|X| = \nu$ and $|R| = s$. Proceed as follows:

1. The parties execute $(\texttt{commit}, \nu + s)$ but using the code C^*, and all the m^* seed OTs from **Init**, instead of just the first m.
2. Based on the result of the **Commit** phase with C^*, denote the tuple of opening information held by P_i as $\{(\mathbf{t}_k^{*0}, \mathbf{t}_k^{*1}, \mathbf{c}_k^{*0})\}_{k\in[\nu+s]}$. Similarly denote the verification info held by P_j as $\{\mathbf{w}_k^*\}_{k\in[\nu+s]}$. Partition $[\nu + s]$ into two sets X' and R' where $|X'| = \nu$ and $|R'| = s$.
3. For each of the $k \in [\nu]$ party P_i computes $\bar{\mathbf{t}}_{X[k]} = \mathbf{t}_{X[k]} - \pi_m(\mathbf{t}^*_{X[k]})$ using the opening information for the commitments based on both C^* and C. P_i then sends $\{\bar{\mathbf{t}}_{X[k]}\}_{k\in[\nu]}$ to P_j.
4. For each $q \in [s]$ party P_i computes $\bar{\mathbf{t}}_{R[q]} = \mathbf{t}_{R[q]} - \pi_m(\mathbf{t}^*_{R'[q]})$ using the opening information for the commitments based on both C^* and C and sends $\{\bar{\mathbf{t}}_{R[q]}\}_{q\in[s]}$ to P_j.
5. P_i and P_j input $(\texttt{toss}, \nu, \mathbb{F})$ to $\mathcal{F}_{\mathsf{CT}}$ for each $q \in [s]$ and thus learn $(\texttt{random}, \mathbf{r}_q)$ (when viewing the output as a vector $\mathbf{r}_q \in \mathbb{F}^\nu$).
6. P_i now opens the linear combination for each $q \in [s]$ by sending the following values to P_j:

$$\tilde{\mathbf{t}}_q^0 = \mathbf{t}_{R[q]}^0 + \sum_{k\in[\nu]} \mathbf{r}_q[k]\cdot(\mathbf{t}_{X[k]}^0), \qquad \tilde{\mathbf{t}}_q^1 = \mathbf{t}_{R[q]}^1 + \sum_{k\in[\nu]} \mathbf{r}_q[k]\cdot(\mathbf{t}_{X[k]}^1),$$
$$\tilde{\mathbf{c}}_q^0 = \mathbf{c}_{R[q]}^0 + \sum_{k\in[\nu]} \mathbf{r}_q[k]\cdot(\mathbf{c}_{X[k]}^0), \qquad \tilde{\mathbf{t}}_q^{*0} = \mathbf{t}_{R'[q]}^{*0} + \sum_{k\in[\nu]} \mathbf{r}_q[k]\cdot(\mathbf{t}_{X'[k]}^{*0})$$
$$\tilde{\mathbf{t}}_q^{*1} = \mathbf{t}_{R'[q]}^{*1} + \sum_{k\in[\nu]} \mathbf{r}_q[k]\cdot(\mathbf{t}_{X'[k]}^{*1}), \qquad \tilde{\mathbf{c}}_q^{*0} = \mathbf{c}_{R'[q]}^{*0} + \sum_{k\in[\nu]} \mathbf{r}_q[k]\cdot(\mathbf{c}_{X'[k]}^{*0})$$

7. For each $q \in [s]$ P_j now computes $\tilde{\mathbf{w}}_q = \mathbf{w}_{R[q]} + \sum_{k\in[\nu]} \mathbf{r}_q[k]\cdot \mathbf{w}_{X[k]}$ and $\tilde{\mathbf{w}}_q^* = \mathbf{w}^*_{R'[q]} + \sum_{k\in[\nu]} \mathbf{r}_q[k]\cdot \mathbf{w}^*_{X'[k]}$. It lets $\tilde{\mathbf{c}}_q = \pi_{m,n}(C(\tilde{\mathbf{t}}_q^0 + \tilde{\mathbf{t}}_q^1))$ and $\tilde{\mathbf{c}}_q^* = \pi_{m,n}(C(\tilde{\mathbf{t}}_q^{*0} + \tilde{\mathbf{t}}_q^{*1}))$. It then computes $\tilde{\mathbf{t}}_q = \tilde{\mathbf{t}}_q^0 + \tilde{\mathbf{t}}_q^1$, $\tilde{\mathbf{c}}_q = \tilde{\mathbf{c}}_q^0 + \tilde{\mathbf{c}}_q^1$, $\tilde{\mathbf{t}}_q^* = \tilde{\mathbf{t}}_q^{*0} + \tilde{\mathbf{t}}_q^{*1}$ and $\tilde{\mathbf{c}}_q^* = \tilde{\mathbf{c}}_q^{*0} + \tilde{\mathbf{c}}_q^{*1}$.
8. P_j verifies the following:
   - That for each $u \in [m]$ and $v \in [n-m]$ it is the case $\tilde{\mathbf{t}}_q^{b_u}[u] = \tilde{\mathbf{w}}_q[u]$ and $\tilde{\mathbf{c}}_q^{b_{m+v}}[v] = \tilde{\mathbf{w}}_q[m+v]$.
   - That for each $u \in [m^*]$ and $v \in [n-m^*]$ it is the case $\tilde{\mathbf{t}}_q^{*b_u}[u] = \tilde{\mathbf{w}}_q^*[u]$ and $\tilde{\mathbf{c}}_q^{*b_{m^*+v}}[v] = \tilde{\mathbf{w}}_q^*[m^*+v]$.
   - That $\tilde{\mathbf{t}}_q - \pi_m\left(\tilde{\mathbf{t}}_q^*\right) + \bar{\mathbf{t}}_{R[q]} + \sum_{k\in[\nu]} \mathbf{r}_q[k]\cdot\bar{\mathbf{t}}_{X[k]} = \mathbf{0}^m$.

   If any check fails then P_j aborts.
9. Both parties store and output the messages $\{(\texttt{pair}, X[k])\}_{k\in[\nu]}$ and set $\mathrm{U} = \mathrm{U}\backslash K$. Thus P_i holds $\{((\mathbf{t}^0_{X[k]}, \mathbf{t}^1_{X[k]}, \mathbf{c}^0_{X[k]}), (\mathbf{t}^{*0}_{X'[k]}, \mathbf{t}^{*1}_{X'[k]}, \mathbf{c}^{*0}_{X'[k]}), \bar{\mathbf{t}}_{X[k]})\}_{k\in[\nu]}$ and P_j holds $\{(\mathbf{w}_{X[k]}, \mathbf{w}^*_{X'[k]}, \bar{\mathbf{t}}_{X[k]})\}_{k\in[\nu]}$.

Fig. 17. Protocol UC-realizing $\mathcal{F}'_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$ in the $\mathcal{F}_{\mathsf{OT}}$-hybrid model – part 3.

basically what is done and thus proved in [18]. For the third part we see the following:

$$\begin{aligned}
&\tilde{\mathbf{t}}_q - \pi\left(\tilde{\mathbf{t}}_q^*\right) - \bar{\mathbf{t}}_{R[q]} - \sum_{k\in[\nu]} \mathbf{r}_q[k]\cdot\bar{\mathbf{t}}_{X[k]} = \tilde{\mathbf{t}}_q^0 + \tilde{\mathbf{t}}_q^1 - \pi\left(\tilde{\mathbf{t}}_q^{*0} + \tilde{\mathbf{t}}_q^{*1}\right) - \bar{\mathbf{t}}_{R[q]} - \sum_{k\in[\nu]} \mathbf{r}_q[k]\cdot\bar{\mathbf{t}}_{X[k]} \\
&= \tilde{\mathbf{t}}_q^0 + \tilde{\mathbf{t}}_q^1 - \pi\left(\mathbf{t}^{*0}_{R'[q]} + \mathbf{t}^{*1}_{R'[q]} + \left(\sum_{k\in[\nu]} \mathbf{r}_q[k]\cdot(\mathbf{t}^{*0}_{X'[k]} + \mathbf{t}^{*1}_{X'[k]})\right)\right) - \bar{\mathbf{t}}_{R[q]} - \sum_{k\in[\nu]} \mathbf{r}_q[k]\cdot\bar{\mathbf{t}}_{X[k]} \\
&= \mathbf{t}^0_{R[q]} + \mathbf{t}^1_{R[q]} + \left(\sum_{k\in[\nu]} \mathbf{r}_q[k]\cdot(\mathbf{t}^0_{X[k]} + \mathbf{t}^1_{X[k]})\right) - \\
&\qquad \pi\left(\mathbf{t}^*_{R'[q]} + \left(\sum_{k\in[\nu]} \mathbf{r}_q[k]\cdot\mathbf{t}^*_{X'[k]}\right)\right) - \bar{\mathbf{t}}_{R[q]} - \sum_{k\in[\nu]} \mathbf{r}_q[k]\cdot\bar{\mathbf{t}}_{X[k]} \\
&= \mathbf{t}_{R[q]} + \left(\sum_{k\in[\nu]} \mathbf{r}_q[k]\cdot\mathbf{t}_{X[k]}\right) - \\
&\qquad \pi\left(\mathbf{t}^*_{R'[q]} + \left(\sum_{k\in[\nu]} \mathbf{r}_q[k]\cdot\mathbf{t}^*_{X'[k]}\right)\right) - \mathbf{t}_{R[q]} - \pi_m(\mathbf{t}^*_{R'[q]}) + \sum_{k\in[\nu]} \mathbf{r}_q[k]\cdot\left(-\mathbf{t}_{X[k]} + \pi_m(\mathbf{t}^*_{X'[k]})\right) \\
&= \mathbf{0}^m
\end{aligned}$$

To verify that $\mathbf{t}^0_{X[k]} + \mathbf{t}^1_{X[k]} = \pi_m(\mathbf{t}^{*0}_{X'[k]} + \mathbf{t}^{*1}_{X'[k]}) + \bar{\mathbf{t}}_k$ for $k \in [v]$, (and identically for the sets R, R' with $k \in [s]$) observe the following:

$$\begin{aligned}\pi_m\left(\mathbf{t}^{*0}_{X'[k]} + \mathbf{t}^{*1}_{X'[k]}\right) + \bar{\mathbf{t}}_{X[k]} &= \pi_m\left(\mathbf{t}^{*}_{X'[k]}\right) + \mathbf{t}_{X[k]} - \pi_m(\mathbf{t}^{*}_{X'[k]})\\ &= \mathbf{t}_{X[k]}\end{aligned}$$

Finally we observe that by definition $\mathbf{t}_{X[k]} = \mathbf{t}^0_{X[k]} + \mathbf{t}^1_{X[k]}$ and correctness follows.

