A Framework for Outsourcing of Secure Computation
(Revised Version)*

Thomas P. Jakobsen, Jesper Buus Nielsen, and Claudio Orlandi

Aarhus University, Denmark, {tpj,jbn,orlandi}@cs.au.dk

Abstract We study the problem of how to efficiently outsource a sensitive computation on secret inputs to a number of untrusted workers, under the assumption that at least one worker is honest.
In our setting there are a number of clients $C_1, \ldots, C_n$ with inputs $x_1, \ldots, x_n$. The clients want to delegate a secure computation of $f(x_1, \ldots, x_n)$ to a set of untrusted workers $W_1, \ldots, W_m$. We want do so in such a way that as long as there is at least one honest worker (and everyone else might be actively corrupted) the following holds:

1. the privacy of the inputs is preserved;
2. the output of the computation is correct (in particular workers cannot change the inputs of honest clients).

We propose a solution where the clients' work is minimal and the interaction pattern simple (one message to upload inputs, one to receive results), while at the same time reducing the overhead for the workers to a minimum. Our solution is generic and can be instantiated with any underlying reactive MPC protocol where linear operations are "for free". In contrast previous solutions were less generic and could only be instantiated for specific numbers of clients/workers.

1 Introduction

In this paper we will use the term *secure multiparty computation* (or MPC for short) to refer to any problem where a number of parties wants to compute a function f on inputs $x_1, \ldots, x_n$ while guaranteeing interesting security properties such as the privacy of the inputs and the correctness of the outputs. In particular we will consider the setting where n parties (the clients) provide inputs and receive outputs, in the presence of m additional parties (the workers) who act as helpers to reduce the computational burden on the clients. Clients do not trust each other, and they wish to trust the workers as little as they have to.

A notable example is the case of *verifiable delegation of computation* [GGP10,PHGR13,BSCG+13,BFR13] where one (or more) computationally bounded clients want to perform a computation on an untrusted cloud provider, and therefore wish to perform this computation in a way that the work required to verify the correctness of the result is much less than the work needed to compute the function itself, while also protecting the privacy of the inputs. Traditionally, the problem of verifiable delegation of computation is studied in the presence of a single untrusted worker. However in this case the only known way of protecting the privacy of the inputs is by using fully-homomorphic encryption schemes. This introduces a huge computational overhead for the worker. If one is interested *only* in verifying the correctness of the output, recent studies show that using SNARKs (succinct non-interactive arguments of knowledge) can be made much more practical than expected a few years ago [PHGR13, BSCG+13].

Another important application is *large-scale* secure computation, where one wants to run a secure computation on thousands or millions of secure inputs. In this setting a (significant) number of clients $C_1, \ldots, C_n$ with inputs $x_1, \ldots, x_n$, want to securely evaluate $f(x_1, \ldots, x_n)$. However running any existing MPC protocols for general functionalities between all the clients would require that all parties are online at the same time [HLP11], and the communication overhead of every practical protocol for dishonest majority scales quadratically with the number of parties. Instead, the clients can delegate their computation to a (small)

* The protocol described in the proceeding version of this paper is susceptible to a selective failure attack which is fixed in this version. We thank Berry Schoenmakers for pointing out the problem.

set of untrusted workers $W_1, \ldots, W_m$. This is a relevant scenario in practice, and many real-world uses of secure computation follow this paradigm, e.g., the Danish sugar beet auction [BCD+09], Sharemind [BLW08], MEVAL [CMF+14], etc. A limitation of these solutions is that they require a majority of the workers to be honest and only guarantee security against passive corruptions (in particular, a dishonest worker can arbitrarily change the input of an honest client).

In this work instead we want to consider solutions which are secure when all but one of the workers are corrupted: this allows to use less workers to achieve the same security, which might be important in practice as the main cost of the system (probably) will be the price to rent computing time on the workers. Since we want to tolerate that all but one of the workers can be corrupted, we cannot use a protocol that guarantees termination. In fact, if we want to tolerate that all but one worker might be corrupted, it must provably be the case that a single worker can deadlock the system. This, however, can be detected and then other workers can be rented next time. However, our protocol guarantees termination whenever all workers are honest, independently of how many clients are corrupted.

There is a lot of prior work looking at this and related problems, both in terms of concrete [KMR11, KMR12, PTK13, CLT14, KMRS14] and asymptotic efficiency [Gen09, BV11, LTV12, GHRW14]. We will compare to related work of the first kind after presenting our protocol. The latter kind of work heavily relies on advanced cryptographic tools such as fully-homomorphic encryption: while this "swiss-army knife" of cryptography allows for wonderful and surprising results in terms of feasibility and asymptotic complexity, it introduces a huge computational overhead for the workers and therefore it is worth studying alternative solutions that can be used in practice.

2 Technical Overview

We want to make sure that the work performed by the clients is minimal – in particular *independent of the size of the function to be computed*. As already discussed, this is possible (and with optimal asymptotic efficiency) using fully-homomorphic encryption. However this will incur a huge computational overhead for the workers. So, following the approach of [KMR11, KMR12, PTK13, CLT14] we seek for a protocol where the client has to trust that at least one of the workers is honest. Moreover, instead of designing a specific protocol to solve the problem, we propose a more generic approach to this problem, which can be instantiated using different building blocks depending on the particular application. This gives more flexibility and allows for a greater range of applications (for instance, solutions based on garbled circuits are typically limited to two parties).

Our main building block will be a protocol for reactive secure computation (that is, a protocol where it is possible to open intermediate values) and where linear operations are for free. Many protocols of this kind are known (e.g., [DO10, BDOZ11, NNOB12, DPSZ12, DKL+13])[1].

It is clear that the overall efficiency will be highly impacted by the efficiency of the underlying protocol, and in this paper we do not try to improve on this (but there is plenty of ongoing research on the subject). Instead, we consider *only* the (somewhat orthogonal) problem of how to let clients provide inputs to the workers in such a way that the clients' work is minimal and the overhead induced on the underlying MPC protocol is as limited as possible. We believe this modular approach is useful, both from a conceptual point of view, and also from a practical point of view e.g., one can imagine that improvements on the underlying MPC protocols for the workers would not require one to update the software on the client side.

We describe now the main idea of our framework. It will be instructive to think of a simple setting where one client, running on a computationally limited device (e.g., a mobile phone) wants to delegate a computation to two workers (e.g., two cloud providers) which are not fully trusted. The client has input x and the workers have no input. At the end of the protocol the client is supposed to learn $z = f(x)$ for some public function f (later on, in the multiclient setting, we will also ask that each client learns *only* this output and nothing else about the inputs of the other parties). The workers should not learn anything.

[1] Also Yao's protocol can be made to fit this framework using standard techniques [HL10].

2.1 A Simple but Inefficient Solution

There is a very simple solution to prevent the workers from learning the input/output of the computation, namely we can let the client additively secret share its inputs between the two workers W_1, W_2: the client C picks random x_1, x_2 such that $x_1 + x_2 = x$ and sends x_i to W_i. In addition, to make sure neither W_1, W_2 learn the output of the function, we let C send one-time pads r_1, r_2 to W_1, W_2.

Now W_1, W_2 run their favorite secure computation protocol (which guarantees security against actively corrupted parties) to securely evaluate the function

$$g((x_1, r_1), (x_2, r_2)) = f(x_1 + x_2) + r_1 + r_2$$

and send the output to C_1 who can reconstruct the output by removing the pads. Note that under the assumption that linear operations are "for free" securely evaluating g is as efficient as securely evaluating f.

This solution only works if at most one of W_1, W_2 is passively corrupted, as a malicious adversary can input a share $\tilde{x}_i = x_i + \epsilon_x$ to the secure computation, thus being able to add an error ϵ to the client's input (or $\tilde{r}_i = r_i + \epsilon_r$, thus adding an error to the output). This can be fixed by having the client send a (shared) MAC together with his shares. That is, now the client picks a key k, computes a MAC $t = \mathsf{Tag}(k, (x, r))$ and secret shares the MAC between the two workers, who now run an MPC protocol for the function:

$$g'(\ (x_1, t_1, k_1, r_1), (x_2, t_2, k_2, r_2)\ ) := \begin{cases} f(x, y) + r, & \text{if } \mathsf{Ver}(t, k, (x, r)) = 1 \\ \texttt{abort}, & \text{else} \end{cases}$$

where $x = x_1 + x_2, r = r_1 + r_2, t = t_1 + t_2$ and $k = k_1 + k_2$. Unfortunately this requires that the MAC is verified by the secure computation protocol, and this will increase the circuit size significantly. Even using simple, information theoretic MACs might add a significant number of multiplications to the workers' computation. An additional problem is that a corrupted worker could change its share of the key k, so a plain MAC is not sufficient.

2.2 Our Solution

The right MAC scheme. Our solution relies on the following observation: in the previous protocol MACs and keys are only required to check that none of the workers lie about the client's inputs. After those values are given as input to the secure computation protocol, they are essentially "committed" and cannot be changed anymore. Therefore, at this point the key can simply be revealed, which together with a careful choice of MAC scheme will turn the MAC verification into a linear computation which does not have any significant impact on the efficiency of the overall protocol. In particular, the MAC needs to satisfy two properties:

1. The MAC should be a linear function of the message i.e., the function $t = \mathsf{Tag}(k, \boldsymbol{x})$[2] should be of the form
$$t = h_0(k) + \sum_{i=1}^{\ell} h_i(k) \cdot x_i$$
for some (arbitrary) function $h_0, \ldots, h_\ell$ (since this function can be computed by the workers "in public" on the revealed key k).
2. The MAC should be secure against *algebraic manipulation*: in the overall protocol, any corrupted worker can add arbitrary errors to the key, the tag and the message i.e., in the presence of corrupted workers the actual MAC verification will be
$$\tilde{t} \stackrel{?}{=} \mathsf{Tag}(\tilde{k}, \tilde{\boldsymbol{x}})$$

