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, Aarhus, Denmark
tore.frederiksen@alexandra.dk
[2] Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel
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

T. K. Frederiksen—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.

M. Abdalla and R. Dahab (Eds.): PKC 2018, LNCS 10769, pp. 587–619, 2018.
https://doi.org/10.1007/978-3-319-76578-5_20

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 [24]. 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 [3,5,24]: 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.

Out 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 [14,16,33].

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:

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[1]. 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] 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].

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 [35,36]. This scheme, along with others [9–11] 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 [29], 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 [23] 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 [12,15,16], 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 [14,16,29]. 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.

Efficiency. In 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.

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.

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

1.1 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 [14,29], 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[2] 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 [7,30,33] 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.

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

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[3] 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 [12,15,16]. 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 the full version [20].

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 [1,2,25,28,33], 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 $0.3\,\mu s$ [28].

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 [22] and independently in [11]. A series of follow-up work [6,9,10,18,35] presented several improvements, both asymptotically and practically. Of these works [35] 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[4].

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

[4] 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.

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 [31,32,37], where the parties first construct an encrypted Boolean circuit and then evaluate it locally. Another approach is "MPC-in-the-head" [26,27] 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.

2 Preliminaries

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].

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

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

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) 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.

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 $\Pi_{\mathsf{MPC\text{-}}\mathbb{F}^m}$.

Functionality $\mathcal{F}_{\text{MPC-}\mathbb{F}^m}$: 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 $\mathcal{F}_{\text{MPC-}\mathbb{F}^m}$

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 [23] and used in MASCOT [29]. 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.

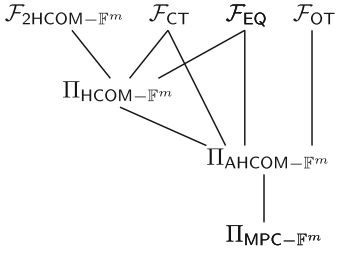

Fig. 5. Outline of functionalities and protocols.

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 $\texttt{transfer}$ command of the $\mathcal{F}_{\mathsf{CT}}$ functionality.

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

Even if using a maliciously secure OT functionality to realize this procedure, it still does not become maliciously secure. We discuss how to handle this in Sect. 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 the full version [20].

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

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

Commit: Upon receiving $(\texttt{commit}, I)$ from P_i where I is a set of γ freshly new identifiers, send the message $(\texttt{commit}, I)$ to $\mathcal{A}$. If $\mathcal{A}$ sends back (no-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 $(\texttt{committed}, \{(k, \mathbf{x}_k)\}_{k \in I})$ to P_i and $(\texttt{committed}, I)$ to P_j and $\mathcal{A}$. If P_i is corrupted and $\mathcal{A}$ instead sends back $(\texttt{corrupt}, \{(k, \bar{\mathbf{x}}_k)\}_{k \in I})$ 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 $(\texttt{Input}, k, \mathbf{y})$ from P_i if $\mathsf{raw}[k] \neq \perp$ then store $\mathsf{raw}[k] = \perp$ and $\mathsf{actual}[k] = \mathbf{y}$. Then send $(\texttt{Input}, k)$ to P_j and $\mathcal{A}$.

Rand: Upon receiving a message $(\texttt{random}, k)$ 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 $(\texttt{random}, k)$ to P_j and $\mathcal{A}$.

Linear Combination: Upon receiving $(\texttt{linear}, (\{(k, \boldsymbol{\alpha}_k)\}_{k \in K}, \boldsymbol{\beta}, k')$ 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 (open, k) from P_i, if $\mathsf{actual}[k] = \mathbf{x}_k \neq \perp$ then send $(\mathsf{opened}, \mathbf{x}_k)$ to P_j and $\mathcal{A}$.

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

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 $\mathtt{actual}[k]$ and $\mathtt{actual}[k']$ 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 the full version [20] 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. To this end, we show how to extend their scheme to supports this operation as well, 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.

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 $\left\{\langle \mathbf{x}^i \rangle^{i,j}\right\}_{j \in [p] \setminus \{i\}}$. In addition, P_i holds the commitment information received from all other parties, $\mathbf{x}^j$ (for $j \neq i$), that is, it holds $\left\{[\mathbf{x}^j]^{j,i}\right\}_{j \in [p] \setminus \{i\}}$.