Public Multiplication: The values verified in step 3 are trivially true by correctness of opening of commitments. For easier readability we recast the index of the C^* encoded part of the prepocessed pair being used to be the same as for the elements in C an thus distinguish between the two only through the superscript $*$. We see that it suffices to show that $\mathbf{t}_l + \tilde{\mathbf{y}}_l = \boldsymbol{\alpha} * \mathbf{y}$ as this is the value that will be opened and learned by P_j. We do this as follows:

$$\begin{aligned}\mathbf{t}_l + \tilde{\mathbf{y}}_l = \mathbf{t}_l + \pi_m(\hat{\mathbf{t}}^*_l) - \bar{\mathbf{t}}_l &= \mathbf{t}_l + \boldsymbol{\alpha} * \tilde{\mathbf{y}}_k + \pi_m(\hat{\mathbf{t}}^{*0}_l + \hat{\mathbf{t}}^{*1}_l) - \bar{\mathbf{t}}^*_l\\ &= \mathbf{t}_l + \boldsymbol{\alpha} * \tilde{\mathbf{y}}_k + \pi_m\left(-\mathbf{t}^{*0}_l - \mathbf{t}^{*1}_l + \pi_{m^*}(C(\boldsymbol{\alpha})) * \left(\mathbf{t}^0_k \| \pi_{m^*-m}(\mathbf{c}^0_k) + \mathbf{t}^1_k \| \pi_{m^*-m}(\mathbf{c}^1_k)\right)\right) - \bar{\mathbf{t}}_l\\ &= \mathbf{t}_l + \boldsymbol{\alpha} * \tilde{\mathbf{y}} + \boldsymbol{\alpha} * (\mathbf{t}^0_k + \mathbf{t}^1_k) + \pi_m\left(-\mathbf{t}^{*0}_l - \mathbf{t}^{*1}_l\right) - \bar{\mathbf{t}}_l\\ &= \mathbf{t}_l + \boldsymbol{\alpha} * (\tilde{\mathbf{y}}_k + \mathbf{t}_k) + \pi_m(-\mathbf{t}^*_l) - \bar{\mathbf{t}}_l\\ &= \mathbf{t}_l + \boldsymbol{\alpha} * (\tilde{\mathbf{y}}_k + \mathbf{t}_k) + \pi_m(-\mathbf{t}^*_l) - \mathbf{t}_l + \pi_m(\mathbf{t}^*_l)\\ &= \boldsymbol{\alpha} * (\tilde{\mathbf{y}}_k + \mathbf{t}_k) = \boldsymbol{\alpha} * (\mathbf{y} - \mathbf{t}_k + \mathbf{t}_k) = \boldsymbol{\alpha} * \mathbf{y}\end{aligned}$$

Which verifies that the protocol is correct.

Security: First notice that the elements preprocessed in the **Pair** method can only used in **Public Multiplication**, and thus not be opened individually or reused. Keeping this in mind, we now prove security in two steps, first assuming a corrupt receiver, P_j and next assuming a corrupt sender, P_i. If both parties are corrupt there is nothing to show.

Before continuing with the proof we first describe the security intuition: Basically the security of the **Init**, **Commit** and **Open** commands follows directly from [18]. The security of the remaining commands rely on the following main observation (from [18]): In the hybrid world it is sufficient to argue that as there is always one, free random variable in the partial sum of some value sent to the receiver from the simulator then the simulation and true execution are indistinguishable (assuming all other correlations are obeyed). Specifically they means that as long as one watch component in $\mathbf{t}$ is freely chosen at random the simulation is sound.

Now we consider the actual proof. We use $\mathcal{A}$ to denote the corrupted receiver. For **Init**, **Commit** we do simulation as in [18]. In this simulation we have that the simulator, $\mathcal{S}$, simulated the $\mathcal{F}_{\mathsf{OT}}$ functionality in **Init** and so it learns $\mathcal{A}$'s choicebits b_l for $l \in [m^*]$ along with its watchbits $\mathbf{w}_k$. With this in mind, the simulation proceeds as follows:

Input:

1. Pick a random value $\tilde{\mathbf{y}}_k \in \mathbb{F}^m$ and sends the message $(\mathtt{chosen}, k, \tilde{\mathbf{y}}_k)$ to $\mathcal{A}$ and internally store $(\mathtt{chosen}, k, \tilde{\mathbf{y}}_k)$.

Pair:

1-2. Pass on the input $(\mathtt{pair}, K)$ to $\mathcal{F}'_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$ and receives back $(\mathtt{pair}, X')$. Emulate **Commit** for C^* as in [18].
3. For each $k \in [v]$ pick a uniformly random value $\bar{\mathbf{t}}_{X[k]} \in \mathbb{F}^m$ and send these to $\mathcal{A}$.
4. For each $q \in [s]$ pick a uniformly random value $\bar{\mathbf{t}}_{R[q]} \in \mathbb{F}^m$ and send these to $\mathcal{A}$.
5. Emulate $\mathcal{F}_{\mathsf{CT}}$ by picking random values $\mathbf{r}_q \in \mathbb{F}^v$ for $q \in [s]$.

6. For $q \in [s]$ pick $\tilde{\mathbf{t}}_q \in \mathbb{F}^m$ and $\tilde{\mathbf{t}}^*_q \in \mathbb{F}^{m^*}$ uniformly at random. Then compute the following values:

$$\begin{aligned}
\text{For } l \in [m]: \quad & \tilde{\mathbf{t}}_q^{b_l}[l] &&= \mathbf{w}_{R[q]}[l] + \sum_{k \in [v]} \mathbf{r}_q[k] \cdot (\mathbf{w}_{X[k]}[l]),\\
& \tilde{\mathbf{t}}_q^{1-b_l}[l] &&= \tilde{\mathbf{t}}_q[l] - \tilde{\mathbf{t}}_q^{b_l}[l],\\
\text{for } l \in [n-m]: \quad & \tilde{\mathbf{c}}_q^{b_{m+l}}[m+l] &&= \mathbf{w}_{R[q]}[m+l] + \sum_{k \in [v]} \mathbf{r}_q[k] \cdot \mathbf{c}^0_{X[k]},\\
& \tilde{\mathbf{c}}_q^{1-b_{m+l}}[m+l] &&= C(\tilde{\mathbf{t}}_q)[m+l] - \tilde{\mathbf{c}}_q^{b_{m+l}}[m+l],\\
\text{for } l \in [m^*]: \quad & \tilde{\mathbf{t}}_q^{*b_l}[l] &&= \mathbf{w}^*_{R'[q]}[l] + \sum_{k \in [v]} \mathbf{r}_q[k] \cdot (\mathbf{w}^*_{X'[k]}[l]),\\
& \tilde{\mathbf{t}}_q^{*1-b_l}[l] &&= \tilde{\mathbf{t}}^*_q[l] - \tilde{\mathbf{t}}_q^{*b_l}[l],\\
\text{for } l \in [n-m^*]: \quad & \tilde{\mathbf{c}}_q^{*b_{m^*+l}}[m^*+l] &&= \mathbf{w}^*_{R'[q]}[m^*+l] + \sum_{k \in [v]} \mathbf{r}_q[k] \cdot \mathbf{w}^*_{X'[q]}[m^*+l],\\
& \tilde{\mathbf{c}}_q^{1-b_{m^*+l}}[m^*+l] &&= C(\tilde{\mathbf{t}}_q)[m^*+l] - \tilde{\mathbf{c}}_q^{*b_{m^*+l}}[m^*+l]
\end{aligned}$$

   Send $\tilde{\mathbf{t}}^0_q, \tilde{\mathbf{t}}^1_q, \tilde{\mathbf{c}}^0_q$ and $\tilde{\mathbf{t}}^{*0}_q, \tilde{\mathbf{t}}^{*1}_q, \tilde{\mathbf{c}}^{*0}_q$ to $\mathcal{A}$.

7-9. If $\mathcal{A}$ did not abort output $\{\texttt{pair}, X[k]\}_{k \in [v]}$.

Public Multiplication:

1. On input $(\texttt{mult}, \boldsymbol{\alpha}, k)$ with $\boldsymbol{\alpha} \in \mathbb{F}^m$ and messages $(\texttt{pair}, l)$ and $(\texttt{chosen}, k, \tilde{\mathbf{y}}_k)$ stored, retrieve values $((\mathbf{t}^0_l, \mathbf{t}^1_l, \mathbf{c}^0_l), (\mathbf{t}^{*0}_l, \mathbf{t}^{*1}_l, \mathbf{c}^{*0}_l), \bar{\mathbf{t}}_l)$ computed in the simulation of **Pair** for $(\texttt{pair}, l)$. Then use these messages to compute and send $\hat{\mathbf{t}}^{*0}_l, \hat{\mathbf{t}}^{*1}_l$ and $\hat{\mathbf{c}}^{*0}_l$ as the true sender would.

2-4. Delete $(\texttt{pair}, l)$ and compute $\tilde{\mathbf{y}}_l = \boldsymbol{\alpha} * \tilde{\mathbf{y}}_k + \pi_m(\hat{\mathbf{t}}^{*0}_l + \hat{\mathbf{t}}^{*1}_l) - \bar{\mathbf{t}}_l$ and store $(\texttt{chosen}, l, \tilde{\mathbf{y}}_l)$.

We now argue indistinguishability of these simulations and the real execution. We start with the method **Input**. See the value $\tilde{\mathbf{y}}_k$ of $(\texttt{chosen}, k, \tilde{\mathbf{y}}_k)$ is indistinguishable with what is sent in the real protocol. To see this first notice that real protocol this value is indistinguishable from random since for each $l \in [m]$ the value $\mathbf{t}^{1-b_l}_k[l]$ will be unknown to $\mathcal{S}$ because the $\mathcal{F}_{\mathsf{OT}}$ used is ideal and thus he will have no knowledge of the seed used in the PRG to compute $\mathbf{t}^{1-b_k}_k[l]$. Meaning $\mathbf{t}^{1-b_k}_k[l]$ is indistinguishable from a random element in $\mathbb{F}$. Thus the value sent in the real protocol from P_i to P_j is indistinguishable from a random element in $\mathbb{F}^m$. So the real and ideal world are clearly indistinguishable.

Next consider the **Pair** method and see that by the security of [18] the simulation in step 1 and 2 is sound. Next we see, piggy backing on the proof of [18] that either $\mathbf{t}^0_{X[k]}$ or $\mathbf{t}^1_{X[k]}$ depending on choice of watchbit. (The same for either $\mathbf{t}^{*0}_{X'[k]}$ or $\mathbf{t}^{*1}_{X'[k]}$). This means that in the real execution $\bar{\mathbf{t}}_{X[k]}$ and $\bar{\mathbf{t}}_{R[k]}$ will both be indistinguishable from random. Thus values sent in step 3 and 4 are indistinguishable between the simulation and a real execution. The same goes for the emulation of $\mathcal{F}_{\mathsf{CT}}$. Finally, see that all values sent to $\mathcal{A}$ in step 6 are either uniformly random or completely determined, both in its view in the real execution and the simulation. In the real execution we see this since values indexed by $R[q]$ for $q \in [s]$ acts as one-time paddings that will never be used again. Thus when considering a component that was not chosen by $\mathcal{A}$ to be watched, the will be completely unknown to it. However, when it has been chosen it must be consistent with its choices of watchbit in order to be indistinguishable. We notice that for exactly the watch components $\mathcal{A}$ will learn exactly the value he would expect in the real execution. For the non-watch components these are instead uniformly random. One exception being the value $\tilde{\mathbf{c}}^0_{R[q]}$ which is computed to be the correct parity bits in accordance with the random codewords. But since $\tilde{\mathbf{c}}^0_{R[q]}$ is picked uniformly at random, and $\mathcal{A}$ will, for the l'th component learn either a uniformly random value, or a specific value one-time padded with a uniformly random value, i.e. $\tilde{\mathbf{c}}^0_{R[q]}[l]$ or $\tilde{\mathbf{c}}^1_{R[q]}[l] = C(\tilde{\mathbf{t}}_q)[m+l] - \tilde{\mathbf{c}}^0_{R[q]}[l]$. Further see, that this also means that we can switch the semantic meaning of $\tilde{\mathbf{c}}^0_{R[q]}[l]$ or $\tilde{\mathbf{c}}^1_{R[q]}[l]$, as each of them, on it own, is indistinguishable from a random value.