[2] Here $\boldsymbol{x} = (x_1, \ldots, x_\ell)$ represents the vector to be MAC'ed e.g., $\boldsymbol{x} = (x, r)$ in the description so far

where $\tilde{t} = t + \epsilon_t, \tilde{k} = k + \epsilon_k$ and $\tilde{\boldsymbol{x}} = \boldsymbol{x} + \boldsymbol{\epsilon_x}$ for adversarially chosen $(\epsilon_t, \epsilon_k, \boldsymbol{\epsilon_x})$. However note that, since there is at least one honest worker (and the clients secret share the tag and key), the corrupted workers do not get to see t and k before choosing ϵ_t and ϵ_k (on the other hand, we make no assumption that the adversary does not know $\boldsymbol{x}$). All in all, we need a MAC scheme with the following security property: the adversary chooses messages $\boldsymbol{x}$ and errors $(\epsilon_t, \epsilon_k, \boldsymbol{\epsilon_x})$, then a random key k is sampled and the adversary wins only if

$$\mathsf{Tag}(k, \boldsymbol{x}) + \epsilon_t = \mathsf{Tag}(k + \epsilon_k, \boldsymbol{x} + \boldsymbol{\epsilon_x}) \ . \tag{1}$$

This is exactly the notion of an *algebraic manipulation detection code* of [CDF+08], and in the same paper the following (optimal) construction is given:

$$t = \mathsf{Tag}(k, \boldsymbol{x}) = k^{\ell+2} + \sum_{i=1}^{\ell} k^i \cdot x_i \ ,$$

where the computation is performed in a large field $\mathbb{F}$. In a nutshell, if $(\epsilon_k, \epsilon_t, \boldsymbol{\epsilon_x}) \neq (0, 0, \mathbf{0})$ then the equality in (1) can be rewritten as $p(k) = 0$ for a non-zero polynomial p of degree at most $\ell + 1$, and since k is chosen at random the equality holds with probability at most $(\ell+1)|\mathbb{F}|^{-1}$. Note that this holds for all $\boldsymbol{x}$ (even adversarially chosen), and therefore implies that no selective-failure attacks are present (i.e., attacks where the validity of the MAC check depends on the inputs of the clients). In particular, if $\epsilon_k \neq 0$ then $p(k)$ is a polynomial of degree *exactly* $\ell + 1$, independently of the value of $\boldsymbol{x}$. Note finally that computing the MAC only involves linear operation in $\boldsymbol{x}$, as required by Property 1.

High level description of the protocol. We are now ready to provide a high level description of our generic framework for outsourcing of computation.

Client Input Phase: Let $\mathbb{F}$ be a sufficiently large finite field, which is *efficient to compute in securely* using the underlying MPC protocol.[3] Assume that all inputs are from $\mathbb{F}$. (If not, simply parse the input as several elements from $\mathbb{F}$ and continue as follows for each element.) Each client C_j, on input $x_j \in \mathbb{F}$ samples a random mask $r_j \in \mathbb{F}$, defines $\boldsymbol{x_j} = (x_j, r_j)$, picks a random key $k_j \in \mathbb{F}$ and computes $t_j = \mathsf{Tag}(k_j, \boldsymbol{x_j})$ as described above. Now every client computes an additive secret sharing (over $\mathbb{F}$) of $(t_j, k_j, \boldsymbol{x_j})$ and sends a share to each worker W_i.

Workers Computation Phase: All workers $\{W_i\}$ input the shares they receive to the MPC protocol, then they "open" the keys k_j's. Once k_j are opened they check that the tags are correct, i.e., they compute (and open) the value

$$\beta = s \cdot \left(\sum_j \left(\mathsf{Tag}(k_j, \boldsymbol{x_j}) - t_j \right) \right) \ ,$$

where s is a secret random field element. If the output is not 0, the workers output abort. Else, the workers compute and open the "encrypted outputs"

$$c_j = f(x_1, \ldots, x_n) + r_j \ ;$$

Client Output Phase: Finally each worker W_i sends the output c_j to each client client C_j. If client C_j receives the same output from *all* workers, he outputs the unmasked value $z_j = c_j - r_j$, else output `abort`.

[3] By $\mathbb{F}$ being *efficient to compute in securely* we mean that taking a multiplication between a secret value and a public value should be very efficient. As an example, if the underlying protocol is Yao garbled circuits, we can take $\mathbb{F} = GF(2^\kappa)$. Then multiplication with a public value will just be taking XOR of some of the bits of the secret value, which is essentially for free using the *free XOR* technique [KS08]: no communication and no additional data storage. If the underlying protocol is based on secret sharing or encrypted data over some field, then $\mathbb{F}$ can be taken to be that field.

The protocol is secure for the client as long as at most $n-1$ workers are (actively) corrupted: in a nutshell, a corrupted W_i cannot change its share without breaking the security of the MAC scheme. Of course, a corrupted worker can make the protocol abort and prevent termination, but this is unavoidable in the dishonest majority setting.[4] Note that revealing the key of the MAC has no impact on the security, as by that time a corrupted worker has already committed to his share of the input and the MAC. Moreover, no (selective failure) attacks can be mounted: using a wrong key as input to the protocol always makes the protocol abort: when $\beta \neq 0$, then β is uniformly random thanks to s; moreover, if there is at least one j such that $\mathsf{Tag}(k_j, \boldsymbol{x_j}) \neq t_j$, then $\beta \neq 0$ i.e., different MAC errors for honest clients do not cancel out, since the MACs are computed using independently random keys.

Finally, as the output is masked, the workers do not learn any information about it. Note that a corrupted worker might try to modify the output value by sending $\tilde{c}_j \neq c_j$ to the client during the output phase – this could be solved by adding a MAC to the output, but in fact a simpler way exists, namely having the client simply check that all the outputs he receives are the same (remember, at least one of the W_i's is honest). Of course a malicious worker can then prevent the client from getting the output, but this is possible anyway as we assume that all but one worker might be corrupted, in which case it is impossible to guarantee termination of the secure multiparty computation protocol run between the workers. Hence a malicious worker might just make the secure computation deadlock, which would have the same effect of the client not getting its output.

In terms of efficiency, the **Workers Computation Phase** requires only (on top of the complexity of securely computing f) a few openings and a single multiplication used to randomize the MAC checks.

The framework can be used in several settings, choosing appropriate number of clients and workers. As discussed earlier, the single-client/many-workers setting can be used for private and verifiable delegation of computation. This is to be compared with single-server verifiable delegation of computation protocols [PHGR13, BSCG+13], which is getting extremely close to practice if one is only interested in correctness of the result, but requires the use of FHE to achieve input privacy. In addition, current solutions do not extend to the case of multiple clients, while our solutions naturally generalizes. The multiple-client/few-workers setting can be used for large-scale secure computation.

Finally, we note that while it is clear that it is not possible to achieve any privacy if all workers are corrupted, it is possible to achieve (a flavour of) correctness by combining the techniques of this paper with those in [BDO14].

2.3 Extensions

In Section 4 we informally describe some extensions of our framework.

Dealing With Malicious Clients. As we will show in Section 4.1, using our framework we can even guarantee termination in the relevant setting where a malicious client is trying to make the computation abort by using an invalid input. This is particularly relevant when the numbers of clients is much bigger than the number of workers, and it therefore is undesirable that a single, corrupted client can make the whole computation abort. Think, e.g., of an electronic election: a single invalid vote should not prevent all honest parties from reaching a consensus, instead it should be counted as a void vote. This of course introduces new challenges, as a malicious worker should not be able to claim that a client is corrupted and therefore replace the input of a honest party.

The main idea behind our solution is the following: If multiple clients are present, we check that *all* of the MACs are valid using a single multiplication. If $\beta \neq 0$ we recursively split the MACs in half and we search for the incorrect MACs. This takes $\log n$ multiplications times the number of incorrect MACs. Once we identify the set of incorrect MACs, we need to decide whether this is due to a corrupted worker or a corrupted client. Note that this is not trivial: one might think that it is enough to let clients sign their messages, but then a malicious worker could claim he did not receive the signature. See Section 4.1 for our solution.

[4] In a later section, we will discuss how to distinguish between a cheating client and a cheating worker. This will be useful in the multi-client setting.

Eventual Output Consensus. In Section 4.2 we address a final issue: in the above protocols a single corrupted worker can make some honest clients output the correct output while making other honest clients output "abort". This is addressed by letting the workers send all the (encrypted) outputs to all clients along with some signatures. Now, every time an honest client outputs a value $\neq \perp$, this client forwards the outputs and the signature to all clients. Thus, eventually, if one honest client gets an output then all honest clients get an output.

2.4 Relationship To Previous Work

Here we discuss the relationship with some of the most relevant work in this area. Kamara et al. [KMR11] studied the problem of server-aided secure computation with relaxed security guarantees (i.e., non-collusion between corrupted parties). In their solution one of the clients also acts as a worker (i.e., it performs computation linear in the circuit size), therefore we interpret this as a setting with two workers (where one of the two happens to have an input as well). They also show how to transform protocols for secure delegation of computation into protocols in the server-aided model, but also this requires clients interacting with each other. The server-aided model with non-colluding parties was also studied in [KMR12], which gives an efficient protocol based on Yao garbled circuits in this setting. The protocol still requires some interaction between the clients. Subsequently Carter et al. [CMTB13] claimed a number of improvements over Kamara et al., but also in their solution the clients need to interact further than the simple *upload input/download output* pattern of our scheme, and also require additional non-collusion assumptions. A recent work known as Whitewash [CLT14] improves this by presenting a protocol that is secure when the client and one of the workers collude. We do not see how to modify Whitewash to allow for multiple clients or workers. In addition, the output message of Whitewash has an extra factor k overhead, whereas in our protocol the output message from the workers to the clients is of the same size as the output of the function. Finally Peter et al. [PTK13] propose a protocol based on a variant of Paillier encryption for the setting of many clients and two workers. However the PTK protocol only offers passive security, and it is not clear whether it can be extended to more than 2 workers.

Compared to all above mentioned protocols, we find our solution to be 1) more elegant, as it decouples the problem of clients providing inputs to the problem of workers performing the computation 2) more flexible, as it supports any number of clients and workers, and it allows the workers to chose the best possible protocols to perform the secure computation at hand, without having to modify the protocol at the client side 3) more efficient for the client, as the interaction pattern is minimal and the clients do not need to perform any cryptographic operations (only simple field operations) 4) more secure, as security holds up to $m-1$ actively corrupted workers and no non-collusion assumptions need to be made (that is, we guarantee security even when the corrupted parties share data among each other).