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

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

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

Rand: Upon receiving a message (`random`, k) from all parties, if $\mathsf{raw}[k] \neq \perp$ then pick a random $\mathbf{x}_k \in_R \mathbb{F}^m$, assign $\mathsf{raw}[k] = \mathbf{x}_k$ and send (`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 $\mathsf{actual}[k] = \mathbf{x}_k \neq \perp$ for all $k \in K$ and $\mathtt{raw}[k'] = \mathtt{actual}[k'] = \perp$ 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 $\mathsf{actual}[k] = \mathbf{x}_k \neq \perp$ then send (`opened`, $k, \mathbf{x}_k$) to $\mathcal{A}$. $\mathcal{A}$ may then abort the protocol, otherwise send (`opened`, $k, \mathbf{x}_k$) to the honest parties.

Partial Open: Upon receiving a message (`open`, i, k) from all parties, if $\mathsf{actual}[k] = \mathbf{x}_k \neq \perp$ then send (`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 $\mathcal{F}_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$

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 $\mathsf{raw}^i[k] = \left\{[\mathbf{x}^j]^{j,i}, \langle \mathbf{x}^i \rangle^{i,j}\right\}_{j\in[p]\setminus\{i\}}$ 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:

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

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

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

and all other parties P_j compute:

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

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

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

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 $\mathcal{F}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$, 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 $\boldsymbol{\epsilon} = \mathbf{y} - \mathbf{x}$ 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 the full version [20].

Theorem 3.1. *Protocol $\Pi_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$ (of Fig. 9) UC-securely realizes functionality $\mathcal{F}_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$ (of Fig. 8) in the $\mathcal{F}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$, $\mathcal{F}_{\mathsf{CT}}$, and $\mathcal{F}_{\mathsf{EQ}}$-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[5]. To this end, we augment functionality $\mathcal{F}_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$ 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

[5] 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 $(\texttt{init})$ from all parties each pair of parties P_i and P_j invoke the command $(\texttt{init})$ 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 $(\texttt{commit}, I')$ and receiving the message $\left(\texttt{committed}, \left\{\left(k, \mathbf{x}_k^{i,j}\right)\right\}_{k\in I'}\right)$. As a result, P_j receives the message $(\texttt{committed}, I')$ from $\mathcal{F}^{j,i}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$ for all $j \neq i$.
3. Every party P_i chooses $\mathbf{x}_k^i \in_R \mathbb{F}^m$ 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 $\left(\texttt{Input}, k, \mathbf{x}_k^i\right)$ and receives back $[\![\mathbf{x}]\!]^i = \left\{[\mathbf{x}_k^j]^{j,i}, \langle \mathbf{x}_k^i\rangle^{i,j}\right\}_{j\in[p]\setminus\{i\}}$ 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 $(\texttt{toss}, \mathbb{F}^{s\times\gamma})$ to functionality $\mathcal{F}_{\mathsf{CT}}$, by which they receive $(\texttt{random}, \mathbf{R})$ 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 $\langle \mathbf{s}_q^i\rangle^{i,j} = \langle \mathbf{x}_q^i\rangle^{i,j} + \sum_{k\in I} \mathbf{R}_{q,k} \cdot \langle \mathbf{x}_k^i\rangle^{i,j} = \langle \mathbf{x}_q^i + \sum_{k\in I} \mathbf{R}_{q,k} \cdot \mathbf{x}_k^i\rangle^{i,j}$ and sends $\langle \mathbf{s}_q^i\rangle^{i,j}$ to P_j. P_j then computes $[\mathbf{s}_q^i]^{i,j} = [\mathbf{x}_q^i]^{i,j} + \sum_{k\in I} \mathbf{R}_{q,k} \cdot [\mathbf{x}_k^i]^{i,j} = [\mathbf{x}_q^i + \sum_{k\in I} \mathbf{R}_{q,k} \cdot \mathbf{x}_k^i]^{i,j}$ and reveals $\mathbf{s}_q^i$.
8. For every $q \in I$ every party P_i computes $\mathbf{c}_q^i = \sum_{j\in[p]} \mathbf{s}_q^j$ and inputs $(\texttt{equal}, i, \mathbf{c}_q^i)$ to $\mathcal{F}_{\mathsf{EQ}}$. If $\mathcal{F}_{\mathsf{EQ}}$ responds with $\texttt{abort}$ or $(\texttt{equal}, \mathbf{s}_q^{\perp}, \ldots, \mathbf{s}_q^p, \texttt{reject})$ in any of these calls then abort, otherwise output $(\texttt{committed}, I)$. For every $k \in I$ store $\mathsf{raw}^i[k] = [\![\mathbf{x}_k]\!]^i$.