For the **Public Multiplication** we see that each component of $\hat{\mathbf{t}}^{*0}_l$ or $\hat{\mathbf{t}}^{*1}_l$ is indistinguishable from random in the view of $\mathcal{A}$ in the real execution following the construction and proof of **Pair**. The same argument follows for the components of $\hat{\mathbf{c}}^{*0}_l$

For the case of **Open** we do almost the same as in the proof in [18]. However, since we now might have a message $(\texttt{chosen}, k, \tilde{\mathbf{y}}_k)$ to add to the opening we need to make some slight changes in the simulation:

When receiving $(\texttt{opened}, \{(k, \alpha_k)\}_{k\in K}, \mathbf{x})$ from the ideal functionality we must simulate the triple $(\bar{\mathbf{t}}^0, \bar{\mathbf{t}}^1, \bar{\mathbf{c}}^0)$ sent to $\mathcal{A}$. We use the fact that in the real protocol P_j can recompute all the values received from P_i given just the value $\mathbf{x}$ and the values $\{(k, \alpha_k, \tilde{\mathbf{y}}_k, \mathbf{w}_k)\}_{k\in K}$, which it already knows. Specifically we compute $\mathbf{w} = \sum_{k\in K} \alpha_k \cdot \mathbf{w}_k$ and $\mathbf{t} = C(\mathbf{x} - \sum_{k\in K} \alpha_k \cdot \tilde{\mathbf{y}}_k)$ and $\mathbf{c} = \pi_{m,n}(\mathbf{t})$ where the values $\tilde{\mathbf{y}}_k$ are retrieved from the messages $(\texttt{chosen}, k, \tilde{\mathbf{y}}_k)$. Then for $u \in [m]$ and $v \in [n-m]$ we define $\bar{\mathbf{t}}^{b_u}[u] = \mathbf{w}[u]$, $\bar{\mathbf{c}}^{b_{m+v}}[v] = \mathbf{w}[m+v]$, $\bar{\mathbf{t}}^{1-b_u}[u] = \bar{\mathbf{t}}[u] - \bar{\mathbf{t}}^{b_u}[u]$ and $\bar{\mathbf{c}}^{1-b_{m+v}}[v] = \mathbf{c}[v] - \bar{\mathbf{c}}^{b_{m+v}}[v]$. Which follows from the fact $\mathbf{t} = \mathbf{t}^0 + \mathbf{t}^1$ and $\mathbf{c} = \mathbf{c}^0 + \mathbf{c}^1$. We then sent the triple $(\bar{\mathbf{t}}^0, \bar{\mathbf{t}}^1, \bar{\mathbf{c}}^0)$ just computed to $\mathcal{A}$.

The argument of indistinguishably is the same as in [18]; basically because $\mathcal{A}$ will be oblivious of one value in each component, and this value is indistinguishable from random in the real world (because we use an ideal OT and a PRG and thus he will learn nothing of the choice he does not make). This is also the case in our simulation since the value $\mathbf{x}$ will be a linear combination of at least one uniformly random value, which is also uniformly random. Or it will be a linear combination of a chosen value, in this case what we send to $\mathcal{A}$ will be in correspondence with the actual chosen value so that if $\mathcal{A}$ is honest, it will learn the same value as in the real world. Furthermore, following the arguments above the other values sent to $\mathcal{A}$ will be indistinguishable from what is sent in the real world.

Now we consider a malicious P_i and denote this by $\mathcal{A}$ and thus $\mathcal{S}$ will simulate an honest P_j. For the methods **Init, Commit** and **Open** we basically piggyback on the proof of [18]. For **Input** we notice that we can simulate this perfectly since we can extract the random commitments $\mathcal{A}$ is uniquely defined to be able to open (by the proof in [18]) and then simply compute the true commitment by adding to this the correction value it sends.

Next we see that for the methods **Pair** and **Public Multiplication** P_j never sends anything, thus we can trivially simulate this. Since P_j does not have any input to the protocol, what is left to show is that the ideal output is equal to the real output in the case of public multiplication. For random commitments this is done by extracting the "actual" values committed to by $\mathcal{A}$ and use them as input to the ideal functionality. In case of public multiplication this means that we must ensure that the value opened in the ideal functionality is the same as the one opened in the real execution. Specifically, we show that $\mathcal{A}$ can only succeed in opening a wrong public multiplication commitment if it can guess at least s uniformly random bits.

First see that we can extract all the random messages P_i commits to using the proof in [18]. This is the case for both the messages using C and C^*. Based on these messages we can compute which values an honest P_i should send. So far we don't abort if $\mathcal{A}$ sends something wrong and we compute everything like an honest P_j would. Now, when we reach step 8 we must argue that if $\mathcal{A}$ send something different than it was supposed to, an honest P_j will catch him. If it sent all the right things, then by the correctness of the protocol and the element we extracted (and passed on to the ideal functionality) the openings in the real and ideal worlds will be consistent.

We notice that step 6, 7, and the first two parts of 8 is exactly the same as opening linear combinations of commitments, which by the proof of security of the underlying commitment scheme means that whatever is the simulator accepts as opening will be the same as in the ideal functionality, had we issued opening commands. Thus what is left to show is the third part of step 8 which verifies that the commitments in C and C^* commits to the same value. To show this proceed as follows:

Denote the values $\mathcal{A}$ is supposed to send as in the protocol and denote those he actual sent in the same way but concatenated with a $'$. After the first two checks in 8 we know that

$$\tilde{\mathbf{t}}_q^{0'} = \tilde{\mathbf{t}}_q^0,\ \tilde{\mathbf{t}}_q^{1'} = \tilde{\mathbf{t}}_q^1,\ \tilde{\mathbf{c}}_q^{0'} = \tilde{\mathbf{c}}_q^0,\ \tilde{\mathbf{t}}_q^{*0'} = \tilde{\mathbf{t}}_q^{*0},\ \tilde{\mathbf{t}}_q^{*1'} = \tilde{\mathbf{t}}_q^{*1},\ \tilde{\mathbf{c}}_q^{*0'} = \tilde{\mathbf{c}}_q^{*0}$$

Since the code has minimum distance s, if the above was not true, at least s positions must have been changed. However, if that was the case then $\mathcal{A}$ would know at least s choicebits of P_j. It cannot do that with probability greater than 2^{-s} because these are only used in the ideal $\mathcal{F}_{\mathsf{OT}}$. From the last check in 8 we have that

$$\tilde{\mathbf{t}}_q^0 + \tilde{\mathbf{t}}_q^1 - \pi_m(\tilde{\mathbf{t}}_q^{*0} + \tilde{\mathbf{t}}_q^{*1}) = \mathbf{t}^{*'}_{R[q]} + \sum_{k\in[v]} \mathbf{r}_q[k] \cdot \bar{\mathbf{t}}^{*'}_{X[k]}$$

Remember that if $\mathcal{A}$ acts honestly it is the case that $\bar{\mathbf{t}}_k^* = \mathbf{t}_k - \pi_m(\mathbf{t}_k^*)$. This means that for the adversary to succeed it must come up with values $\bar{\mathbf{t}}_k^{*'}) \neq \mathbf{t}_k - \pi_m(\mathbf{t}_k^*)$ for at least one $k \in X$ s.t. the last check in 8 still holds, *before* he learns the values $\mathbf{r}_q$ for $q \in [s]$. We can describe this such that $\bar{\mathbf{t}}_k^{*'} = \epsilon_k + \mathbf{t}_k - \pi_m(\mathbf{t}_k^*)$. Again, since we have by assumption that the check in step 8 pass we have that the following must be true:

$$\mathbf{t}_{R[q]}+\left(\sum_{k\in[v]}\mathbf{r}_q[k]\cdot\mathbf{t}_{X[k]}\right)-\pi_m\left(\mathbf{t}^*_{R'[q]}+\left(\sum_{k\in[v]}\mathbf{r}_q[k]\cdot\mathbf{t}^*_{X[k]}\right)\right)=\bar{\mathbf{t}}^{*'}_{R[q]}+\sum_{k\in[v]}\mathbf{r}_q[k]\cdot\bar{\mathbf{t}}^{*'}_{X[k]}$$

$$\mathbf{t}_{R[q]}+\left(\sum_{k\in[v]}\mathbf{r}_q[k]\cdot\mathbf{t}_{X[k]}\right)-\pi_m\left(\mathbf{t}^*_{R'[q]}+\left(\sum_{k\in[v]}\mathbf{r}_q[k]\cdot\mathbf{t}^*_{X[k]}\right)\right)$$

$$=\pi_m\left(\epsilon_{R'[q]}+\mathbf{t}_{R[q]}-\pi_m(\mathbf{t}^*_{R'[k]})+\sum_{k\in[v]}\mathbf{r}_q[k]\cdot(\epsilon_k+\mathbf{t}_{X[k]}-\pi_m(\mathbf{t}^*_{X'[k]}))\right)$$

$$\mathbf{0}=\epsilon_{R'[q]}+\sum_{k\in[v]}\mathbf{r}_q[k]\cdot\epsilon_{X'[k]}$$

So there must be at least one $\epsilon_{X'[k]}\neq\mathbf{0}$, otherwise P_i is acting honestly (or no incorrect pair will be constructed) and there is nothing to show.[10] It is easy to see the best strategy for the adversary is to pick one value $\epsilon_{X'[k]}$ and then values $\epsilon_{R'[q]}$ for $q\in[s]$ s.t. $\epsilon_{R'[q]}+\mathbf{r}_q[k]\cdot\epsilon_{X'[k]}=\mathbf{0}$. Since $\mathbf{r}_q[k]$ is unknown to P_i when he makes his choice, and it is uniformly random and $\epsilon_{X'[k]}\neq 0$, we see that each value in $\mathbb{F}$ is equally likely to be hit. Thus he has $|\mathbb{F}|^{-1}$ probability of guessing $\epsilon_{R'[q]}$ for each $q\in[s]$. So his advantage is clearly at most 2^{-s}.

We notice that for **Public Multiplication**, we are basically just performing an **Open** and and **Input** of the commitments based on C^* and thus security follows from the base proof of [18].

B Proof of Theorem 3.1

We prove security in the presence of an adversary $\mathcal{A}$ who corrupts $A\subset\{P_1,\dots,P_p\}$. We denote the honest parties by $\bar{A}=\{P_1,\dots,P_p\}\setminus A$. The simulator $\mathcal{S}$ participates in the ideal execution, corrupts the same set of parties A and simulates the messages from the honest parties when the adversary is in the ideal world. The simulator $\mathcal{S}$ does as follows:

Init. For every $i\in A$ return the message (`init`) and pass on the call to $\mathcal{F}_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$.

Commit. The following simulation steps (and step numbers) are equivalent to the steps in the protocol.