It is worth noting that these advantages are achieved without sacrificing the overall efficiency of the system: interestingly all previous solutions seem to obtain protocols that are *essentially as efficient* as the underlying protocol. This is also true in our case, as we only increase the size of the secure computation circuit by a single secure multiplication when there is no cheating (and $\log(n)$ additional multiplications if detection of corrupted clients is desired). A summary of this comparison can be found in Table 1.

3 Our Framework

3.1 Notation and Preliminaries

Let $\mathbb{F}$ be a finite field with $\log|\mathbb{F}| > \kappa$, with κ the security parameter. We write $x \leftarrow \mathbb{F}$ to say that x is sampled uniformly from a finite field $\mathbb{F}$. When we write $x + y$, the addition refers to the finite field $\mathbb{F}$ (therefore, $k \leftarrow \mathbb{F}$, $c = x + k$ is a "one-time-pad" of x). We want to compute a function f described by an arithmetic circuit (multiplications and additions) over $\mathbb{F}$.

Protocol	n	m	Security	Based on	Client Work	Client Interaction	Notes
This	any	> 1	Active	any	$m \cdot size_{inp}$ (fo)	N	
PTK [PTK13]	any	2	Passive	Paillier	$size_{inp}$ (pk)	N	
Whitewash [CLT14]	1	2	Active	GC	$size_{inp}$ (sk)	N	
CMTB [CMTB13]	1	2	Active	GC	$size_{inp}$ (pk)	Y	Non-collusion
Salus [KMR12]	any	2	Active	GC	$size_{inp}$ (sk)	Y	Non-collusion

Table 1. Comparison with previous work. n, m are the number of allowed clients and workers respectively. In the Client Work column, (pk/sk/fo) indicate whether the client needs to perform public key operations, secret key operations or simple field operations. The Interaction column states N if clients interaction is limited to sending/receiving one message to/from the workers or Y otherwise. The work of the client in our solution is linear in m, but in all other rows m is a constant.

We divide our parties into clients $C_1, \ldots, C_n$ and workers $W_1, \ldots, W_m$. Each client has an input of λ field elements.[5] Workers have no inputs. Note that clients and workers need not be disjoint sets. The function

$$f : \mathbb{F}^{\lambda \times n} \to \mathbb{F}^n$$

is publicly agreed upon by all clients and workers and takes n inputs of size λ (one from each client) and outputs a single field element to each client (it is trivial to extend this to the case of longer outputs). We write $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$ to denote the input vectors of the clients, with $\boldsymbol{x_j} \in \mathbb{F}^\lambda \ \forall j$, and we write $\boldsymbol{z} = f(\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}) \in \mathbb{F}^n$.

In the protocol we will actually compute an enhanced function:

$$\hat{f} : \mathbb{F}^{(\lambda+1) \times n} \to \mathbb{F}^n \ ,$$

where each client C_j inputs also a random $r_j \leftarrow \mathbb{F}$ and where the (public) output is $\boldsymbol{c} = \boldsymbol{z} + \boldsymbol{r}$.

In the protocol the clients send a single message (to each worker), then the workers perform some joint computation (using interaction) and finally send a single message to all clients. Therefore the communication pattern for the clients is optimal: one message to provide input, one to receive output. We require in addition that the workload of the clients should not depend on the size of the function f to be evaluated and in the protocol the size of the input message will be $(\lambda + 3) \cdot m$ field elements, while the size of the received output message is m field elements.

We use AMD codes from [CDF+08]. In particular we will use the following notation:

Key Generation: $k \leftarrow \mathsf{Gen}(1^\kappa)$ outputs a random field element $k \leftarrow \mathbb{F}$;

Tag Generation: On input a vector $\boldsymbol{x}$ of field elements with $|\boldsymbol{x}| = \ell$ and a key k the function $t \leftarrow \mathsf{Tag}(k, \boldsymbol{x})$ outputs

$$t = k^{\ell+2} + \sum_{h=1}^{\ell} x_h k^h \ .$$

Verification: On input a tag $t' \in \mathbb{F}$, a key $k' \in \mathbb{F}$ and a vector $\boldsymbol{x'}$ with $|\boldsymbol{x'}| = \ell$ the function $\mathsf{Ver}(t', k', \boldsymbol{x'})$ outputs 1 iff $t' = \mathsf{Tag}(k', \boldsymbol{x'})$.

We only assume that at least one of the workers is honest. In particular, clients are *not* assumed to be honest. We only consider static corruptions. We prove security in the UC framework [Can01].

3.2 The Underlying MPC Protocol

As discussed before, our framework can be instantiated with any secure computation protocol that allows for reactive computation and where linear operations (additions) are "for free" that is, their efficiency can

[5] We will assume all clients input the same number of field elements. Our protocol can be trivially extended to handle clients with different input sizes.

be essentially ignored when considering the overall complexity of the protocol. In order to keep generality, we will describe our protocol assuming that the workers have access to an ideal functionality for reactive computation as in Figure 1. Thanks to the UC composition theorem, one can replace the functionality with any protocol that UC-implements it, and the overall protocol will still be secure. This allows for a modular presentation and to separate the issues of the clients interacting with the workers (giving inputs and receiving outputs) without worrying about which specific protocol is used by the workers.

Functionality $\mathcal{F}_{\text{MPC}}$
The functionality is for the parties $W_1, \ldots, W_m$.

Rand: On input $(rand, W_i, vid)$ from all parties W_i, with vid a fresh identifier, the functionality outputs $(rand, vid)$ to the adversary. On a later input $(complete, vid)$ from the adversary, the functionality samples $r \leftarrow \mathbb{F}$, stores (vid, r) and outputs $(complete, vid)$ to all parties.

Input: On input $(input, W_i, vid, x \in \mathbb{F})$ from W_i and $(input, W_i, vid, ?)$ from all other parties, with vid a fresh identifier, the functionality outputs $(input, W_i, vid, ?)$ to the adversary. On a later input $(complete, vid)$ from the adversary, the functionality stores (vid, x) and outputs $(complete, vid)$ to all parties.

Secret Eval: On command

$$(eval, vid_1, \ldots, vid_p, f, vid_{p+1}, \ldots, vid_{p+q})$$

from all parties (if $vid_{p+1}, \ldots, vid_{p+q}$ are present in memory and $vid_1, \ldots, vid_p$ are not), the functionality outputs $(eval, vid_1, \ldots, vid_p, f, vid_{p+1}, \ldots, vid_{p+q})$ to the adversary. On a later input

$$(complete, vid_1, \ldots, vid_p, f, vid_{p+1}, \ldots, vid_{p+q})$$

from the adversary, the functionality retrieves $(vid_{p+1}, x_1), \ldots, (vid_{p+q}, x_q)$ computes

$$(z_1, \ldots, z_p) \leftarrow f(x_1, \ldots, x_q)$$

and stores $(vid_1, z_1), \ldots, (vid_p, z_p)$. Then it outputs $(complete, vid_1, \ldots, vid_p, f, vid_{p+1}, \ldots, vid_{p+q})$ to all parties.

Output: On input $(output, W_i, vid)$ from all parties (if vid is present in memory), the functionality outputs $(output, W_i, vid)$ to the adversary. On a later input $(complete, vid)$ from the adversary, the functionality retrieves (vid, x) and outputs it to W_i.

Further adversarial behavior: All the completion messages are delivered at the will of the adversary only, i.e., a message like $(complete, \ldots)$ is not delivered to an honest party until the adversary signals that it should be delivered, and the adversary might deliver to different parties at different times.

Figure 1. The ideal functionality for reactive MPC.

We use some short-hand notation: $[x]$ is a secret representation of x, i.e., a value uploaded to the ideal functionality using the **Input** command or computed via the **Secret Eval** command. The representation is assumed to be *cheap to compute on* using linear operations on elements from $\mathbb{F}$, so we will write $[ax + by] = a[x] + b[y]$ for publicly known $a, b \in \mathbb{F}$ and secrets $x, y \in \mathbb{F}$, and we will not count these operations towards the complexity of the protocol. All our notation generalizes to vectors in a straightforward way i.e., we will write $[a\boldsymbol{x} + b\boldsymbol{y}] = a[\boldsymbol{x}] + b[\boldsymbol{y}]$. We use this notation:

Input: $[x] \leftarrow \mathsf{Input}(W_i, x)$ allows party W_i to input the value $x \in \mathbb{F}$ to the computation; We also define a command $[r] \leftarrow \mathsf{Rand}()$ which can be simply implemented by having all W_i perform $[r_i] \leftarrow \mathsf{Input}(W_i, r_i)$; for random $r_i \in \mathbb{F}$ and then $[r] = \sum_i [r_i]$;

Eval: $([z_1], \ldots, [z_p]) \leftarrow f([x_1], \ldots, [x_q])$ allows to compute an agreed upon function f of q inputs and p outputs on secret representations, producing again secret representations. This is done via the **Secret Eval** command.

Linear: For public $a, b \in \mathbb{F}$ and secret $x, y \in \mathbb{F}$ the command $[z] \leftarrow a[x] + b[y]$ allows parties to compute a linear combination in $\mathbb{F}$. This is a special case of **Secret Eval**, but we single it out for notational

convenience and because the command is assumed to be essentially for free for our framework to make sense.

Multiplication: $[z] \leftarrow \mathsf{Mul}([x],[y])$ allows parties to compute a representation of $z = x \cdot y$. This is a special case of **Secret Eval**, but we single it out for notational convenience and because the command is assumed to be not too expensive for our framework to make sense.

Open: $x \leftarrow \mathsf{Open}([x])$ publicly reveals the value inside $[x]$; We also define $x \leftarrow \mathsf{OpenTo}(W_i,[x])$ which allows to reveal a value only to party W_i, and can always be implemented doing $[r] \leftarrow \mathsf{Input}(W_i, r)$ for uniform random r chosen by W_i and $c \leftarrow \mathsf{Open}([k+r])$, and then party W_i outputs $x = c - r$.

3.3 Protocol Analysis

The protocol is given in Figure 2 (see Section 2 for a high-level description of the protocol).