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

1. Party P_j (for $j \neq i$) aborts if $\mathsf{raw}^j[k] = \perp$. Otherwise P_j sends $\langle \mathbf{x}_k^j\rangle^{j,i}$ to P_i (using $(\texttt{open}, k)$), who learns $\mathbf{x}_k^j$.
2. Party P_i computes $\mathbf{x}_k = \sum_{j\in[p]} \mathbf{x}_k^j$ 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 $\langle \mathbf{x}^i + \epsilon_k\rangle^{i,j} = \langle \mathbf{x}_k^i\rangle^{i,j} + \epsilon_k$ 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 $[\mathbf{x}_k^i + \epsilon_k]^{i,j} = [\mathbf{x}_k^i]^{i,j} + \epsilon_k$.
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 $\langle \mathbf{x}_k^i\rangle^{i,j}$ to P_j for all $j \neq i$. Then, party P_i obtains $\mathbf{x}_k^j$ from the commitment and the opening information $[\mathbf{x}_k^j]^{j,i}$ and $\langle \mathbf{x}_k^j\rangle^{j,i}$ respectively. Finally P_i computes $\mathbf{x}_k = \sum_{j\in[p]} \mathbf{x}_k^j$.

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

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

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 $\mathcal{F}_{\mathsf{HCOM}\text{-}\mathbb{F}^m}$ 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 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.

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 $(\texttt{mult}, K)$ 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 $q = 1, \ldots, T$:

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 $\left(\texttt{mult}, (X_q, Y_q, Z_q)_{q \in [T]}\right)$ to all parties and $\mathcal{A}$.

ReOrg: Upon receiving a message $(\texttt{reOrg}, \phi, K)$ 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 $\left(\texttt{reOrg}, \{(X_q, Y_q)\}_{q \in [T]}\right)$ to all parties and $\mathcal{A}$.

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

Realizing $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$. The protocol $\Pi_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ is formally presented in Figs. 12 and 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 that should be constructed in one batch for the protocol to be secure. Specifically, in Sect. 4.2 we describe how to implement the **Mult** command and in Sect. 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 Sect. 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 Sect. 2 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 Sect. 2) 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 Sect. 2). 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 Sect. 2). 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 [21] 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 [21]). *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

Procedure CorrectnessTest(t_1, t_2)**.**
Given the two triples $t_1 = (\llbracket \mathbf{a} \rrbracket, \llbracket \mathbf{b} \rrbracket, \llbracket \mathbf{c} \rrbracket)$ and $t_2 = (\llbracket \mathbf{x} \rrbracket, \llbracket \mathbf{y} \rrbracket, \llbracket \mathbf{z} \rrbracket)$ 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 $\llbracket \boldsymbol{\epsilon} \rrbracket = r \cdot \llbracket \mathbf{x} \rrbracket - \llbracket \mathbf{a} \rrbracket$ and $\llbracket \boldsymbol{\rho} \rrbracket = \llbracket \mathbf{y} \rrbracket - \llbracket \mathbf{b} \rrbracket$ and publicly open $\boldsymbol{\epsilon}$ and $\boldsymbol{\rho}$, both in $\mathbb{F}^m$.
3. Locally compute $\llbracket \mathbf{e} \rrbracket = r \cdot \llbracket \mathbf{z} \rrbracket - \llbracket \mathbf{c} \rrbracket - \boldsymbol{\epsilon} * \llbracket \mathbf{b} \rrbracket - \boldsymbol{\rho} * \llbracket \mathbf{a} \rrbracket - \boldsymbol{\rho} * \boldsymbol{\epsilon}$ 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 CorrectnessTest(t_1, t_2)

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.

Security. The correctness and security is explained in [14]. However, for completeness we prove the following lemma in the full version [20], 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 $\{(\llbracket \mathbf{x}_k \rrbracket, \llbracket \mathbf{y}_k \rrbracket, \llbracket \mathbf{z}_k \rrbracket)\}_{k \in [\tau_2]}$. We produce the triple $(\llbracket (\mathbf{x}') \rrbracket, \llbracket \mathbf{y}' \rrbracket, \llbracket (\mathbf{z})' \rrbracket)$ out of that set in the following way: The parties compute

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

which constitute the triple $(\llbracket \mathbf{x}' \rrbracket, \llbracket \mathbf{y}' \rrbracket, \llbracket \mathbf{z}' \rrbracket)$. It is easy to see that $\llbracket \mathbf{x}' \rrbracket$ 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 $[\![\left(\sum_{k\in[\tau_2]} \mathbf{x}_k\right)]\!]$ 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 $[\![\mathbf{z}']\!] = [\![\mathbf{z}_1 + \sum_{k=2}^{\tau_2} \boldsymbol{\epsilon}_k * \mathbf{x}_k + \mathbf{z}_k]\!]$ by a local computation only.