1. Let I' be the agreed set of $\gamma+s$ new identifiers.
2. To simulate step 2 the simulator $\mathcal{S}$ (who acts as in functionality $\mathcal{F}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$) chooses $p(p-1)$ sets of $|I'|$ random messages from $\mathbb{F}^m$. That is, $\mathcal{S}$ uniformly picks $\mathbf{x}_k^{i,j}$ for every $k\in I'$, every $i\in[p]$ and every $j\in[p]\setminus\{i\}$. For every $i\in A$ in the instance $\mathcal{F}^{i,j}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$, $\mathcal{S}$ returns the messages $(\mathtt{commit},I')$ and $\left(\mathtt{committed},\left\{\left(k,\mathbf{x}_k^{i,j}\right)\right\}_{k\in I'}\right)$ for every $j\neq i$ to the adversary. In addition, for every $i\in A$ in the instance $\mathcal{F}^{j,i}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$, $\mathcal{S}$ returns the message $(\mathtt{committed},I')$ for every $j\neq i$ to the adversary.
3. At this point every party P_i chooses a message $\mathbf{x}_k^i$ for every $k\in I'$ to be committed to toward all other parties. However, we need to consider an adversary who chooses different values to input toward different parties. That is, we denote by $\mathbf{x}_k^{i,j\prime}$ the value that party P_i chooses to input (in the next step) toward party j.
4. To complete the simulation up to Step 4, for every $k\in I'$, every $j\in\bar{A}$ and every $i\in A$ send the message $(\mathtt{Input},k)$ to P_i. That is, return these messages to the adversary. In addition, as the corrupted parties sends the message $\left(\mathtt{Input},k,\mathbf{x}_k^{i,j\prime}\right)$ to the instances $\mathcal{F}^{i,j}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$ for every $k\in I'$ and every $j\in\bar{A}$, the simulator (who acts as the trusted party) extracts those messages $\mathbf{x}_k^{i,j\prime}$ (which might be non-equal for every $j\in\bar{A}$).
5. Let I and S be the agreed partitioning of I' as in Step 5 of the protocol.
6. To simulate Step 6 the simulator $\mathcal{S}$ chooses a random matrix $\mathbf{R}\in\mathbb{F}^{s\times\gamma}$, sends the message $(\mathtt{random},\mathbf{R})$ (as the output of $\mathcal{F}_{\mathsf{CT}}$) to the adversary.
7. For every $q\in S$, every $j\in\bar{A}$ and every $i\in A$ the simulator returns the message $(\mathtt{linear},(\{(k,\mathbf{R}_{q,k})\}_{k\in I}\cup\{(q,1)\},\beta,k')$ (for a freshly new identifier k') to the adversary, by emulating the **Linear Combination** instruction in $\mathcal{F}^{j,i}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$.

[10] It is not sufficient to pick one $\epsilon_{R'[q]}$ since these will not be used in an online pair and thus will have not effect on the openings.

The results of the random linear combinations are then opened to the adversary: For every $q \in S$, every $i \in A$ and every $j \in \bar{A}$ choose a uniformly random value $\mathbf{s}_q^j$ and returns to the adversary the message $\left(\texttt{opened}, \mathbf{s}_q^j\right)$. It remains to check consistency on the adversary's inputs: For every $q \in S$, every $j \in \bar{A}$ and every $i \in A$ compute $\mathbf{s}_q^{i,j} = \mathbf{x}_q^{i,j\prime} + \sum_{k \in I} \mathbf{R}_{q,k} \cdot \mathbf{x}_k^{i,j\prime}$.

8. For every $q \in S$ and every $j \in \bar{A}$ compute $\mathbf{c}_q^j = \sum_{i \in A} \mathbf{s}_q^{i,j}$. In addition, receive the input of the corrupted parties to functionality $\mathcal{F}_{\mathsf{EQ}}$, that is, for every $q \in S$ and every $i \in A$ receive $\{\mathbf{c}_q^j\}_{j \in [p]}$. If all $\mathbf{c}_q^j$ are equal for all $j \in [p]$ then output the message $(\texttt{equal}, \texttt{accept})$ as the output of $\mathcal{F}_{\mathsf{EQ}}$. Otherwise output the message $(\texttt{equal}, \mathbf{c}_q^{1\prime}, \ldots, \mathbf{c}_p^{1\prime}, \texttt{reject})$ where $\mathbf{c}_q^{j\prime} = \mathbf{c}_q^j$ for $j \in A$ and $\mathbf{c}_q^{j\prime}$ is uniformly random sampled from $\mathbb{F}^m$ for $j \in \bar{A}$. If the reject message was given as output then make $\mathcal{F}_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$ abort. Otherwise pass on the message $(\texttt{commit}, I)$ to $\mathcal{F}_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$.

Input. If an honest party gives input, then the simulator simply pass on the message $(\texttt{Input}, i, k)$ to $\mathcal{F}_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$ on behalf of the corrupted parties. It then returns the messages it received from the ideal functionality back to the adversary. If a corrupted party gives input, $\mathcal{S}$ receives $(\texttt{Input}, i, k, \mathbf{y})$ from $\mathcal{A}$ and picks $\mathbf{x}_k^{j\prime}$ uniformly at random and sends $\left(\texttt{opened}, \mathbf{x}_k^{j\prime}\right)$ to P_i on behalf of each honest party j. It then receives ϵ_k from the corrupt party and sets $\mathbf{y}' = \epsilon_k + \left(\sum_{i \in A} \mathbf{x}_k^i\right) + \left(\sum_{j \in \bar{A}} \mathbf{x}_k^{j\prime}\right)$ and inputs $(\texttt{Input}, i, k, \mathbf{y}')$ to $\mathcal{F}_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$.

Rand Extract the messages from $P_i \in A$ and pass on the call from P_i to $\mathcal{F}_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$. Furthermore define $\mathbf{x}_k^{A\prime} = \sum_{P_i \in A} \mathbf{x}_k^{i,j\prime}$.

Linear Combination Extract the messages from $P_i \in A$ and pass on the call from P_i to $\mathcal{F}_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$.

Open When opening commitment k $\mathcal{S}$ inputs $(\texttt{open}, k)$ to the ideal functionality on behalf of each corrupted party and receives back $(\texttt{opened}, k, \mathbf{x}_k)$. Then $\mathcal{S}$ computes the honest parties' share of the kth commitment $\mathbf{x}_k^{\bar{A}\prime} = \mathbf{x}_k - \mathbf{x}_k^{A\prime}$, chooses $|\bar{A}|$ uniformly random elements that sum up to $\mathbf{x}_k^{\bar{A}\prime}$, i.e. the elements $\{\mathbf{x}_k^{j\prime}\}_{j \in \bar{A}}$ such that $\mathbf{x}_k^{\bar{A}\prime} = \sum_{j \in \bar{A}} \mathbf{x}_k^{j\prime}$. If any honest shares of commitment $\mathbf{x}_k$ have already been sent to a corrupt party previously (through the **Input, Open** or **Partial Open** commands) then use the same values. Finally $\mathcal{S}$ sends the messages $\{(\texttt{opened}, k, \mathbf{x}_k^{j\prime})\}_{j \in \bar{A}}$ on every instance $\mathcal{F}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}^{i,j}$ with $i \in A$ to the adversary. If $\mathcal{A}$ aborts or don't opens its shares towards the honest party then input abort to the ideal functionality so the honest parties don't receive the opened value.

Partial Open When partially opening commitment k towards a malicious party P_i, $\mathcal{S}$ inputs $(\texttt{open}, i, k)$ to the ideal functionality on behalf of each corrupted parties and receives back $(\texttt{opened}, i, k, \mathbf{x}_k)$. Proceed like the simulation of the **Open** command.

To argue indistinguishability between the real and ideal world we show the following:

1. The simulation aborts during **Commit** with the same probability as it aborts in the real execution, which is negligible in s.
2. All values sent to $\mathcal{A}$ in the simulation are indistinguishable from the values sent by the honest parties in the real execution.

In the following we go through the two items.

1. We see that the simulation aborts in Step 8 with exactly the same probability and cases as in the real execution. The protocol aborts in one of two cases:
   - If the corrupted parties input different values toward different honest parties notice that the simulation aborts with exactly the same probability as it aborts in the real execution since the simulator executes exactly the same check (on behalf of the honest parties) using random coins that were chosen from exactly the same distribution, thus, the simulation and real execution abort in this case in the same probability.
   - Even though the simulation aborts with the same probability as the real protocol we must still argue that this happens if the adversary is inconsistent in *any* input between two honest parties. If not then the multi-party commitment is not well-defined as it can be opened to different values towards the two different honest parties. To succeed the adversary must pass the linear combination check. However, since a random linear combination is a universal hash function and it is sampled *after* he commits towards the parties, then the probability of a collision in a single linear combination is at most $|\mathbb{F}|^{-1}$, since the linear combination is based on component-wise multiplication of a *single* element in $\mathbb{F}$. However, since we do s independent random linear combinations we get that the adversary succeeds in finding a collision with probability at most $|\mathbb{F}|^{-s}$.

– In regards to the equality test functionality $\mathcal{F}_{\mathsf{EQ}}$, we notice that the simulator sees the inputs of the corrupted parties to this functionality. Regarding the honest parties we see that since $\mathbf{s}_q^j$ of $j \in \bar{A}$ is uniformly random and completely unknown to the adversary (because $\mathbf{x}_q^j$ is a random one time pad constructed by $\mathcal{F}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$ and only used here) the values $\mathbf{c}_q^{j\prime}$ for $j \in \bar{A}$ are indistinguishable from uniformly random values which is exactly the same in the real protocol. This means that the simulation outputs `reject` in the same cases as in the real protocol along with inputs of the parties which are indistinguishable from the real protocol.

2. We go over the protocol instructions one-by-one:
   – **Commit.** The first step where non-trivial information is sent to $\mathcal{A}$ is in Step 7 of **Commit**. Specifically, the openings $\{\mathbf{s}_q^{j\prime}\}_{j\in\bar{A}}$ to $\mathcal{A}$. We notice that in the real protocol these values will be uniformly random for all honest parties because the value $\mathbf{x}_q^i$ is used to hide $\sum_{k\in I} \mathbf{R}_{q,k} \cdot \mathbf{x}_k^i$ since this is the only place $\mathbf{x}_q^i$ is used. Thus simply picking a random value as $\mathcal{S}$ does is indistinguishable from the real world.
   – **Input.** We notice that in the real execution a corrupt party giving an input with index k receives an opening to each honest party's share of commitment k. Observe that in both the real execution and the simulation the share is uniformly random. However, in the real execution it depends on the values sent in Step 7 of **Commit**, whereas in the simulation it is independent. Even though, as we have discussed, the values sent in Step 7 are one-time padded with another uniformly random value and thus the real and simulated worlds are indistinguishable. To ensure that the input of the corrupt party gets correctly used in the rest of the protocol the simulator computes the value $\mathbf{y}'$, which is the value that would be opened to in the real protocol and inputs this on behalf of the corrupted party to the ideal functionality. To see that this is in fact that value that would be opened in the real execution, notice that the corrupt party is free to pick ϵ_k in any way, but that once it is broadcast to the honest parties it defines exactly what the sum of the underlying $\mathcal{F}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$ commitments will open to.
   – **Rand, Linear Combination.** No information is sent in these steps, so the simulation is perfect.
   – **Open, Partial Open.** The simulator receive the message $(\mathtt{opened}, k, \mathbf{x}_k)$ from the ideal functionality. First see that by the computation of $\mathbf{x}_k^{\bar{A}\prime}$ we ensure that that the opened shares $\mathcal{A}$ receives, summed with the shares he committed to, will always be the same in the real and simulated world. To see that the opened values by each honest party are distributed similarly in the real and simulated world. Consider the case where there is only a single honest party. In this case its share is completely defined from the shares $\mathcal{A}$ is committed to along with the value opened to by the ideal functionality. Thus it is clearly distributed similarly in the real and simulated world. Next see that if there are more honest parties the simulator picks their shares randomly under the constraint that they sum to the well-defined value $\mathbf{x}_k^{\bar{A}\prime}$. This is also the way the shares are picked in the real world and thus they are indistinguishable. In particular we notice that since the simulator uses any randomly picked shares $\mathbf{x}_k^j$ for a random party $j \in \bar{A}$ it has already sent to the adversary, there will be no inconsistency. Finally, see that the values will always be well defined since consistency between the opened values will be ensured by $\mathcal{F}_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$ and that since $\mathcal{S}$ has extracted the shares of the corrupted parties (which cannot be changed because of the consistency check except with probability at most $|\mathbb{F}|^{-s} \leq 2^{-s}$ as explained previously) and the honest parties shares are defined from these, once and for all.