Theorem 1. *Let π be the protocol in Figure 2. We prove that π securely implements the ideal functionality $\mathcal{F}_{\text{FE}}^{f}$ against any static adversary corrupting any number of clients and at most $m-1$ workers.*

We do the proof in the UC framework [Can01]. We prove static security against an adversary corrupting any number of clients and up to all but one of the workers. Recall that we have the following proof burden.

There is a real world where we run the protocol. The parties of the protocol have access to secure point-to-point channels (which can in turn be implemented using cryptography) plus a copy of $\mathcal{F}_{\text{MPC}}$. In the real world there is an adversary A attacking the protocol. It is the adversary A which controls the corrupted parties, i.e., it sends messages on behalf of the corrupted parties and sees all messages sent to the corrupted parties, including the messages to and from $\mathcal{F}_{\text{MPC}}$. In addition the adversary has access to the adversarial behavior allowed by $\mathcal{F}_{\text{MPC}}$, like deciding when messages are delivered. There is also an environment Z. It is the environment which provides the inputs to the honest parties of the protocol and which sees the outputs of the honest parties of the protocol. The environment Z can also interact with A, in any way that they desire and at any time. The interaction is via exchanging messages. At the end of the interaction, the environment outputs a bit. We denote the distribution of this bit by $\text{EXEC}_{\pi,A,Z}(\kappa)$, where κ is the security parameter. Both A and Z are restricted to poly-time computations.

In the ideal process there are three entities, the ideal functionality $\mathcal{F}_{\text{FE}}^{f}$, the adversary S and the environment Z. It is the environment which provides inputs to $\mathcal{F}_{\text{FE}}^{f}$ via the (dummy) honest parties, and it sees their outputs from $\mathcal{F}_{\text{FE}}^{f}$. It is the adversary S which provides inputs to $\mathcal{F}_{\text{FE}}^{f}$ on behalf of the corrupted parties, and it sees the outputs to the corrupted parties from $\mathcal{F}_{\text{FE}}^{f}$. In addition it has access to the adversarial behavior allowed by $\mathcal{F}_{\text{FE}}^{f}$. Besides this, Z and S can interact by exchanging messages. At the end of the execution, Z will output a bit. We use $\text{EXEC}_{\mathcal{F}_{\text{FE}}^{f},S,Z}(\kappa)$ to denote the distribution of this bit. Both S and Z are restricted to poly-time computations.

To prove security of the protocol we have to construct for all adversaries A for the real world an adversary S for the ideal process such that no Z can guess whether it interacts with π and A or $\mathcal{F}_{\text{FE}}^{f}$ and S. We also call this adversary S a simulator. Technically we require that for all A there exists S such that for all Z the value $|\Pr[\text{EXEC}_{\pi,A,Z}(\kappa)=1] - \Pr[\text{EXEC}_{\mathcal{F}_{\text{FE}}^{f},S,Z}(\kappa)=1]|$ goes to 0 faster than any inverse polynomial in k.

We proceed to the proof. Assume we are given any A. The simulator S works as follows:

Simulated Protocol S runs internally a copy of π, i.e., a copy of each party $C_1,\ldots,C_n,W_1,\ldots,W_m$ along with a copy of $\mathcal{F}_{\text{MPC}}$. We call this the *simulated protocol*. To distinguish the simulated parties from the corresponding parties in the real execution we write $\tilde{C}_1,\ldots,\tilde{C}_n,\tilde{W}_1,\ldots,\tilde{W}_m$ and $\tilde{\mathcal{F}}_{\text{MPC}}$ for the parties and the ideal functionality and $\tilde{\pi}$ for the simulated protocol as a whole.

Simulated Adversary S also runs internally a copy of A, we call this the *simulated adversary* and denote it by $\tilde{A}$.

Monitor Corrupted Parties S lets the simulated adversary $\tilde{A}$ and the simulated parties interact exactly as in the real execution, i.e., whenever $\tilde{A}$ instructs a corrupted party to send a given message, the simulator performs this command in the simulated protocol, and whenever a corrupted simulated party

All algebraic notation denotes operations in $\mathbb{F}$.

Clients Input Phase: Each client C_j with input $\boldsymbol{x_j}$:

1. Pick random $\{\boldsymbol{x_{j,i}}\}_{i=1}^m$ from $\mathbb{F}^\lambda$ s.t., $\sum_{i=1}^m \boldsymbol{x_{j,i}} = \boldsymbol{x_j}$;
2. Pick random $\{k_{j,i}\}_{i=1}^m$ from $\mathbb{F}$; let $k_j = \sum_{i=1}^m k_{j,i}$;
3. Pick random $\{r_{j,i}\}_{i=1}^m$ from $\mathbb{F}$; let $r_j = \sum_{i=1}^m r_{j,i}$;
4. Compute
$$t_j = \mathsf{Tag}(k_j, (\boldsymbol{x_j}, r_j)) = k_j^{\lambda+3} + k_j^{\lambda+1} \cdot r_j + \sum_{h=1}^{\lambda} k^h \cdot x_{j,h} \ ;$$
5. Pick random $\{t_{j,i}\}_{i=1}^m$ from $\mathbb{F}$ s.t., $\sum_{i=1}^m t_{j,i} = t_j$;
6. Send the values $v_{j,i} = (x_{j,i}, t_{j,i}, k_{j,i}, r_{j,i})$ to W_i for $i \in \{1, \ldots, m\}$.

Workers Computation Phase: The workers $W_1, \ldots, W_m$ do:

1. Each worker W_i, for $i \in \{1, \ldots, m\}$, waits until receiving input (*eval*) and then proceeds as below:
2. Each worker W_i, for $i \in \{1, \ldots, m\}$ executes:
   (a) $[\boldsymbol{x_{j,i}}] \leftarrow \mathsf{Input}(W_i, \boldsymbol{x_{j,i}})$;
   (b) $[t_{j,i}] \leftarrow \mathsf{Input}(W_i, t_{j,i})$;
   (c) $[k_{j,i}] \leftarrow \mathsf{Input}(W_i, k_{j,i})$;
   (d) $[r_{j,i}] \leftarrow \mathsf{Input}(W_i, r_{j,i})$;
3. All workers jointly compute[a] for all $j \in \{1, \ldots, n\}$:
   (a) $[\boldsymbol{x_j}] = \sum_{i=1}^m [\boldsymbol{x_{j,i}}]$;
   (b) $[t_j] = \sum_{i=1}^m [t_{j,i}]$;
   (c) $[k_j] = \sum_{i=1}^m [k_{j,i}]$;
   (d) $[r_j] = \sum_{i=1}^m [r_{j,i}]$ (and let $[\boldsymbol{r}] = ([r_1], \ldots, [r_n])$);
   (e) $k_j \leftarrow \mathsf{Open}([k_j])$;
   (f) $[\alpha_j] = [t_j] - \mathsf{Tag}(k_j, ([\boldsymbol{x_j}], [r_j]))$;
4. $[s] \leftarrow \mathsf{Rand}()$;
5. $[\beta] = \mathsf{Mul}([s], \sum_{j=1}^n [\alpha_j])$;
6. $\beta \leftarrow \mathsf{Open}([\beta])$; If $\beta \neq 0$ output `abort` and halt, else continue;
7. Compute: $[\boldsymbol{z}] = f([\boldsymbol{x_1}], \ldots, [\boldsymbol{x_n}])$;
8. Compute: $[\boldsymbol{c}] = [\boldsymbol{z}] + [\boldsymbol{r}]$;
9. $\boldsymbol{c} \leftarrow \mathsf{Open}([\boldsymbol{c}])$;

Client Output Phase:

1. (Each worker W_i) Send c_j to C_j;
2. Let $c_{j,i}$ be the output that C_j receives from W_i;
3. If $\exists i_0, i_1$ such that $c_{j,i_0} \neq c_{j,i_1}$, then C_j outputs `abort` and halts, else let $c_j = c_{j,1}$;
4. C_j outputs $z_j = c_j - r_j$.

[a] Note that all the operations described below can be computed "for free" since they are linear in the secret representation $[\cdot]$. The only non linear operations – computing the powers of k inside Tag – is done in public.

Figure 2. The protocol.

receives a message in the execution of the simulated protocol, S gives this message to A. Notice that as a consequence of this simulation strategy, S knows all messages sent and received by corrupted parties, including the messages to and from $\tilde{\mathcal{F}}_{\mathrm{MPC}}$. The simulator also knows the internal state of $\tilde{\mathcal{F}}_{\mathrm{MPC}}$, as it is S which runs the copy $\tilde{\mathcal{F}}_{\mathrm{MPC}}$. We use these facts later.

Functionality $\mathcal{F}_{\text{FE}}^{f}$
The functionality is for the parties $C_1, \ldots, C_n, W_1, \ldots, W_m$.

Input: On input $(input, C_j, \boldsymbol{x_j})$ from C_j and $(input, C_j, ?)$ from all W_i, the functionality outputs $(input, C_j, ?)$ to the adversary. On a later input $(complete, C_j)$ from the adversary, the functionality stores $(input, C_j, \boldsymbol{x_j})$ and outputs $(complete, input, C_j)$ to all W_i.

Eval: On command $(eval)$ from all W_i at a point where $(input, C_j, \boldsymbol{x_j})$ is stored for all $j \in \{1, \ldots, n\}$, compute

$$\boldsymbol{z} = f(\boldsymbol{x_1}, \ldots, \boldsymbol{x_n})$$

and store $(output, C_j, z_j)$ for all $j \in \{1, \ldots, n\}$. Then output $(eval, \{z_j\}_{j \in J})$ to the adversary, where J is the set of indices of corrupted clients.

Output: On command $(deliver, C_j)$ from the adversary, at a point where $(output, C_j, z_j)$ is stored, output z_j to C_j and delete $(output, C_j, z_j)$.

Figure 3. The ideal functionality for (outsourced) function evaluation of f.

Relay Between Adversary and Environment S lets the simulated adversary A and Z interact exactly as in the real execution, i.e., whenever $\tilde{A}$ sends a message to its environment S passes it on to Z, and whenever Z sends a message to S, the simulator just passes it on to A.