We prove the following lemma in the full version [20]:

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_q^i and P_2 inputs $s_q^0 = r_q$ and $s_q^1 = r_q + y^j$ to the $\mathcal{F}_{\mathsf{OT}}$ functionality. The functionality outputs $s^{x_q^i}$ 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_q^0 or s_q^1 are not in the prime field, i.e. they belong to the gap $\mathsf{gap} = [2^{\lceil \log \mathcal{P} \rceil}] \setminus \mathbb{F}_{\mathcal{P}}$. 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_q^0, s_q^1 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 $s_q^0 \neq r_q$ then this is equivalent to add $s_q^0 \mod \mathcal{P}$, which leads to an incorrect share of z. This case is detected in the sacrificing step.
(b) If the incorrect value is $s_q^1 \neq r_q + y^j$ then this is equivalent to add $s_q^1 \mod \mathcal{P}$. 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_q^1 does. Then this is equivalent to choosing $y^j \in \mathbb{F}_\mathcal{P}$ and $r'_q = s_q^1 - y^j \mod \mathcal{P}$ 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 the full version [20].

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.*

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 $\nu + s$ pairs. Finally, the parties complete s random linear combination tests over the batch by producing

Protocol $\Pi_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$. Describes the implementation of $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ in the $\mathcal{F}_{\mathsf{OT}}$, $\mathcal{F}_{\mathsf{EQ}}$- $\mathcal{F}_{\mathsf{CT}}$-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 $|K| = 2\nu+2s$ for some ν. 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 $|X| = |Y| = \nu$ and $|A| = |B| = s$ and proceed as follows:

1. For each of the ν 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 $(\mathtt{toss}, s \cdot \nu, \mathbb{F})$ to $\mathcal{F}_{\mathsf{CT}}$ and thus learn $(\mathtt{random}, \mathbf{R})$ (when viewing the output as a matrix $\mathbf{R} \in \mathbb{F}^{s\times\nu}$).
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 $\mathbf{R}$:
$$\left[\!\left[\mathbf{s}_q\right]\!\right] = \left[\!\left[\mathbf{x}_{A_q}\right]\!\right] + \sum_{k\in[\nu]} \mathbf{R}_{q,k} \cdot \left[\!\left[\mathbf{x}_{X_k}\right]\!\right] \quad \text{and} \quad \left[\!\left[\bar{\mathbf{s}}_q\right]\!\right] = \left[\!\left[\mathbf{x}_{B'_q}\right]\!\right] + \sum_{k\in[\nu]} \mathbf{R}_{q,k} \cdot \left[\!\left[\mathbf{x}_{Y'_k}\right]\!\right]$$
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 $q \in [\nu]$ 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

a uniformly random matrix $\mathbf{R} \in \mathbb{F}^{s\times\nu}$ (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 $\nu + s$ pairs into two sets A, B of ν and s pairs respectively. For each pair $([\![\mathbf{z}_q]\!], [\![\mathbf{z}_{q'}]\!])$ in B for $q \in s$ compute and open

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

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

Based on this we state the following theorem, which we prove in the full version [20].

Protocol $\Pi_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$. Describes the implementation of $\mathcal{F}_{\mathsf{AHCOM}\text{-}\mathbb{F}^m}$ in the $\mathcal{F}_{\mathsf{OT}}$, $\mathcal{F}_{\mathsf{EQ}}$- $\mathcal{F}_{\mathsf{CT}}$-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 $(\texttt{mult}, K)$ 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 $\mathbf{x} * \mathbf{y} =^? \mathbf{z}$. 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

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 Fig. 10 in the* $\mathcal{F}_{OT}$*-,* $\mathcal{F}_{EQ}$*- and* $\mathcal{F}_{CT}$*-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.