■

C Proof of Lemma 4.2

First consider the correctness of CorrectnessTest in Fig. 11, i.e. that if both triples are correct then the procedure will *never* abort. Let $t_1 = ([\![\mathbf{a}]\!], [\![\mathbf{b}]\!], [\![\mathbf{c}]\!])$ and $t_2 = ([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{z}]\!])$ be two correct triples, then the following holds:

$$\begin{aligned} [\![\mathbf{e}]\!] &= r \cdot [\![\mathbf{z}]\!] - [\![\mathbf{c}]\!] - \boldsymbol{\epsilon} * [\![\mathbf{b}]\!] - \boldsymbol{\rho} * [\![\mathbf{a}]\!] - \boldsymbol{\rho} * \boldsymbol{\epsilon} \\ &= r \cdot [\![\mathbf{z}]\!] - [\![\mathbf{c}]\!] - (r \cdot [\![\mathbf{x}]\!] - [\![\mathbf{a}]\!]) * [\![\mathbf{b}]\!] - ([\![\mathbf{y}]\!] - [\![\mathbf{b}]\!]) * [\![\mathbf{a}]\!] \\ &\quad -([\![\mathbf{y}]\!] - [\![\mathbf{b}]\!]) * (r \cdot [\![\mathbf{x}]\!] - [\![\mathbf{a}]\!]) \\ &= r \cdot [\![\mathbf{z}]\!] - [\![\mathbf{c}]\!] - r \cdot [\![\mathbf{x}]\!] * [\![\mathbf{b}]\!] + [\![\mathbf{a}]\!] * [\![\mathbf{b}]\!] - [\![\mathbf{y}]\!] * [\![\mathbf{a}]\!] + [\![\mathbf{b}]\!] * [\![\mathbf{a}]\!] \\ &\quad -r \cdot [\![\mathbf{y}]\!] * [\![\mathbf{x}]\!] + r \cdot [\![\mathbf{b}]\!] * [\![\mathbf{x}]\!] + [\![\mathbf{y}]\!] * [\![\mathbf{a}]\!] - [\![\mathbf{b}]\!] * [\![\mathbf{a}]\!] \\ &= r \cdot [\![\mathbf{z}]\!] - [\![\mathbf{c}]\!] + [\![\mathbf{a}]\!] * [\![\mathbf{b}]\!] - r \cdot [\![\mathbf{y}]\!] * [\![\mathbf{x}]\!] \end{aligned} \tag{1}$$

which is opened to $\mathbf{0}$ since $\mathbf{z} = \mathbf{x} * \mathbf{y}$ and $\mathbf{c} = \mathbf{a} * \mathbf{b}$.

Next see that nothing is leaked on the elements of triple t_1. This follows because the r picked is never 0. Thus the values opened, $\boldsymbol{\epsilon}$ and $\boldsymbol{\rho}$, will not leak anything on $\mathbf{a}$, respectively $\mathbf{b}$, as these values will be one-time padded by $\mathbf{x}$, respective $\mathbf{y}$. Furthermore, if $\mathbf{e} \neq \mathbf{0}$, then the protocol will abort. This is in the preprocessing phase, thus before any private data is in play, and thus any leakage is acceptable. If instead $\mathbf{e} = \mathbf{0}$, then clearly nothing is leaked as $\mathbf{0}$ is constant.

Let us examine the possible outcomes of procedure CorrectnessTest when the assumption that they are both correct does not hold. That is, if t_2 is malformed then we have $\mathbf{z} = \mathbf{x} * \mathbf{y} + \boldsymbol{\Delta}_2$ for some $\boldsymbol{\Delta}_2 \in \mathbb{F}^m$, thus the result of Eq. 1 is $\mathbf{e} = r \cdot \boldsymbol{\Delta}_2$. If t_1 is malformed then we have $\mathbf{c} = \mathbf{a} * \mathbf{b} + \boldsymbol{\Delta}_1$ for some $\boldsymbol{\Delta}_1 \in \mathbb{F}^m$ and the result of Eq. 1 is $\mathbf{e} = -\boldsymbol{\Delta}_1$. Finally if both are incorrect than we have $\mathbf{e} = r \cdot \boldsymbol{\Delta}_2 - \boldsymbol{\Delta}_1$. Thus, after applying procedure CorrectnessTest to two triples we end up in one of the following cases:

1. **Both triples are correct.** From the correctness shown above the result of the procedure is a correct triple.
2. **Exactly one triple is malformed.** Note that either $\boldsymbol{\Delta}_1 = \mathbf{0}$ or $\boldsymbol{\Delta}_2 = \mathbf{0}$ (but not both). If $\boldsymbol{\Delta}_2 = \mathbf{0}$ then the result is $\mathbf{e} = -\boldsymbol{\Delta}_1 \neq \mathbf{0}$ and the parties abort. If $\boldsymbol{\Delta}_1 = \mathbf{0}$ then $\mathbf{e} = r\boldsymbol{\Delta}_2 \neq \mathbf{0}$ (since $r \neq 0$) and the parties also abort. Thus, either we will abort or we accept a correct triple t_1.
3. **Both are malformed.** In this case we have $\boldsymbol{\Delta}_1, \boldsymbol{\Delta}_2 \neq \mathbf{0}$. Notice that we have $\mathbf{e} = r \cdot \boldsymbol{\Delta}_2 - \boldsymbol{\Delta}_1 = \mathbf{0}$ if and only if $r \cdot \boldsymbol{\Delta}_2 = \boldsymbol{\Delta}_1$ which means that $r = \boldsymbol{\Delta}_1 * (\boldsymbol{\Delta}_2)^{-1}$ (i.e. $\boldsymbol{\Delta}_1$ multiplied with the multiplicative inverse of $\boldsymbol{\Delta}_2$). Since r is chosen uniformly at random from $\mathbb{F}\backslash\{0\}$ we have that the parties will *not* abort with probability of at most $\frac{1}{|\mathbb{F}|-1}$.

From the above analysis it follows that an incorrect triple from the $\tau_1 \cdot (\tau_2)^2 T$ triples will end up being considered as one of the $(\tau_2)^2 \cdot T$ correct triples if and only if it was assigned to a bucket with $\tau_1 - 1$ triples and pass the CorrectnessTests applied to it. Notice that an incorrect triple can only pass an instance of CorrectnessTest if it gets paired with another incorrect triple *and* $r = \boldsymbol{\Delta}_1 * (\boldsymbol{\Delta}_2)^{-1}$. Thus we wish to bound the probability that there exists a bucket consisting entirely of incorrect triples *and* all the $\tau_1 - 1$ checks done in this bucket pass. We have from Corollary 4.1 that the first event only happens with probability at most $N\binom{N\tau_1+\tau_1}{\tau_1}^{-1}$ and the probability of the second event is at most $(|\mathbb{F}|-1)^{-1}$ for each CorrectnessTest. Since CorrectnessTest will be carried out $\tau_1 - 1$ independent times (using a new triple each time), we get the probability of the second event is at most $(|\mathbb{F}|-1)^{-\tau_1+1}$. Thus the probability that a specific incorrect triple gets accepted is at most $N\binom{N\tau_1+\tau_1}{\tau_1}^{-1} \cdot (|\mathbb{F}|-1)^{-\tau_1+1}$.

Furthermore, let $0 < t < (\tau_2)^2 \cdot T$ be the amount of buckets the adversary choose to corrupt. Then we have from [20] that the probability of t bad buckets remaining after *Cut-and-Choose* is at most $\binom{(\tau_2)^2 \cdot T}{t}\binom{(\tau_2)^2 \cdot T\tau_1+\tau_1}{t\tau_1}^{-1}$. Thus for the adversary to succeed in the sacrificing without abort, it must be the case that the checks in all t buckets pass. Thus this happens with probability $(|\mathbb{F}|-1)^{-\tau_1+1^t}$. Thus the overall success probability of the adversary is at most:

$$\binom{(\tau_2)^2 \cdot T}{t}\binom{\tau_1 \cdot (\tau_2)^2 \cdot T + \tau_1}{t\tau_1}^{-1} \cdot (|\mathbb{F}|-1)^{-t\tau_1+t} .$$

It was already shown in [20] that the first term is maximized for $t = 1$. Now see that this is also true for the second term $((|\mathbb{F}|-1)^{-t\tau_1+t})$ as $\tau_1 \geq 2$ and so $-t\tau_1 + t$ is maximized for as small t as possible, which in our case is $t = 1$. Thus we wish to have

$$\begin{aligned} 2^{-s} &\geq (\tau_2)^2 \cdot T\binom{\tau_1 \cdot (\tau_2)^2 \cdot T + \tau_1}{\tau_1}^{-1} \cdot (|\mathbb{F}|-1)^{-\tau_1+1} \\ &= (\tau_2)^2 \cdot T\left(\frac{(\tau_1 \cdot (\tau_2)^2 \cdot T + \tau_1)!}{(\tau_1 \cdot (\tau_2)^2 \cdot T)!\tau_1!}\right)^{-1} \cdot (|\mathbb{F}|-1)^{-\tau_1+1} \\ &= \frac{(|\mathbb{F}|-1)^{-\tau_1+1} \cdot (\tau_2)^2 \cdot T \cdot (\tau_1 \cdot (\tau_2)^2 \cdot T)! \cdot \tau_1!}{(\tau_1 \cdot (\tau_2)^2 \cdot T + \tau_1)!} \end{aligned}$$

and the lemma follows directly.

■

D Proof of Lemma 4.3

Before proving Lemma 4.3, we present the following helper lemma:

Lemma D.1. *Given a bucket of τ_2 triples where at least one is non-leaky on x (resp. on y) then the combining produces a triple that is non-leaky on x (resp. on y).*

Proof. We first argue correctness of the combining approach by showing that for the triple $([\![\mathbf{x}']\!], [\![\mathbf{y}']\!], [\![\mathbf{z}']\!])$, resulting from the execution of step 1-4 in **Mult** in Fig. 13 it holds that $\mathbf{z}' = \mathbf{x}' * \mathbf{y}'$ (given that $\mathbf{z}_k = \mathbf{x}_k * \mathbf{y}_k$ for $k \in [\tau_2]$) by the following:

$$\begin{aligned}
\mathbf{z}' &= \mathbf{x}_1 * \mathbf{y}_1 + \textstyle\sum_{k=2}^{\tau_2} \epsilon_k * \mathbf{x}_k + \mathbf{z}_k \\
&= \mathbf{x}_1 * \mathbf{y}_1 + \textstyle\sum_{k=2}^{\tau_2} (\mathbf{y}_1 - \mathbf{y}_k) * \mathbf{x}_k + \mathbf{z}_k \\
&= \mathbf{x}_1 * \mathbf{y}_1 + \textstyle\sum_{k=2}^{\tau_2} \mathbf{y}_1 * \mathbf{x}_k - \mathbf{y}_k * \mathbf{x}_k + \mathbf{z}_k \\
&= \mathbf{x}_1 * \mathbf{y}_1 + \textstyle\sum_{k=2}^{\tau_2} \mathbf{y}_1 * \mathbf{x}_k - \mathbf{y}_k * \mathbf{x}_k + \mathbf{x}_k * \mathbf{y}_k \\
&= \mathbf{x}_1 * \mathbf{y}_1 + \textstyle\sum_{k=2}^{\tau_2} \mathbf{y}_1 * \mathbf{x}_k = \textstyle\sum_{k=1}^{\tau_2} \mathbf{y}_1 * \mathbf{x}_k = \mathbf{x}' * \mathbf{y}_1
\end{aligned}$$