Dummy Honest Inputs Whenever Z gives an input $\boldsymbol{x_j}$ to an honest C_j, the simulator S is given $(input, C_j, ?)$. It then picks a dummy input $\tilde{\boldsymbol{x}}_{\boldsymbol{j}}$ for $\tilde{C}_j$, e.g., $\tilde{\boldsymbol{x}}_{\boldsymbol{j}} = 0$ or some other legal input. Then it simply runs $\tilde{\pi}$ according to the protocol, but with this dummy input $\tilde{\boldsymbol{x}}_{\boldsymbol{j}}$ to $\tilde{C}_j$ instead of the correct input $\boldsymbol{x_j}$ (which is unknown to S by the rules of the game).

Eval Whenever Z gives an input $(eval)$ to an honest W_i, the simulator S is given $(eval, W_i)$. It then simply inputs $(eval)$ to $\tilde{W}_i$ in the simulated protocol and then runs $\tilde{W}_i$ according to the protocol.

Extracting Corrupted Inputs Recall that it is S which must provide inputs to $\mathcal{F}_{\text{FE}}^{f}$ on behalf of the corrupted parties. It must try to make these inputs consist with the "input" of the corrupted $\tilde{C}_j$. Note that $\tilde{C}_j$ has no explicit input, it is defined via its behavior in the protocol. The job of the simulator is hence to extract this implicit input. The simulator extracts the input $\boldsymbol{x_j}$ of a corrupted $\tilde{C}_j$ as follows: It waits until all workers have input the values $\tilde{\boldsymbol{x}}_{\boldsymbol{i,j}}, \tilde{t}_{i_j}, \tilde{k}_{i,j}, \tilde{r}_{i,j}$ to $\tilde{\mathcal{F}}_{\text{MPC}}$. Then it computes $\tilde{\boldsymbol{x}}_{\boldsymbol{j}} = \sum_{i=1}^{m} \tilde{\boldsymbol{x}}_{\boldsymbol{j,i}}$ and $\tilde{r}_j = \sum_{i=1}^{m} \tilde{r}_{j,i}$. Then it inputs $\boldsymbol{x_j} = \tilde{\boldsymbol{x}}_{\boldsymbol{j}}$ to $\mathcal{F}_{\text{FE}}^{f}$ on behalf of the corrupted C_j. It saves $\tilde{r}_j$ for later use.

Patching Corrupted Outputs Recall that A sees the outputs of corrupted parties in the simulated protocol and can talk to Z which sees the inputs and outputs of $\mathcal{F}_{\text{FE}}^{f}$. Hence it is important that the outputs in the simulated protocol are consistent with the inputs and outputs of $\mathcal{F}_{\text{FE}}^{f}$. This is not necessarily the case, as we ran the simulated protocol on dummy inputs for all honest parties. The simulator deals with this as follows. Assume that the simulated protocol reaches Step 9 in the Workers Computation Phase. Before executing this step, the simulator S will modify the value of c_j inside $\tilde{\mathcal{F}}_{\text{MPC}}$ for each corrupted $\tilde{C}_j$. Observe that if the simulated protocol reaches Step 9, then all parties in the simulated protocol must have given inputs, or the simulated protocol would not have passed Step 2d in the Workers Computation Phase. Hence S will have given an input $\boldsymbol{x_j}$ to $\mathcal{F}_{\text{FE}}^{f}$ on behalf of each corrupted C_j at this point. Furthermore, since it is S which gives (dummy) inputs to the honest $\tilde{C}_j$ in the simulated protocol and since S only does so when Z gives input to the corresponding C_j on $\mathcal{F}_{\text{FE}}^{f}$, we can conclude that when $\tilde{\pi}$ reaches Step 9, the environment gave input to $\mathcal{F}_{\text{FE}}^{f}$ on behalf of all honest C_j. So, all in all, when the simulated protocol reaches Step 9, all C_j received an input $\boldsymbol{x_j}$ in $\mathcal{F}_{\text{FE}}^{f}$. Using a similar line of reasoning we see that if $\tilde{\pi}$ reaches Step 9, then Z must have input $(eval)$ to all honest W_i on $\mathcal{F}_{\text{FE}}^{f}$. So, now S can input $(eval)$ to all corrupted W_i on $\mathcal{F}_{\text{FE}}^{f}$. In response $\mathcal{F}_{\text{FE}}^{f}$ computes $\boldsymbol{z} = f(\boldsymbol{x_1}, \ldots, \boldsymbol{x_n})$ and outputs $(eval, \{z_j\}_{j \in J})$ to S, where J is the set of corrupted parties. Now S computes $\tilde{c}_j = z_j + \tilde{r}_j$, where $\tilde{r}_j$ was computed and

stored in Extracting Corrupted Inputs. Then S changes the internal state of $\tilde{\mathcal{F}}_{\text{MPC}}$ to hold the value $c_j = \tilde{c}_j$. Then it simply runs the simulated protocol according to the protocol.

Honest Output Delivery Whenever an honest client $\tilde{C}_j$ reaches Step 4 in Client Output Phase, the simulator inputs $(\textit{deliver}, C_j)$ to $\mathcal{F}_{\text{MPC}}$, which makes $\mathcal{F}_{\text{MPC}}$ output z_j to Z on behalf of C_j.

That completes the description of the simulator. We now prove that

$$|\Pr[\text{EXEC}_{\pi,A,Z}(\kappa) = 1] - \Pr[\text{EXEC}_{\mathcal{F}_{\text{FE}}^f,S,Z}(\kappa) = 1]| \leq (\lambda + 2)2^{-\kappa+1} \ .$$

Let E be the event that some client C_j has its input replaced, i.e., C_j runs with input $\boldsymbol{x_j}$, but after Step 2d in Workers Computation phase it holds for the values $\boldsymbol{x_{i,j}}$ in $\mathcal{F}_{\text{MPC}}$ that $\boldsymbol{x_j} \neq \sum_{i=1}^{m} \boldsymbol{x_{i,j}}$ (or similarly, $r_j \neq \sum r_{i,j}$). We can also define E in the simulation, but via the (dummy) input $\tilde{\boldsymbol{x}}_{\boldsymbol{j}}$ and the values $\tilde{\boldsymbol{x}}_{\boldsymbol{i,j}}$ in $\tilde{\mathcal{F}}_{\text{MPC}}$ and say that E occurs when $\tilde{\boldsymbol{x}}_{\boldsymbol{j}} \neq \sum_{i=1}^{m} \tilde{\boldsymbol{x}}_{\boldsymbol{i,j}}$. Let $\bar{E}$ denote the event that E did not occur. We clearly have that

$$\begin{aligned}\Pr[\text{EXEC}_{\pi,A,Z}(\kappa) = 1] = \\ &\Pr[E]\Pr[\text{EXEC}_{\pi,A,Z}(\kappa) = 1|E]+ \\ &(1 - \Pr[E])\Pr[\text{EXEC}_{\pi,A,Z}(\kappa) = 1|\bar{E}]\end{aligned}$$

and

$$\begin{aligned}\Pr[\text{EXEC}_{\mathcal{F}_{\text{FE}}^f,S,Z}(\kappa) = 1] = \\ &\Pr[E]\Pr[\text{EXEC}_{\mathcal{F}_{\text{FE}}^f,S,Z}(\kappa) = 1|E]+ \\ &(1 - \Pr[E])\Pr[\text{EXEC}_{\mathcal{F}_{\text{FE}}^f,S,Z}(\kappa) = 1|\bar{E}] \ .\end{aligned}$$

We will first show *Claim 1*: the probability $\Pr[E]$ is the same in the real execution and in the ideal process. We will also prove *Claim 2*: $\Pr[E] \leq (\lambda + 2)2^{-k+1}$. Then we show *Claim 3*: It holds that

$$\Pr[\text{EXEC}_{\mathcal{F}_{\text{FE}}^f,S,Z}(\kappa) = 1|\bar{E}] = \Pr[\text{EXEC}_{\pi,A,Z}(\kappa) = 1|\bar{E}] \ .$$

From these three claims it follows that

$$\begin{aligned}&|\Pr[\text{EXEC}_{\pi,A,Z}(\kappa) = 1] - \Pr[\text{EXEC}_{\mathcal{F}_{\text{FE}}^f,S,Z}(\kappa) = 1]| \\ &\quad\leq |\Pr[E]\Pr[\text{EXEC}_{\pi,A,Z}(\kappa) = 1|E]- \\ &\qquad\Pr[E]\Pr[\text{EXEC}_{\mathcal{F}_{\text{FE}}^f,S,Z}(\kappa) = 1|E]| \\ &\quad\leq \Pr[E] \leq (\lambda + 2)2^{-\kappa+1}\end{aligned}$$

as desired.

Proof of Claims 1 and 2 We prove that if the adversary uses values $\boldsymbol{x_{i,j}}$ such that $\boldsymbol{x_j} \neq \sum_{i=1}^{m} \boldsymbol{x_{i,j}}$, then $\beta \neq 0$ with probability $\leq (\lambda + 2)2^{-\kappa+1}$, and the probability is independent of $\boldsymbol{x_j}$. The same proof applies to the simulation, as the simulation is just a run of the real protocol but on different inputs. From this Claims 1 and 2 then follow.

Let $\alpha = \sum_j \alpha_j$. There are two (non-disjoint) ways it can happen that $\beta = 0$, namely $\alpha = 0$ and $s = 0$. Since $\Pr[s = 0] \leq 2^{-\kappa}$, independently of $\boldsymbol{x_j}$, it is sufficient to prove that

$$\Pr[\alpha = 0] \leq (\lambda + 2)2^{-\kappa}$$

and that the probability is independent of $\boldsymbol{x_j}$.

If all clients are corrupted, there is nothing to prove. So, since corrupting more parties gives the adversary strictly more powers and since the role of all clients is symmetric, let us assume without loss of generality

that all clients except C_1 are corrupted, which means that all values α_j with $j \neq 1$ are under full control of the adversary. Thus we define $\phi = -\sum_{j=2}^{m} \alpha_j$ and we clearly see that

$$\Pr[\alpha_1 = \phi] = \Pr[\alpha = 0] \ .$$

Our model assumes that at least one worker is honest. Since the role of all workers is symmetric, let us assume without loss of generality that all workers except W_1 are corrupted. Since Z and A can communicate, we cannot assume that A does not know $\boldsymbol{x_1}$, and since getting $\boldsymbol{x_1}$ clearly cannot make A worse off, let us assume without loss of generality that A knows $\boldsymbol{x_1}$. To be able to distinguish between the correct values of $\boldsymbol{x_{i,1}}$ and the wrong ones chosen by corrupted workers, use $\boldsymbol{x_{i,1}}$ to denote the values chosen by C_1 and use $\bar{\boldsymbol{x}}_{\boldsymbol{i,1}}$ to denote the values in $\mathcal{F}_{\text{MPC}}$. Note that $\bar{\boldsymbol{x}}_{\boldsymbol{1,1}} = \boldsymbol{x_{1,1}}$ since W_1 is honest. We use similar notation for the values $k_{i,j}$, $r_{i,j}$ and $t_{i,j}$.