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, $(\llbracket \mathbf{x} \rrbracket, \llbracket \mathbf{y} \rrbracket, \llbracket \mathbf{z} \rrbracket)$ and reorganization pairs $(\llbracket \mathbf{x} \rrbracket, \llbracket \phi(\mathbf{x}) \rrbracket)$.

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 $\llbracket \mathbf{y}_k \rrbracket$.

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 $\llbracket \mathbf{x}_k \rrbracket$.

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

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

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

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

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

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

Output: The parties open the value $\llbracket \mathbf{x} \rrbracket$ 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.

We prove the following theorem in the full version [20]:

Theorem 5.1. *The protocol in Fig. 14 UC-securely implements the functionality* $\mathcal{F}_{\mathsf{MPC}\text{-}\mathbb{F}^m}$ *of 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

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] can be used as an implementation of functionality $\mathcal{F}_{\mathsf{2HCOM}\text{-}\mathbb{F}^m}$. Briefly, in this two party commitment 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 $[\![\mathbf{x}]\!]$: First, every party P_i sends its share $\mathbf{x}^i$ to P_1. Then 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 [29]. 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 $s = 40$.

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.* ([28] for OT) and of Frederiksen *et al.* ([18], for two-party homomorphic commitments). It 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 [35].

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) [29]. We assume that the computational security parameter $\kappa \geq m\log(|\mathbb{F}|)$ some complexities increase. $\mathbb{F} = GF(2)$. For [7,29] $m = 1$ is possible. We assume at least 2^{21} triples are generated which gives the smallest numbers to the protocols. *) Using optimization 4. in Sect. 6, requiring $|\mathbb{F}| \geq 2^s$.

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

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 [7,19,29] 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 [19,29] (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 [19,29] 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 the full version [20].

7 Applications

Practically all maliciously secure MPC protocols require some form of commitments. Some, e.g. the LEGO family of protocols [17,18,34,35], 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 $\mathbb{F} = \mathrm{GF}(2)$ 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 $\mathbb{F} = \text{GF}(2)$ (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 [7,33] 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. Asharov, G., Lindell, Y., Schneider, T., Zohner, M.: More efficient oblivious transfer and extensions for faster secure computation. In: ACM CCS, pp. 535–548 (2013)
2. Asharov, G., Lindell, Y., Schneider, T., Zohner, M.: More efficient oblivious transfer extensions with security for malicious adversaries. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9056, pp. 673–701. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46800-5_26
3. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 420–432. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-46766-1_34
4. Beaver, D.: Correlated pseudorandomness and the complexity of private computations. In: STOC, pp. 479–488 (1996)
5. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In: STOC, pp. 1–10 (1988)
6. Brandão, L.T.A.N.: Very-efficient simulatable flipping of many coins into a well (and a new universally-composable commitment scheme). In: Cheng, C.-M., Chung, K.-M., Persiano, G., Yang, B.-Y. (eds.) PKC 2016. LNCS, vol. 9615, pp. 297–326. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49387-8_12
7. Burra, S.S., Larraia, E., Nielsen, J.B., Nordholt, P.S., Orlandi, C., Orsini, E., Scholl, P., Smart, N.P.: High performance multi-party computation for binary circuits based on oblivious transfer. IACR Cryptology ePrint Archive, 2015:472 (2015)
8. Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: FOCS, pp. 136–145 (2001)
9. Cascudo, I., Damgård, I., David, B., Döttling, N., Nielsen, J.B.: Rate-1, linear time and additively homomorphic UC commitments. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016. LNCS, vol. 9816, pp. 179–207. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53015-3_7