We now show that combining τ_2 triples where at least *one* of them is non-leaky is enough for generating a new non-leaky triple. Thus, by combining $s + 1$ triples it is guaranteed that the result triple is non-leaky. However, this would incur a multiplicative overhead of $O(s)$ on the number of multiplication triples that the parties are required to generate. Instead, in the batch model, it is possible to generate sufficiently many multiplication triples, then divide them into buckets, where each buckets contains τ_2 triples, where τ_2 is significantly less than s. If we combine the τ_2 triples contained in some bucket, we get that the new triple is non-leaky if at least one of the τ_2 is non-leaky as well. For a given statistical security parameter s and number of triples to be contained in each bucket τ_2, we want to know the amount of triples, denoted T', the parties need to generate in order to have all combined triples (i.e. from all buckets) to be non-leaky with overwhelming probability (in s). ∎

We now proceed with the proof of Lemma 4.3: By generating T' triples and uniformly dividing them into buckets of size τ_2 we note that the probability of having some bucket full of leaky triples equals the number of buckets times the probability of choosing τ_2 leaky triples out of the T' generated such that s of them are leaky. That is, let $\mathsf{bad\text{-}bucket}$ be the event of choosing τ_2 leaky triples out of T' triples, where s of them are leaky. Then the probability that at least one of the combined triples is leaky is $Pr[\mathsf{bad\text{-}bucket}] \cdot \frac{T'}{\tau_2}$ (by the union bound), and we want this to be less than 2^{-s}. The probability of choosing τ_2 leaky triples out of the T' is $\frac{\binom{s}{\tau_2}}{\binom{T'}{\tau_2}}$ by a counting argument as there are $\binom{s}{\tau_2}$ possible ways of choosing a combined triple and there are $\binom{s}{\tau_2}$ possible ways of choosing this consisting entirely of leaky triples. Using the bounds on the binomial coefficient,i.e. $\left(\frac{n}{k}\right)^k \leq \binom{n}{k} \leq \left(\frac{n \cdot e}{k}\right)^k$ where e is the base of the natural logarithm, we get:

$$\frac{\binom{s}{\tau_2}}{\binom{T'}{\tau_2}} \leq \frac{\left(\frac{s \cdot e}{\tau_2}\right)^{\tau_2}}{\left(\frac{T'}{\tau_2}\right)^{\tau_2}} = \left(\frac{s \cdot e}{T'}\right)^{\tau_2} \quad \text{and we want} \quad \frac{T'}{\tau_2} \cdot \left(\frac{s \cdot e}{T'}\right)^{\tau_2} = \frac{(s \cdot e)^{\tau_2}}{\tau_2 \cdot T'^{\tau_2 - 1}} < \frac{1}{2^s}$$

thus, it follows that the number of triples required is

$$T' > \sqrt[\tau_2 - 1]{\frac{(s \cdot e)^{\tau_2} \cdot 2^s}{\tau_2}}$$

∎

E Proof of Theorem 4.4

Let $\mathcal{A}$ be the adversary and $A \subset \{P_1, \ldots, P_p\}$ the corrupted parties. Also denote by $\bar{A} = \{P_1, \ldots, P_p\} \backslash A$ the honest parties. In the following we describe the simulator $\mathcal{S}$ who interacts in the ideal execution of the protocol and produces a view statistically close to the adversary's view in the real execution. We assume that before issuing the command **Mult** the parties have raw multiparty commitments of $3(\tau_1 + \tau_1 \cdot (\tau_2)^2 \cdot T)$ uniformly random values in $\mathbb{F}^m$, those values are defined (from Appendix B) and for every $[\![\mathbf{x}]\!]$ the simulator $\mathcal{S}$ already extracted the adversaries' shares denoted by $\mathbf{x}_k^{A\prime} = \sum_{P_i \in A} \mathbf{x}_k^{i,j\prime}$ for any $j \in \bar{A}$. The simulation goes as follows:

1. Upon receiving a message $(\mathtt{mult}, C)$ from all parties where $c \in C$ is a an index of a raw commitment, invoke $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ with the command $(\mathtt{mult}, C)$.
2. **Construction:**
   (a) Choose random values as the honest parties' shares of $\mathbf{x}$, that is, for every raw commitment $[\![\mathbf{x}]\!]$ choose $\mathbf{x}^i$ for every $i \in \bar{A}$.
   (b) For every 3 raw multiparty commitments $[\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{z}]\!]$ with indexes $x, y, z \in C$, for every execution of $\mathsf{ArithmeticOT}(x^i, y^j))$ between P_i (who inputs x^i) and P_j (who inputs y^j) simulate the procedure as follows:
      i. If party P_i is corrupted then $\mathcal{S}$ extracts its input x_q, picks a random value $r_q \in \mathbb{F}$ and returns this to $\mathcal{A}$ on behalf of P_j as the output of $\mathcal{F}_{\mathsf{OT}}$. $\mathcal{S}$ then defines $z^i = \sum_{q \in [\ell]} r_q \cdot 2^{q-1} \in \mathbb{F}$.
      ii. If party P_j is corrupted then $\mathcal{S}$ extracts s_q^0, s_q^1 from P_j's input to $\mathcal{F}_{\mathsf{OT}}$. It defines $z^j = -\sum_{q \in [\ell]} s_1^0 \cdot 2^{q-1} \in \mathbb{F}$.
   (c) Receive the values $\mathbf{t}^i$ from each $i \in A$. The simulator check if it holds that

$$\sum_{i \in A} \mathbf{t}^i = \sum_{i \in A} \left(\mathbf{x}^i * \mathbf{y}^i + \sum_{j \neq i} \mathbf{s}^i_{i \leftarrow j} + \sum_{j \neq i} \mathbf{s}^i_{j \leftarrow i} \right) - \mathbf{z}^i ,$$

   where $\mathbf{z}^i$ corresponds to P_i's share of commitment $\mathbf{z}$ defined in the **Commit** phase by $\mathcal{S}$ (the value which in the simulation of **Commit** is denoted by $\mathbf{x}^{i,j\prime}$) and $\mathbf{s}^i_{i \leftarrow j}$ is the simulated output of $\mathsf{ArithmeticOT}$ when P_i is the receiver and $\mathbf{s}^i_{j \leftarrow i}$ when P_i is the sender. If it does not hold then $\mathcal{S}$ marks the triple $([\![\mathbf{x}]\!], [\![\mathbf{y}]\!], [\![\mathbf{z}]\!])$ as bad and stores the difference
   $\delta = \sum_{i \in A} \mathbf{t}^i - \left(\mathbf{x}^i * \mathbf{y}^i + \sum_{j \neq i} \mathbf{s}^i_{i \leftarrow j} + \sum_{j \neq i} \mathbf{s}^i_{j \leftarrow i} \right) + \mathbf{z}^i$.
3. **Cut-and-Choose:** $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{CT}}$ to sample a random grouping of τ_1 triples. For each of these it proceeds as follows:
   (a) Simulate the opening of commitments $[\![\mathbf{x}]\!], [\![\mathbf{y}]\!]$ and $[\![\mathbf{z}]\!]$ by picking the honest parties' shares uniformly at random and using these to emulate the opening of the underlying $\mathcal{F}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$ functionality under the constraint that $\mathbf{z} = \mathbf{x} * \mathbf{y}$. However, if $\mathbf{z}$ is marked as bad, then the simulator instead picks the honest parties shares under the constraint that $\mathbf{z} = \mathbf{x} * \mathbf{y} + \delta$. Furthermore, abort and make $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ abort as well if any of the triples opened are marked as bad.
4. **Sacrificing:**
   $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{CT}}$ to sample a random grouping of the constructed multiplication triples into buckets of τ_1 triples each. $\mathcal{S}$ simulates $\tau_1 - 1$ executions of $\mathsf{CorrectnessTest}$ (Fig. 11) using the extracted values of the corrupt parties as follows:
   (a) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{CT}}$ to select a random $r \in \mathbb{F} \backslash \{0\}$.
   (b) $\mathcal{S}$ picks the honest parties' shares of $[\![\boldsymbol{\epsilon}]\!]$ and $[\![\boldsymbol{\rho}]\!]$ uniformly at random and emulate the opening of these values based on the underlying $\mathcal{F}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$ functionality.
   (c) Based on $\boldsymbol{\epsilon}$ and $\boldsymbol{\rho}$ it uses $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ to compute and open $\mathbf{e}$. Note that this will be based on the original random commitments to $\mathbf{z}$ and $\mathbf{c}$ and not the adjusted values from the **Construction** phase. If $[\![\mathbf{z}]\!]$ is marked as bad then let δ_z be the difference associated with $\mathbf{z}$, otherwise let $\delta_z = \mathbf{0}$. Similarly if $[\![\mathbf{c}]\!]$ is marked as bad then let δ_c be the difference associated with $\mathbf{c}$, otherwise let $\delta_c = \mathbf{0}$. $\mathcal{S}$ then checks if $r \cdot \delta_z - \delta_c = \mathbf{0}$. If this is so it emulates the opening of $[\![\mathbf{e}]\!]$ to $\mathbf{0}$, otherwise it emulates the opening to the value $r \cdot \delta_z - \delta_c$.
5. **Combining:** Continue the simulation with the values that the corrupted parties are committed to:
   (a) Combine over $\mathbf{x}$: $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{CT}}$ to sample a random grouping of the constructed multiplication triples into buckets of τ_2 triples each.
      – Emulate the opening to ϵ_k for $k \in \{2, \ldots, \tau_2\}$ by picking the honest parties' shares uniformly at random.
   (b) Combine over $\mathbf{y}$ is done similarly to the combining over $\mathbf{x}$.

The above simulation produces a view with the same distribution as the view of the environment in the real execution. To see this, first notice that in the real execution the honest parties' shares of the multiparty commitments are uniformly random sampled and for everything opened in the simulation above this is also the case (under the constraint that things add up correctly). Next see that in the simulation above, whenever something is opened, there always remain at least one random additive share of at least one honest party, which means that everything done in the simulation can be explained during **Open**, no matter what the true shares of the honest parties were in the ideal functionality. In addition, note that the opened triples in cut-and-choose along with the sacrificed triples and triples that were used in the combining step are never used again in the protocol after those steps and thus could not be used by the environment in an attempt to distinguish between the views.

More specifically, first see that in **Construction** if the sending party in ArithmeticOT is corrupt then it learns nothing, but the simulator can extracts its input to $\mathcal{F}_{\mathsf{OT}}$ and thus compute which value $\mathbf{t}^i$ it should broadcast in step (b). On the other hand, if the receiving party is corrupt, see that what the corrupt party receives is a uniformly random value in $\mathbb{F}$ no matter if it is executing in the real world or with the simulator. More specifically if it requests message 0 then it gets $s_q^0 = r_q \in_R \mathbb{F}$. If instead it requests message 1 it get $s_q^1 = r_q \in_R \mathbb{F}$ in the simulation and $s_q^1 = y + r_q$ in the real execution, which is also uniformly random since r_q is uniformly distributed and thus acts as a one-time pad. In particular this holds since the only other place where r_q is only used, is to compute $\mathbf{t}^i$, but the malicious party should accordingly compute $\mathbf{t}^j$ such that it gets canceled out. If it does not do that then the protocol will abort according to Lemma 4.2. However, the adversary could try to learn something of the honest parties' input by a selective attack, and thus be able to distinguish between the real execution and the simulation. However, Lemma 4.3 shows that such an attempt is futile, since triple that is not leaky on $[\![\mathbf{x}]\!]$ will act as a one-time pad and thus remove the leakage. Similarly for $[\![\mathbf{y}]\!]$.