Notice that for $i > 1$ the adversary knows both $\boldsymbol{x_{i,1}}$ and $\bar{\boldsymbol{x}}_{\boldsymbol{i,1}}$, as it received $\boldsymbol{x_{i,1}}$ and it chose $\bar{\boldsymbol{x}}_{\boldsymbol{i,1}}$. Hence it also knows the relative error $\boldsymbol{X_{i,1}} = \bar{\boldsymbol{x}}_{\boldsymbol{i,1}} - \boldsymbol{x_{i,1}}$. Notice that $\bar{\boldsymbol{x}}_{\boldsymbol{i,1}} = \boldsymbol{x_{i,1}} + \boldsymbol{X_{i,1}}$. I.e., the adversary inputs the correct input plus some known error. Similarly we can write $\bar{k}_{i,1} = k_{i,1} + K_{i,1}$, $\bar{r}_{i,1} = r_{i,1} + R_{i,1}$ and $\bar{t}_{i,1} = t_{i,1} + T_{i,1}$ for values $K_{i,1}, R_{i,1}$ and $T_{i,1}$ known by the adversary. We use $\boldsymbol{X_1} = \sum_{i=2}^{m} \boldsymbol{X_{i,1}}$ to denote the sum of relative errors. Note that $\boldsymbol{X_1}$ is known by the adversary. Similarly, let $T_1 = \sum_{i=2}^{m} T_{i,1}$, $R_1 = \sum_{i=2}^{m} R_{i,1}$ and let $K_1 = \sum_{i=2}^{m} K_{i,1}$. Note that $\tilde{\boldsymbol{x}}_{\boldsymbol{1}} = \boldsymbol{x_1} + \boldsymbol{X_1}$, $\tilde{t}_1 = t_1 + T_1$, $\tilde{r}_1 = r_1 + R_1$ and $\tilde{k}_1 = k_1 + K_1$. We have that

$$\mathsf{Tag}(k_1, (\boldsymbol{x_1}, r_1)) - t_1 = 0$$

by design. We define a polynomial

$$p(k_{1,1}) = \alpha_1 = \mathsf{Tag}(\tilde{k}_1, (\tilde{\boldsymbol{x}}_{\boldsymbol{1}}, \tilde{r}_1)) - \tilde{t}_1$$

and note that we can rewrite this as

$$p(k_{1,1}) = \mathsf{Tag}(k_1 + K_1, (\boldsymbol{x_1} + \boldsymbol{X_1}, r_1 + R_1)) - \mathsf{Tag}(k_1, (\boldsymbol{x_1}, r_1)) - T_1 \ .$$

If $(\boldsymbol{X_1}, R_1, K_1, T_1) = (\boldsymbol{0}, 0, 0, 0)$ there is nothing to prove (i.e., the adversary is behaving honestly), so assume that this is not the case. Recall that

$$\mathsf{Tag}(k_1, (\boldsymbol{x_1}, r_1)) = (k_1)^{\lambda+3} + (k_1)^{\lambda+1} \cdot (r_1) + \sum_{h=1}^{\lambda} (k_1)^h \cdot x_{1,h}$$

and that $k_1 = \sum_i k_{1,i}$. Thus we note that if $K_1 \neq 0$ then $p(k_{1,1})$ is a polynomial of degree exactly $\lambda + 2$ (independent of anything else) and therefore $\forall \phi \in \mathbb{F}$

$$\Pr_{k_{1,1} \leftarrow \mathbb{F}}[p(k_{1,1}) = \phi | K_1 \neq 0] \leq (\lambda + 2) \cdot |\mathbb{F}|^{-1} \ .$$

If $K_1 = 0$ then we have that $p(k_{1,1})$ is a non-zero polynomial of degree at most $\lambda + 1$. In particular, since the tag is linear in the message we have that

$$p(k_{1,1}) = \mathsf{Tag}(k_1, (\boldsymbol{X_1}, R_1) - T_1 \ .$$

This implies that

$$\Pr_{k_{1,1} \leftarrow \mathbb{F}}[p(k_{1,1}) = \phi | K_1 = 0] \leq (\lambda + 1) \cdot |\mathbb{F}|^{-1} \ ,$$

which implies the claim.

Proof of Claim 3 First consider the following mind game. Consider an execution $\text{EXEC}_{\mathcal{F}_{\text{FE}}^f, S^1, Z}(\kappa)$, which runs exactly as the execution $\text{EXEC}_{\mathcal{F}_{\text{FE}}^f, S, Z}(\kappa)$, except that each time where S is about to use a dummy input $\tilde{\boldsymbol{x}}_{\boldsymbol{i}} = 0$ on behalf of honest C_i, it instead cheats and inspects $\mathcal{F}_{\text{FE}}^f$ to get the real value $\boldsymbol{x_i}$ and then it uses $\tilde{\boldsymbol{x}}_{\boldsymbol{i}} = \boldsymbol{x_i}$. Besides this cheat, everything runs as in the simulation. Note that in a simulation S is of course not allowed to perform the above cheat. However, we are here only defining a random variable $\text{EXEC}_{\mathcal{F}_{\text{FE}}^f, S^1, Z}(\kappa)$ for sake of the proof, and we are of course free to define it as we want. We claim that

$$|\Pr[\text{EXEC}_{\mathcal{F}_{\text{FE}}^f, S, Z}(\kappa) = 1|\bar{E}] - \Pr[\text{EXEC}_{\mathcal{F}_{\text{FE}}^f, S^1, Z}(\kappa) = 1|\bar{E}]| = 0 \ .$$

The reason is that the views of A and Z do not depend on the dummy inputs at all. To see this notice that $\tilde{\boldsymbol{x}}_{\boldsymbol{i}}$ is input to $\tilde{\mathcal{F}}_{\text{MPC}}$ and is not used anywhere else. And, the only values leaked by $\tilde{\mathcal{F}}_{\text{MPC}}$ which might depend on $\tilde{\boldsymbol{x}}_{\boldsymbol{i}}$ are the values c_j. As for the value c_j for all corrupted C_j, note that it is patched to $\tilde{c}_j$ before it is output from $\tilde{\mathcal{F}}_{\text{MPC}}$, hence it has the same distribution no matter whether S uses $\tilde{\boldsymbol{x}}_{\boldsymbol{j}} = 0$ or $\tilde{\boldsymbol{x}}_{\boldsymbol{j}} = \boldsymbol{x_j}$. As for the value c_j for all honest C_j, note that $c_j = \tilde{z}_j + r_j$ for a uniformly random value r_j. Hence, even though $\tilde{z}_j$ might depend on whether S uses $\tilde{x}_j = 0$ or $\tilde{x}_j = x_j$, the value c_j does not, as it is one-time pad encrypted with r_j which is known only to the honest C_j.

Consider then the mind game $\text{EXEC}_{\mathcal{F}_{\text{FE}}^f, S^2, Z}(\kappa)$, which runs exactly as the previous mind game $\text{EXEC}_{\mathcal{F}_{\text{FE}}^f, S^1, Z}(\kappa)$, except that we change the step Patching Corrupted Outputs such that S does not perform the patching $c_j = \tilde{c}_j$. Instead it just runs the simulated protocol with the value c_j already inside $\tilde{\mathcal{F}}_{\text{MPC}}$. We claim that

$$\Pr[\text{EXEC}_{\mathcal{F}_{\text{FE}}^f, S^1, Z}(\kappa) = 1|\bar{E}] - \Pr[\text{EXEC}_{\mathcal{F}_{\text{FE}}^f, S^2, Z}(\kappa) = 1|\bar{E}] = 0 \ .$$

Notice that in $\text{EXEC}_{\mathcal{F}_{\text{FE}}^f, S^2, Z}(\kappa)$ each honest $\tilde{C}_i$ is run with input $\tilde{\boldsymbol{x}}_{\boldsymbol{i}} = \boldsymbol{x_i}$, where $\boldsymbol{x_i}$ is the input to C_i on $\mathcal{F}_{\text{FE}}^f$. Then for each corrupted $\tilde{C}_i$ define $\tilde{\boldsymbol{x}}_{\boldsymbol{i}}$ as in Extracting Corrupted Inputs. Then it follows from the fact that E does not happen that the values $\boldsymbol{z}$ computed by $\tilde{\mathcal{F}}_{\text{MPC}}$ are equal to $f(\tilde{\boldsymbol{x}}_{\boldsymbol{1}}, \ldots, \tilde{\boldsymbol{x}}_{\boldsymbol{n}})$, as no honest party has its input replaced. Since the input to $\mathcal{F}_{\text{FE}}^f$ is $\boldsymbol{x_i} = \tilde{\boldsymbol{x}}_{\boldsymbol{i}}$ for the honest party (qua the cheat) and $\boldsymbol{x_i} = \tilde{\boldsymbol{x}}_{\boldsymbol{i}}$ for the corrupted parties (by design of Extracting Corrupted Inputs), it follows that the values $(z_1, \ldots, z_n)$ computed by $\tilde{\mathcal{F}}_{\text{MPC}}$ are equal to the values $(z_1, \ldots, z_n)$ computed by $\mathcal{F}_{\text{FE}}^f$. Furthermore, when the values $(z_1, \ldots, z_n)$ computed by $\tilde{\mathcal{F}}_{\text{MPC}}$ are equal to the values $(z_1, \ldots, z_n)$ computed by $\mathcal{F}_{\text{FE}}^f$, the patching $c_j = \tilde{c}_j$ clearly has no effect as $c_j = z_j + \tilde{r}_j$ and $\tilde{c}_j = z_j + \tilde{r}_j$. Ergo, it does not matter whether we do the patching or not.