10. Cascudo, I., Damgård, I., David, B., Giacomelli, I., Nielsen, J.B., Trifiletti, R.: Additively homomorphic UC commitments with optimal amortized overhead. In: Katz, J. (ed.) PKC 2015. LNCS, vol. 9020, pp. 495–515. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46447-2_22
11. Damgård, I., David, B., Giacomelli, I., Nielsen, J.B.: Compact VSS and efficient homomorphic UC commitments. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014. LNCS, vol. 8874, pp. 213–232. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45608-8_12
12. Damgård, I., Lauritsen, R., Toft, T.: An empirical study and some improvements of the MiniMac protocol for secure computation. In: Abdalla, M., De Prisco, R. (eds.) SCN 2014. LNCS, vol. 8642, pp. 398–415. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10879-7_23
13. Damgård, I., Orlandi, C.: Multiparty computation for dishonest majority: from passive to active security at low cost. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 558–576. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14623-7_30
14. Damgård, I., Pastro, V., Smart, N., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 643–662. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32009-5_38
15. Damgård, I., Zakarias, R.: Fast oblivious AES a dedicated application of the MiniMac protocol. In: Pointcheval, D., Nitaj, A., Rachidi, T. (eds.) AFRICACRYPT 2016. LNCS, vol. 9646, pp. 245–264. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-31517-1_13
16. Damgård, I., Zakarias, S.: Constant-overhead secure computation of Boolean circuits using preprocessing. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 621–641. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36594-2_35
17. Frederiksen, T.K., Jakobsen, T.P., Nielsen, J.B., Nordholt, P.S., Orlandi, C.: MiniLEGO: efficient secure two-party computation from general assumptions. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 537–556. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38348-9_32
18. Frederiksen, T.K., Jakobsen, T.P., Nielsen, J.B., Trifiletti, R.: On the complexity of additively homomorphic UC commitments. In: Kushilevitz, E., Malkin, T. (eds.) TCC 2016. LNCS, vol. 9562, pp. 542–565. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49096-9_23
19. Frederiksen, T.K., Keller, M., Orsini, E., Scholl, P.: A unified approach to MPC with preprocessing using OT. In: Iwata, T., Cheon, J.H. (eds.) ASIACRYPT 2015. LNCS, vol. 9452, pp. 711–735. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48797-6_29
20. Frederiksen, T.K., Pinkas, B., Yanai, A.: Committed MPC - maliciously secure multiparty computation from homomorphic commitments. IACR Cryptology ePrint Archive, 2017:550 (2017)
21. Furukawa, J., Lindell, Y., Nof, A., Weinstein, O.: High-throughput secure three-party computation for malicious adversaries and an honest majority. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10211, pp. 225–255. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56614-6_8
22. Garay, J.A., Ishai, Y., Kumaresan, R., Wee, H.: On the complexity of UC commitments. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 677–694. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-55220-5_37

23. Gilboa, N.: Two party RSA key generation. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 116–129. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48405-1_8
24. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or a completeness theorem for protocols with honest majority. In: STOC, pp. 218–229 (1987)
25. Ishai, Y., Kilian, J., Nissim, K., Petrank, E.: Extending oblivious transfers efficiently. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 145–161. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45146-4_9
26. Ishai, Y., Kushilevitz, E., Ostrovsky, R., Sahai, A.: Zero-knowledge from secure multiparty computation. In: STOC, pp. 21–30 (2007)
27. Ishai, Y., Prabhakaran, M., Sahai, A.: Secure arithmetic computation with no honest majority. In: Reingold, O. (ed.) TCC 2009. LNCS, vol. 5444, pp. 294–314. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00457-5_18
28. Keller, M., Orsini, E., Scholl, P.: Actively secure OT extension with optimal overhead. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9215, pp. 724–741. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47989-6_35
29. Keller, M., Orsini, E., Scholl, P.: MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In: ACM CCS, pp. 830–842 (2016)
30. Larraia, E., Orsini, E., Smart, N.P.: Dishonest majority multi-party computation for binary circuits. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. LNCS, vol. 8617, pp. 495–512. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44381-1_28
31. Lindell, Y., Pinkas, B.: An efficient protocol for secure two-party computation in the presence of malicious adversaries. In: Naor, M. (ed.) EUROCRYPT 2007. LNCS, vol. 4515, pp. 52–78. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72540-4_4
32. Lindell, Y., Pinkas, B., Smart, N.P., Yanai, A.: Efficient constant round multi-party computation combining BMR and SPDZ. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9216, pp. 319–338. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48000-7_16
33. Nielsen, J.B., Nordholt, P.S., Orlandi, C., Burra, S.S.: A new approach to practical active-secure two-party computation. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 681–700. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32009-5_40
34. Nielsen, J.B., Orlandi, C.: LEGO for two-party secure computation. In: Reingold, O. (ed.) TCC 2009. LNCS, vol. 5444, pp. 368–386. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00457-5_22
35. Nielsen, J.B., Schneider, T., Trifiletti, R.: Constant round maliciously secure 2PC with function-independent preprocessing using LEGO. In: NDSS (2017)
36. Rindal, P., Trifiletti, R.: SplitCommit: implementing and analyzing homomorphic UC commitments. IACR Cryptology ePrint Archive, 2017:407 (2017)
37. Yao, A.C.-C.: How to generate and exchange secrets (extended abstract). In: FOCS, pp. 162–167 (1986)