For the opening of $\mathbf{z}$ in **Cut-and-Choose** see that if a corrupt party did any sort of cheating in **Construction** s.t. $[\![\mathbf{z}]\!] \neq [\![\mathbf{x} * \mathbf{y}]\!]$ then the simulator will know exactly how big the difference is, since it knows what each corrupt party should send if they followed the protocol. In particular notice that this is the case, even when the simulator does not know the honest parties shares of $[\![\mathbf{z}]\!]$ as the error will be additive as can be seen from step **Construction** (c). Thus picking any random share for each honest party obeying this constraint will yield the same distribution for an incorrectly constructed triple.

The same argument goes for **Sacrificing**. In particular notice that when one or two incorrect triples are paired in a bucket the simulator will ensure that it picks the honest parties shares s.t. the difference between the true value $\mathbf{e}$ from the ideal functionality and the simulated output will be the same.

For the combining we simply simulate the honest parties' shares using random values, since ϵ will always be one-time padded with a random commitment only used once. Furthermore, from Lemma 4.3 we see that even a selective attack on an honest party's input will not yield any further information.

F Proof of Theorem 4.5

We see that the methods **Init, Commit, Rand, Linear Combination, Open** and **Partial Open** are implemented like in $\Pi_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$ and that the ideal functionality of these methods, from $\mathcal{F}_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$, are the same. Thus we piggyback on the proof of security of $\mathcal{F}_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$ of 3.1. Specifically this means that after **Commit** has been executed without abort the simulator has uniquely defined values of each of $P_i \in A$ shares of commitments (with overwhelming probability), denoted by $\mathbf{x}_k^{A\prime} = \sum_{P_i \in A} \mathbf{x}_k^{i,j\prime}$ for any $j \in \bar{A}$ and commitment k where the adversary $\mathcal{A}$ corrupts $A \subset \{P_1, \ldots, P_p\}$ and $\bar{A} = \{P_1, \ldots, P_p\} \backslash A$. We start by defining a simulator $\mathcal{S}$ simulating the honest parties $\bar{A} = \{P_1, \ldots, P_p\} \setminus A$. As before, the simulator knows the values that the adversary is committed to and as proved above, the same values are committed toward all honest parties with overwhelming probability. That is, the simulator knows $\mathbf{x}_x^i, \mathbf{x}_y^i, \mathbf{x}_a^i, \mathbf{x}_b^i$ for $i \in A, x \in X, y \in Y, a \in A, b \in B$ and proceeds as follows:

ReOrg.

1. Simulate the honest parties by picking values $\epsilon_{x,y}^j \in \mathbb{F}^m$ uniformly at random for $P_j \in \bar{A}$ and broadcast these to $\mathcal{A}$ like in the protocol. Receive $\epsilon_{x,y}^i$ from the adversary for every $i \in A$.
2. Simulate the honest parties by picking values $\epsilon_{a,b}^j \in \mathbb{F}^m$ uniformly at random for $P_j \in \bar{A}$ and broadcast these to $\mathcal{A}$ like in the protocol.

3. Do nothing.
4. Sample $\mathbf{R} \in \mathbb{F}^{\nu \times s}$ uniformly at random. Pick $\mathbf{s}_q^{\bar{A}} \in \mathbb{F}^m$ uniformly at random and let $\mathbf{s}_q{}' = \mathbf{s}_q^{\bar{A}}{}' + \sum_{k \in [\nu]} \mathbf{R}_{q,k} \cdot \sum_{i \in A} \mathbf{x}_{X_k}^i{}'$. Then pick $\mathbf{s}_q^j{}'$ for each $j \in \bar{A}$ uniformly random shares under the constraint that $\mathbf{s}_q^{\bar{A}} = \sum_{j \in \bar{A}} \mathbf{s}_q^j{}'$. Use these values to simulate an opening to $\mathbf{s}_q^A + \mathbf{s}_q^{\bar{A}}$ and $\phi(\mathbf{s}_q^A + \mathbf{s}_q^{\bar{A}})$ for each $q \in [s]$.
5. Perform the same random linear combination test on $\boldsymbol{\epsilon}_{x,y}^i$ for $i \in A$ exactly as done in Step 5 of the protocol (but only on the shares of the adversary). If the test fails then `abort`.
6. Input (`reOrg`, C) into the ideal functionality on behalf of the malicious parties.

Since there is no private output from **ReOrg** it is sufficient to prove that the values sent to $\mathcal{A}$ are indistinguishable in the real world and the simulation and that the simulation aborts with the same probability as the real protocol. First note that the simulation aborts with exactly the same probability as the real execution aborts since the randomness (used as coefficients) is taken from the same distribution in both cases and the same linear combination test is done. It follows that the adversary pass the linear combination test with a negligible probability in s (as we abuse terminology, that means less than 2^{-s}) because this basically reduces to guessing a collision of a randomly sampled universal hash function, as discussed in the proof of Theorem 3.1. Next, we show indistinguishability between the simulation and the real execution using a hybrid argument, on every incoming message to the adversary:

Let H_1 be as the real execution. Define hybrid H_2 where everything is the same as in H_1 (but using the simulator for $\Pi_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$ for **Init, Commit, Rand, Input, Linear Combination**) except that in step 4 the value $\mathbf{s}_q^j$ for $P_j \in \bar{A}$ is uniformly random sampled on-the-fly and setting $\bar{\mathbf{s}}_q = \phi(\mathbf{s}_q^j)$. Furthermore opening of these values is handled without calling the **Open** method, but by H_2. Specifically it ensures the adversary is giving correct openings in accordance with the adversarial shares extracted by the simulator in **Commit**. It also computes the openings of the honest parties to send to $\mathcal{A}$. It does so by randomly selecting $\mathbf{s}_q^j$ for each $j \in \bar{A}$ under the constraint that $\sum_{j \in \bar{A}} \mathbf{s}_q^j$. Note that we do not need to ensure that the values $\mathbf{s}_q$ and $\bar{\mathbf{s}}_q$ opened towards the honest parties in the real world and the hybrid are indistinguishable since these are not opened in the ideal functionality and are thus only internal parts of the **ReOrg** method. This means that the ideal functionality does not perform any **Open** commands as part of **ReOrg**. This is purely part of the real world implementation of **ReOrg** and simulated in the hybrid.

Now to see that H_1 is computationally indistinguishable from H_2 we see that in the real protocol $\mathbf{s}_q^i$ has one term, $\mathbf{x}_{r_q}^i$ which is also uniformly random sampled. Furthermore we see that $\mathbf{x}_{r_q}^i$ is never used again (since it is removed from the set of raw commitments). Thus $\mathbf{s}_q^i$ is actually a random and independently sampled valued. Furthermore $\mathcal{A}$ has at most negligible knowledge of it because of the security of the security of the **Commit** method as proved in Theorem 3.1. This means that the opened commitments $\mathcal{A}$ learns in step 4 are indistinguishable between H_1 and H_2. Furthermore we see that the methods **Init, Commit, Rand, Input, Linear Combination** are indistinguishable from H_1 and H_2 because of the proof of Theorem 3.1. Finally we see that the method **Open** and **Partial Open** are perfectly indistinguishable between H_1 and H_2 by definition.

Next define the hybrid H_3 to be the same as H_2 except that in step 1 the value $\boldsymbol{\epsilon}_{x,y}^i$ for $P_i \in \bar{A}$ is uniformly random sampled on-the-fly. Now to see that H_3 is computationally indistinguishable from H_2 we see that in H_2 the value $\boldsymbol{\epsilon}_{x,y}^i$ has one term, $\mathbf{x}_y^i$ which is also uniformly random sampled. First see that when executing **ReOrg** the adversary is oblivious to $\mathbf{x}_y^i$ for each $i \in \bar{A}$ because of the security of the **Commit** method as proved in Theorem 3.1. Next we see that after step 1, $\mathbf{x}_y^i$ is only used again to construct a new commitment:

$$\begin{aligned}\llbracket \mathbf{x}_{y'} \rrbracket = \llbracket \mathbf{x}_y \rrbracket + \textstyle\sum_{j \in [P]} \boldsymbol{\epsilon}_{x,y}^i &= \left\llbracket \left(\textstyle\sum_{i \in A} \mathbf{x}_y^i + \boldsymbol{\epsilon}_{x,y}^i\right) + \left(\textstyle\sum_{i \in \bar{A}} \mathbf{x}_y^i + \boldsymbol{\epsilon}_{x,y}^i\right)\right\rrbracket \\ &= \left\llbracket \left(\textstyle\sum_{i \in A} \mathbf{x}_y^i + \boldsymbol{\epsilon}_{x,y}^i\right) + \left(\textstyle\sum_{i \in \bar{A}} \phi(\mathbf{x}_x^i)\right)\right\rrbracket\end{aligned}$$

This means that the new commitment is unrelated to $\mathbf{x}_y^i$ for $i \in \bar{A}$ (since the $\mathbf{x}_y^i$ terms are canceled out for $i \in \bar{A}$). Thus the adversary will not be able to tell the difference of whether we use the correct $\boldsymbol{\epsilon}_{x,y}^i$ as in the H_2, or the random one in H_3. A crucial part of this argument is that H_3 does not actually construct the commitment $\llbracket \mathbf{x}_{y'} \rrbracket$ but rely on the ideal functionality to make this, thus when opening $\llbracket \mathbf{x}_{y'} \rrbracket$ or a linear combination of this, H_3 ensures that the honest parties term is exactly $\left(\sum_{i \in \bar{A}} \phi(\mathbf{x}_x^i)\right)$ by definition.

Finally we argue that H_3 is indistinguishable from the simulation we see that the only difference between the two is that **Open, Partial Open** opens to the values $\mathbf{x}_x$ and $\phi(\mathbf{x}_x)$ in H_3 and we must argue that this is the same in the simulation. First see that $\mathbf{x}_x$ is by definition random and is picked directly from a raw commitment, thus this is the same as calling **Rand**. So we only need to show that **Open** will open the other value correctly, i.e. to $\phi(\mathbf{x}_x)$. So what we need to show is that $\sum_{P_i \in A} \mathbf{x}^i_{y'} = \sum_{P_i \in A} \phi(\mathbf{x}^i_x)$. We only need to show this for the malicious shares since the honest parties do everything correctly and the simulation ensure that the values used for the honest parties are consistent with the actual values opened (by storing exactly the expected value to be opened when one value of the reorganization pair is opened). We see that if $\sum_{P_i \in A} \mathbf{x}^i_{y'} \neq \sum_{P_i \in A} \phi(\mathbf{x}^i_x)$ then we abort step 5 as this step uses uniquely defined shares of the adversary using the same argument of step 3 of Theorem 3.1. ■

G Proof of Theorem 5.1

Consider the following simulator $\mathcal{S}$:

Init, Input, Rand, Add, Public Add, Public Multiply: Simulate the protocol trivially by simply passing on messages from $\mathcal{A}$ to the ideal functionality and vice versa, while internally simulating $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ in accordance with its ideal functionality.

Multiply: Pick $\epsilon, \rho \in \mathbb{F}^m$ uniformly at random and open towards these to $\mathcal{A}$ by trivially simulating $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$.

Reorganize: Pick $\epsilon \in \mathbb{F}^m$ uniformly at random and open towards these to $\mathcal{A}$ by trivially simulating $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$.

Output: Receive $\mathbf{x}$ from the ideal functionality and send this to $\mathcal{A}$. If it does not abort then allow the ideal functionality to output this to the honest parties.

The outputs of the real world and simulation is the same by correctness of $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ and multiplication using Beaver triples. Furthermore, we see that ϵ and ρ are indistinguishable from random in the protocol since they are one-time padded with the values $\mathbf{a}$, respectively $\mathbf{b}$ from a multiplication triple. These are random by the $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ functionality and are never used again. Thus the real world and simulation are indistinguishable. ■

H Issues With [19] When Used as the Preprocessing Phase of MiniMAC [16]

In the following we point out an attack on the sacrificing step in the construction of MiniMAC multiplication triples in [19]. The attack seems to be easily fixable with multiplicative overhead of 3 or 4, in the amount of unchecked triples that must be sacrificed to construct a correct triple. However, more efficient fixes might exist.