We finally claim that the following holds

$$|\Pr[\text{EXEC}_{\mathcal{F}_{\text{FE}}^f, S^2, Z}(\kappa) = 1|\bar{E}] - \Pr[\text{EXEC}_{\pi, A, Z}(\kappa) = 1|\bar{E}]| = 0 \ .$$

Notice that after replacing dummy inputs by true inputs and dropping the patching of corrupted outputs, the "simulated" protocol run by S^2 is actually just a correct run of π on exactly the same inputs $\boldsymbol{x_i}$ given to $\mathcal{F}_{\text{FE}}^f$ by Z. I.e., S is internally running π *exactly* as it is being run in $\text{EXEC}_{\pi, A, Z}(\kappa)$. There might, however, still be a difference in the view of Z: In $\text{EXEC}_{\mathcal{F}_{\text{FE}}^f, S^2, Z}(\kappa)$ the output of an honest client C_i to Z is the value z_i output by $\mathcal{F}_{\text{FE}}^f$. In $\text{EXEC}_{\pi, A, Z}(\kappa)$ the output of an honest client C_i to Z is the value z_i output by C_i. We hence need to argue that in $\text{EXEC}_{\mathcal{F}_{\text{FE}}^f, S^2, Z}(\kappa)$ the output of an honest client $\tilde{C}_i$ is identical to the value z_i output by $\mathcal{F}_{\text{FE}}^f$. To see this, recall that we argued above that the values $(z_1, \ldots, z_n)$ computed by $\tilde{\mathcal{F}}_{\text{MPC}}$ are equal to the values $(z_1, \ldots, z_n)$ computed by $\mathcal{F}_{\text{FE}}^f$. Hence, all we have to argue is that the output of an honest client $\tilde{C}_i$ is identical to the value z_i computed by $\mathcal{F}_{\text{MPC}}$. This is so, as $c_i = z_i + r_i$ and the correct value of c_i is sent to C_i by at least one honest worker, so C_i only accepts c_i, and then it outputs $c_i - r_i = z_i$.

4 Extensions

4.1 Coping with Malicious Clients

In the previous protocol a malicious client can force the entire protocol to abort by using an invalid input (e.g., providing a MAC that is not consistent). As a result it will happen that $\beta \neq 0$ and the computation

will abort. In an application where there are thousands or millions of clients it is undesirable that a single malicious (or faulty) client can prevent the whole computation from terminating. We therefore want a protocol which guarantees termination whenever all workers are honest. The problem is that when $\beta \neq 0$ we cannot see if it happened because of a malicious worker or a malicious client. We therefore need to add extra mechanisms for detecting who was cheating and for recovering when it was a client.

The high-level idea of our solution is to identify clients which provided invalid inputs and replace their inputs with a default value. This means that all attacks possible by a client are equivalent to choosing an alternative input, which is of course an allowed option for even an honest party. The only difference is that in case of cheating, all workers will learn that the client cheated, and all workers will learn the (alternative) input of the client. This should deter clients from cheating at all. "Cheating" might, however, also happen because a client is faulty, so we should expect a few cheating parties occasionally. Since we want to consider cases where n is huge, we will therefore present a solution optimized for the case where there are very few cheating parties relative to the size of n.

The first step is to identify the troublesome clients. Note that $\sum_{j=1}^{n} \alpha_j \neq 0$ just shows that some $\alpha_j \neq 0$. We can find it by securely computing $\sum_{j=1}^{n/2} \alpha_j$ and $\sum_{j=n/2+1}^{n} \alpha_j$ and revealing blinded values to check which of them is non-zero (possibly both). Then continue like this recursively, until all j for which $\alpha_j \neq 0$ are known. At the point where $O(n)$ sums have been computed, blinded and opened, switch to a mode where we simply open blinded versions of all α_j. This way we never use more than $O(n)$ multiplications, and if there is a constant number of non-zero α_j, then we use only $O(\log n)$ multiplications. Let I be the set of indices j such that $\alpha_j \neq 0$. The above strategy finds I except with negligible probability. Now we must for each $j \in I$ find out whether C_j is corrupted or whether some worker input a different value to the computation than the one provided by C_j. Assume for now that 1) a worker can reveal which values it received from C_j without the worker being able to reveal a value different from the ones actually received from C_j and 2) it is OK to reveal the input $\boldsymbol{x_j}$ of C_j. In that case the solution is trivial: for each $j \in I$ each worker W_i will reveal the values $(\boldsymbol{x_j}, r_{j,i}, t_{j,i}, k_{j,i})$ received from C_j and the workers will open the values $[\boldsymbol{x_j}]$,$[r_j]$, $[t_j]$ and $[k_j]$ to check that W_i uploaded the right values to $\mathcal{F}_{\text{MPC}}$. If not, W_i is corrupted, and the protocol is terminated. Otherwise, it must be the case that $t_j \neq \mathsf{Tag}(k_j, (\boldsymbol{x_j}, r_j))$, as $\alpha_j \neq 0$. So, since all the uploaded values were the ones received from the client, it follows that C_j is corrupted. In that case the protocol continues, using, e.g., $x_j = 0$ as input on behalf of C_j. We now discuss how to get rid of the two assumptions.

As for the second assumption, we will simply let each C_j split the input $\boldsymbol{x_j}$ into two random shares $\boldsymbol{y_j^1}$ and $\boldsymbol{y_j^2}$ for which $\boldsymbol{x_j} = \boldsymbol{y_j^1} + \boldsymbol{y_j^2}$. E.g., pick $\boldsymbol{y_j^1}$ uniformly at random and let $\boldsymbol{y_j^2} = \boldsymbol{x_j} - \boldsymbol{y_1^j}$. Then proceed as before, but let C_j give both of the inputs $\boldsymbol{y_j^1}$ and $\boldsymbol{y_j^1}$ as above, i.e., with separate keys and MACs. Now, if any one of them turns out to be troublesome, open it and find out who was cheating. If it was C_j, use a default value. If it was W_i, terminate. Then compute $\boldsymbol{z} = f(\boldsymbol{y_1^1} + \boldsymbol{y_1^2}, \ldots, \boldsymbol{y_n^1} + \boldsymbol{y_n^2})$. Note that there is never a need to reveal both $\boldsymbol{y_j^1}$ and $\boldsymbol{y_j^2}$: if they are both troublesome, simple reveal $\boldsymbol{y_j^1}$ and use that value to make the decision. Furthermore, revealing a single of the values $\boldsymbol{y_j^1}$ or $\boldsymbol{y_j^2}$ is secure, as they are individually uniformly random and independent of $\boldsymbol{x_j}$.

As for the first assumption, notice that it is not enough to ask C_j to sign the values it sends to W_i: The client might simply not send a signature, and when W_i complains that C_j did not send a signature, it might be W_i that is lying. In fact, any solution where the client sends something over a secure channel will fall prey to this problem: The client might refuse to send the value, but it might also be the worker lying about not having received that value. We therefore need a solution where clients only send public values. Furthermore, since all but one worker might be corrupted, any public value not sent to all workers, might still fall prey to the above attack: the $m - 1$ workers seeing the value might refuse to have received it. Hence we might essentially restrict ourselves to solutions where the client sends one public value and sends it to all workers.

We describe one such solution. Assume that each W_i has a public encryption key e_i for a public-key encryption scheme and that only W_i knows the decryption key d_i. We need that this encryption scheme is secure against chosen-ciphertext attack (IND-CCA2) and that decryption yields the message plus the randomness used to encrypt, and that IND-CCA2 security holds even if the decryption oracle returns this randomness. In the random oracle model, RSA-OAEP is such an encryption scheme, assuming that the RSA

Algorithm FIND: Input: $([a_1], [a_2], \ldots, [a_q])$, a list of q secret shared values.

1. $[s] \leftarrow \mathsf{Rand}()$;
2. $[\beta] = \mathsf{Mul}([s], \sum_{j \in 1,\ldots,q}[a_q])$;
3. $\beta \leftarrow \mathsf{Open}([\beta])$;
4. If $\beta = 0$ return $\{\}$;
5. Else, if $q = 1$ return $\{[a_1]\}$;
6. Else, let $r = \lfloor q/2 \rfloor$ and return $\mathsf{FIND}([a_1], \ldots, [a_r]) \cup \mathsf{FIND}([a_{r+1}], \ldots, [a_q])$.

Figure 4. Algorithm to identify potentially inconsistent client inputs.

function is one-way. Our solution then proceeds as follows: Use $v_{j,i}$ to denote the value that a client C_j should send secretly to W_i. The client will compute an encryption $\gamma_{j,i} = E_{e_i}((i, j, v_{j,i}); s_{j,i})$, where $s_{j,i}$ is the randomness used by the encryption algorithm. Then C broadcasts $(\gamma_{j,1}, \ldots, \gamma_{j,n})$ to all workers. Then W_i computes $(i', j', v_{j,i}, s_{j,i}) = D_{d_i}(\gamma_{j,i})$. If $i' \neq i$ or $j' \neq j$, then W_i broadcasts $(i', j', v_{j,i}, s_{j,i})$ and all workers check that $\gamma_{j,i} = E_{e_i}((i', j', v_{j,i}); s_{j,i})$ and that $i' \neq i$ or $j' \neq j$. If this is the case, use a default input for C_j. If it is not the case, then W_i is cheating. In that case, terminate the protocol. If W_i is later asked to reveal $v_{j,i}$, W_i broadcasts $(v_{j,i}, s_{j,i})$ and all workers check that $\gamma_{j,i} = E_{e_i}((i, j, v_{j,i}); s_{j,i})$.

It is also possible to get a solution not using the random oracle model. Each key e_i is the parameters for an identity-based encryption scheme and d_i is the master secret key. The client will compute an encryption $\gamma_{j,i} \leftarrow E_{e_i,(i,j,sid)}(v_{j,i})$, i.e., encrypt $v_{j,i}$ under the identity (i, j, sid), where sid is a session identifier which is fresh for each run of the protocol. If W_i is later asked to reveal $v_{j,i}$, W_i generates and broadcasts the secret key $d_{i,j,sid}$ for identity (i, j, sid). Then all workers compute $v_{j,i} = D_{d_{i,j,sid}}(\gamma_{j,i})$.