The preprocessing of the multiplication triples [19] used in MiniMAC consists of a sacrificing step in which, possibly malformed, triples are paired up and checked. One of the triples in the pair is multiplied with a random value, thus ensuring that a potential error gets randomized. The triples are subtracted from each other and a 0-check is performed (similar to the CorrectnessTest described in Fig. 11).

In the following we first describe their sacrificing method and then describe the issue and a possible fix. Let the values contained in the two triples be denoted by $(\mathbf{x}, \mathbf{y}, \mathbf{z})$ and $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ where $\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{F}^m$ and $\mathbf{z} = \mathbf{x} * \mathbf{y} + \boldsymbol{\Delta}_1$ and $\mathbf{c} = \mathbf{a} * \mathbf{b} + \boldsymbol{\Delta}_2$ for some errors $\boldsymbol{\Delta}_1, \boldsymbol{\Delta}_2 \in \mathbb{F}^m$. A correct triple is one with a zero error. The parties sample a public random value $\mathbf{r} \in \mathbb{F}^m$ and then check that $\mathbf{r} * (\mathbf{z} - \mathbf{x} * \mathbf{y}) + \mathbf{c} - \mathbf{a} * \mathbf{b} = \mathbf{r} * \boldsymbol{\Delta}_1 + \boldsymbol{\Delta}_2 = \boldsymbol{\zeta} = \mathbf{0}$. If this is the case, the parties conclude that $(\mathbf{x}, \mathbf{y}, \mathbf{z})$ is a correct triple and discard $(\mathbf{a}, \mathbf{b}, \mathbf{c})$. Otherwise the parties abort the protocol.

In the following we assume that an adversary can freely determine $\boldsymbol{\Delta}_1$ and $\boldsymbol{\Delta}_2$ (we show later that it can in fact do so with high probability, even though the ideal functionalities in [19] does not exactly allow this). We now describe how an adversary could affect these errors such that the parties end up with an incorrect triple with high probability.

First notice that if the adversary determines $\boldsymbol{\Delta}_1 = (c, 0, \ldots, 0)$ with $c \neq 0$ (i.e. the first component of $\boldsymbol{\Delta}_1$ is some non zero value c and the rest $m-1$ components are zero) and $\boldsymbol{\Delta}_2 = (0, \ldots, 0)$, then the check goes through whenever $\mathbf{r} = 0$, which happens with probability $\frac{1}{|\mathbb{F}|}$. Thus, the parties use an incorrect triple.

Changing the sacrificing step in a way that the triple $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ will be considered as correct (rather than concluding that $(\mathbf{x}, \mathbf{y}, \mathbf{z})$ is correct) and discard $(\mathbf{x}, \mathbf{y}, \mathbf{z})$ does not solve the problem since now the adversary can set $\boldsymbol{\Delta}_2 = -\boldsymbol{\Delta}_1$. That is $\boldsymbol{\Delta}_1 = (c, 0, \ldots, 0)$ and $\boldsymbol{\Delta}_2 = (-c, 0, \ldots, 0)$. Thus, whenever $\mathbf{r} = 1$ the check $\mathbf{r} * \boldsymbol{\Delta}_1 + \boldsymbol{\Delta}_2$ will be $\mathbf{0}$, and $\mathbf{r} = 1$ with high probability of $\frac{1}{|\mathbb{F}|}$ for small fields (i.e. if $|\mathbb{F}| < 2^s$).

One might be tempted to fix this issue by picking $\mathbf{r} \neq 0$ or $\mathbf{r} \neq 1$. However, for $\mathbb{F}$ =GF(2) this actually means that $\mathbf{r}$ is fixed and known to the adversary a priori, which makes the sacrificing step useless. Furthermore, even for a general field $\mathbb{F}$ (which is not GF(2)), the adversary may pick $\boldsymbol{\Delta}_1$ and $\boldsymbol{\Delta}_2$ arbitrarily and hope that $\mathbf{r} * \boldsymbol{\Delta}_1 + \boldsymbol{\Delta}_2 = 0$ which actually means that $\mathbf{r} = (-\boldsymbol{\Delta}_2)(\boldsymbol{\Delta}_1)^{-1}$, as before, this happen with high probability of $\frac{1}{|\mathbb{F}|}$.

This problem in [19] seems to be fixable using the same approach as in our sacrificing step, by constructing triples in a batch and pair them randomly two or three times. This incurs an overhead of a factor 3 or 4 in the construction of a single correct triple.

Determining the errors $\boldsymbol{\Delta}_1$ *and* $\boldsymbol{\Delta}_2$. It is required to argue that the adversary may indeed add two distinctive errors in a single component to $\mathbf{z}$ and $\mathbf{c}$ for the above issue to occur.

The ideal functionality in [19] constructs multiplication triples and allows the adversary to add a *random* error to them. However, that means that a random error is added to all components of $\mathbf{z}$ and $\mathbf{c}$, in contrast to the above attack which requires the adversary to add an error to the first component only.

In the following let $\mathbf{x}^i, \mathbf{y}^i, \mathbf{a}^i, \mathbf{b}^i$ be the additive shares of party P_i for $\mathbf{x}, \mathbf{y}, \mathbf{a}, \mathbf{b}$ respectively. We now notice that if there are at least 2 honest parties, the functionality allows the adversary to set $\boldsymbol{\Delta}_1 = \mathbf{x}^i * \mathbf{s}^i_x + \mathbf{y}^i * \mathbf{s}^i_y$ and $\boldsymbol{\Delta}_2 = \mathbf{a}^i * \mathbf{s}^i_a + \mathbf{b}^i * \mathbf{s}^i_b$ where $\mathbf{s}^i_x, \mathbf{s}^i_y, \mathbf{s}^i_a, \mathbf{s}^i_b \in \mathbb{F}^m$ are the adversary's choice. Since $\mathbf{x}^i, \mathbf{y}^i, \mathbf{a}^i, \mathbf{b}^i$ are random, the probability of the first component of $\mathbf{a}^i$ be 0 is $\frac{1}{|\mathbb{F}|}$. The adversary picks $\mathbf{s}^i_x = \mathbf{s}^i_a = (1, 0, \ldots, 0)$ and $\mathbf{s}^i_y = \mathbf{s}^i_b = \mathbf{0}$. This means that $\boldsymbol{\Delta}_1 = (x^i, 0, \ldots, 0)$ and $\boldsymbol{\Delta}_2 = (a^i, 0, \ldots, 0)$. Let r be the first component of $\mathbf{r}$. This means that $\mathbf{r} * \boldsymbol{\Delta}_1 + \boldsymbol{\Delta}_2 = \mathbf{0}$ whenever $rx^i + a^i = 0$, which happens if $r \neq 0$ and $rx^i = -a^i$ or if $r = x^i = a^i = 0$. The first case happens with probability $\frac{|\mathbb{F}|-1}{|\mathbb{F}|} \cdot \frac{1}{|\mathbb{F}|}$ and the second case happens with probability $\left(\frac{1}{|\mathbb{F}|}\right)^3$ which adds up to $\left(\frac{1}{|\mathbb{F}|}\right)^2$. Thus the adversary succeeds in the attack with probability which is clearly not negligible for small $\mathbb{F}$. For example, for $\mathbb{F}$ =GF(2^8) his success probability is 2^{-16} and for the binary field it is 2^{-4}.

The attentive reader might observe that the description above is oversimplified since all checks are based on encodings of message vectors. In particular this means the potential error vectors $\boldsymbol{\Delta}_1$ and $\boldsymbol{\Delta}_2$ will be encoded in the C, but by the construction of the unchecked triples we have that the encoding of $\mathbf{c}$ and $\mathbf{z}$ will be in the Schur transform. Specifically when all parties are honest we have $C^*(\mathbf{c}) = C(\mathbf{a}) * C(\mathbf{b})$, $C^*(\mathbf{z}) = C(\mathbf{x}) * C(\mathbf{y})$. In case of the adversary adding an error we have $C^*(\mathbf{c}) + C^*(\boldsymbol{\Delta}_1) = C(\mathbf{a}) * C(\mathbf{b}) + C(\mathbf{a}^i) * C(\mathbf{s}^i_a)$, $C^*(\mathbf{z}) + C^*(\boldsymbol{\Delta}_2) = C(\mathbf{x}) * C(\mathbf{y}) + C(\mathbf{x}^i) * C(\mathbf{s}^i_x)$. Thus the error is moved into the Schur space. This means that at least $d^* > s$ positions of $C^*(\boldsymbol{\Delta}_1)$ and $C^*(\boldsymbol{\Delta}_2)$ will be non-zero (if $\boldsymbol{\Delta}_1 \neq \mathbf{0}$, respectively $\boldsymbol{\Delta}_2 \neq \mathbf{0}$. In particular since the message space of the Schur transform, m^*, is greater than the message space m of the C encoding we get that, even if the errors cancel out in the message components of C chosen by the adversary, some remnants of the error might persist in the components $]m; m^*]$. That is, just because $\mathbf{r} * \boldsymbol{\Delta}_1 + \boldsymbol{\Delta}_2 = \mathbf{0}$ it might be the case that $C(\mathbf{r}) * C^*(\boldsymbol{\Delta}_1) + C^*(\boldsymbol{\Delta}_2) \neq \mathbf{0}$. This is so since there are 2^{m^*-m} valid encodings of the error vectors in C^* and the encoding which will be used depends on the shares of the honest party. However, this is unfortunately not clearly detectable by an honest party. The reason being that the honest parties only get a share of the result of the sacrificing, $C^*(\boldsymbol{\zeta})$ thus they do not know whether the values in components $]m; m^*]$ are part of the parity components coming from multiplying correct codewords, or if they are part of remnants of an error in the components $[m]$. Specifically when they open $C^*(\boldsymbol{\zeta})$ by each party sending his/her share and check that it is 0 in the first m components, they cannot simply extend this check to the first m^* components, since the components $]m; m^*]$ are not expected to be 0. This is because the value checked, $C^*(\boldsymbol{\zeta})$ is the result of the computation $C(\mathbf{r}) * C(\mathbf{z}) + C^*(\mathbf{c}) - C(\mathbf{x}) * C(\boldsymbol{\rho}) - C(\mathbf{b}) * C(\boldsymbol{\sigma})$, where all terms are independently computed products in the Schur transform. Thus it is not possible to know what each of the terms contribute to the components $]m; m^*]$. Furthermore, the parties cannot send their shares to each of the terms without leaking too much info on the triple that would otherwise be kept after the sacrificing.

Ideal functionality. We find it appropriate to note that [19] implements ideal preprocessing functionalities which are different from the standard ones required by the MiniMAC online phase in [16] or SPDZ [14]. Specifically the ideal functionalities in [19] give the adversary the power to manipulate an honest party's private share based on the private shares of other honest parties. It is discussed in [19] that the ideal preprocessing functionalities can be used directly with the "standard" MiniMAC [16] and SPDZ [14] protocols. However, no proof is provided for that. Furthermore, in [28] it was insinuated that this is not the case for the input phase of SPDZ. In any case, it does not seem trivial to prove that the preprocessing of [19] can work with the functionalities required by the online phases in the literature. Thus we think that our protocol has an advantage over the MiniMAC protocol of [19] as we both prove our protocol secure with a light and uncomplicated online phase and our ideal preprocessing functionalities fit well with those required in the literature.