We are then left with the problem of how the client broadcasts to the workers. Note that we can use a standard authenticated broadcast protocol like Dolev-Strong broadcast [DS83], as this protocol requires the sender to send just a single message to each of the other participant, here the workers.

Note that the above solution requires that the client be able to sign messages. We would ideally like a solution which works under the sole assumption that the clients have an authenticated channel to each of the workers, as this is strictly more general and better models practice, where clients might authenticate themselves towards the computation providers using a simple password mechanism on top of a server authenticated secure transport layer. However, such a solution is not possible. It is well-known that broadcast among m parties without the use of signatures, requires that $> m/2$ of the parties are honest, and we want to tolerate that all but one worker is corrupted. We must therefore settle for a solution where clients need to have public keys for a signature scheme.

We finally note that the security property that we are trying to achieve in this section i.e., how to make sure that a single faulty client cannot make the computation stall, cannot be captured in the UC framework: since the adversary fully controls the network and the delivery of messages, in the UC framework the adversary can make the computation stall even if *no parties* are corrupted.

Figure 5 shows the resulting protocol: the protocol assumes that a public encryption scheme (E, D) has been set up such that each worker W_i holds a private decryption key d_i while all other parties, workers as well as clients, hold the corresponding public encryption key e_i. Furthermore, the protocol assumes a broadcast primitive; we use $\mathsf{BROADCAST}(id, msg, \{R_i\})$ to denote that a party P broadcasts msg to a set of receivers $\{R_i\}$ while $msg \leftarrow \mathsf{RECEIVE}(id, P)$ is used by a party to receive msg broadcast by P. As noted, broadcast can be realized using Dolev-Strong [DS83]. A recursive algorithm FIND, listed in Figure 4, is used for identifying potentially inconsistent client inputs. FIND may result in $O(n \log n)$ multiplications. In order to have $O(n)$ multiplications, let e be any constant, e.g. $e = 2$, and modify FIND such that a global counter δ is increased for every multiplication (i.e., $\delta := \delta + 1$ each time Step 2 of FIND is executed). If at some point

All algebraic notation denotes operations in $\mathbb{F}$.

Clients Input Phase: Each client C_j with input $\boldsymbol{x_j}$:

1. Pick random $\boldsymbol{y_j^1}$. Compute $\boldsymbol{y_j^2} = \boldsymbol{x_j} - \boldsymbol{y_j^1}$;
2. For $w \in \{1, 2\}$ do the following:
   (a) Pick random $\{\boldsymbol{y_{j,i}^w}\}_{i=1}^m$ from $\mathbb{F}^\lambda$ s.t., $\sum_{i=1}^m \boldsymbol{y_{j,i}^w} = \boldsymbol{y_j^w}$;
   (b) Pick random $\{k_{j,i}^w\}_{i=1}^m$ from $\mathbb{F}$; let $k_j^w = \sum_{i=1}^m k_{j,i}^w$;
   (c) Pick random $\{r_{j,i}^w\}_{i=1}^m$ from $\mathbb{F}$; let $r_j^w = \sum_{i=1}^m r_{j,i}^w$;
   (d) Compute
   $$t_j^w = \mathsf{Tag}(k_j^w, (\boldsymbol{y_j^w}, r_j^w)) = (k_j^w)^{\lambda+3} + (k_j^w)^{\lambda+1} \cdot (r_j^w) + \sum_{h=1}^{\lambda} (k_j^w)^h \cdot (y_{j,h}^w) \ ;$$
   (e) Pick random $\{t_{j,i}^w\}_{i=1}^m$ from $\mathbb{F}$ s.t., $\sum_{i=1}^m t_{j,i}^w = t_j^w$;
   (f) Define the values $v_{j,i}^w := (\boldsymbol{y_{j,i}^w}, t_{j,i}^w, k_{j,i}^w, r_{j,i}^w)$ for $i \in \{1, \ldots, m\}$.
   (g) Compute $\gamma_{j,i}^w := E_{e_i}((i, j, v_{j,i}^w); s_{j,i}^w)$ for $i \in \{1, \ldots, m\}$;
   (h) Broadcast $\gamma_{j,i}^w$ to the workers, i.e., invoke $\mathsf{BROADCAST}(\texttt{input}, (\gamma_{j,i}^w)_{i=1}^m, \{W_i\}_{i=1}^m)$.

Workers Input Phase: Each worker W_i does:

1. Upon $(\gamma_{j,1}^w, \gamma_{j,2}^w, \ldots, \gamma_{j,m}^w) \leftarrow \mathsf{RECEIVE}(\texttt{input}, C_j)$ do:
   (a) Compute $(i', j', v_{j,i}^w, s_{j,i}^w) = D_{d_i}(\gamma_{j,i}^w)$;
   (b) If $i' \neq i$ or $j' \neq j$, $\mathsf{BROADCAST}(\texttt{bad-input}, (i', j', v_{j,i}^w, s_{j,i}^w), \{W_l\}_{l=1}^m)$.
2. Upon $(l', j', v_{j,l}^w, s_{j,l}^w) \leftarrow \mathsf{RECEIVE}(\texttt{bad-input}, W_l)$, if $\gamma_{j,l}^w = E_{e_l}((l', j', v_{j,l}^w); s_{j,l}^w)$ and if it is also the case that $l' \neq l$ or $j' \neq j$, set $[\boldsymbol{x_j}] \leftarrow [0]$. Otherwise, abort the protocol.

Workers Computation Phase: Upon (*eval*) each worker W_i does:

1. For $w \in \{1, 2\}$ do:
   (a) For all $j \in \{1, \ldots, n\}$ do:
      i. $[\boldsymbol{y_{j,i}^w}] \leftarrow \mathsf{Input}(W_i, \boldsymbol{y_{j,i}^w})$; $[\boldsymbol{y_j^w}] = \sum_{i=1}^m [\boldsymbol{y_{j,i}^w}]$;
      ii. $[t_{j,i}^w] \leftarrow \mathsf{Input}(W_i, t_{j,i}^w)$; $[t_j^w] = \sum_{i=1}^m [t_{j,i}^w]$;
      iii. $[k_{j,i}^w] \leftarrow \mathsf{Input}(W_i, k_{j,i}^w)$; $[k_j^w] = \sum_{i=1}^m [k_{j,i}^w]$;
      iv. $[r_{j,i}^w] \leftarrow \mathsf{Input}(W_i, r_{j,i}^w)$; $[r_j^w] = \sum_{i=1}^m [r_{j,i}^w]$ (and let $[\boldsymbol{r^w}] = ([r_1^w], \ldots, [r_n^w])$);
      v. $k_j^w \leftarrow \mathsf{Open}([k_j^w])$;
      vi. $[\alpha_j^w] = [t_j^w] - \mathsf{Tag}(k_j^w, ([\boldsymbol{y_j^w}], [r_j^w]))$;
   (b) Compute $I^w := \mathsf{FIND}([\alpha_1^w], [\alpha_2^w], \ldots, [\alpha_n^w])$.
2. Let $I := I^1 \cup I^2$. For $j \notin I$, set $[\boldsymbol{x_j}] := [\boldsymbol{y_j^1}] + [\boldsymbol{y_j^2}]$. For $j \in I$, do as follows:
   (a) If $j \in I^1$, let $w := 1$, else $w := 2$. W_i now reveals $v_{j,i}^w := (\boldsymbol{y_{j,i}^w}, t_{j,i}^w, k_{j,i}^w, r_{j,i}^w)$ as follows:
   (b) W_i invokes $\mathsf{BROADCAST}(\texttt{reveal}, (v_{j,i}^w, s_{j,i}^w), \{W_l\}_{l=1}^m)$;
   (c) Upon $(v_{j,i}^w, s_{j,i}^w) \leftarrow \mathsf{RECEIVE}(\texttt{reveal}, W_i)$, a worker W_l aborts if $\gamma_{j,i}^w \neq E_{e_i}((i, j, v_{j,i}^w); s_{j,i}^w)$;
   (d) Upon receiving all $v_{j,l}^w := (\boldsymbol{y_{j,l}^w}, t_{j,l}^w, k_{j,l}^w)$ for $l \in \{1, \ldots, m\}$, compute $\boldsymbol{y_j^w} := \sum_l \boldsymbol{y_{j,l}^w}$, $t_j^w := \sum_l t_{j,l}^w$, $k_j^w := \sum_l k_{j,l}^w$; $r_j^w := \sum_l r_{j,l}^w$;
   (e) W_i then computes $\boldsymbol{y_j'^w} \leftarrow \mathsf{Open}(\boldsymbol{y_j'^w})$; $t_j'^w \leftarrow \mathsf{Open}([t_j^w])$; $k_j'^w \leftarrow \mathsf{Open}([k_j^w])$; $r_j'^w \leftarrow \mathsf{Open}([r_j^w])$;
   (f) If $(\boldsymbol{y_j'^w}, t_j'^w, k_j'^w, r_j'^w) \neq (\boldsymbol{y_j^w}, t_j^w, k_j^w, r_j^w)$, W_i aborts, otherwise $[x_j] \leftarrow [0]$.
3. Compute: $[\boldsymbol{z}] = f([\boldsymbol{x_1}], \ldots, [\boldsymbol{x_n}])$;
4. Compute: $[\boldsymbol{c}] = [\boldsymbol{z}] + [\boldsymbol{r}]$;
5. $\boldsymbol{c} \leftarrow \mathsf{Open}([\boldsymbol{c}])$;

Client Output Phase:

1. (Each worker W_i) Send c_j to C_j;
2. Let $c_{j,i}$ be the output that C_j receives from W_i;
3. If $\exists i_0, i_1$ such that $c_{j,i_0} \neq c_{j,i_1}$, then C_j outputs $\texttt{abort}$, else let $c_j = c_{j,1}$;
4. C_j outputs $z_j = c_j - r_j$.

Figure 5. The protocol coping with malicious clients.

$\delta > e \cdot n$, abort the entire recursion and compute $\cup_{j=1}^{n} \mathsf{FIND}([\alpha_j^w])$ as result instead. The default value used for inconsistent client input is 0. We assume that [0] has been created at the beginning of the protocol.[6]

4.2 Eventual Output Consensus

One problem with the above protocol is that it might happen that some honest clients learn their outputs and some other honest clients do not learn their outputs. This can be provoked by a single malicious worker. If the outputs are used as input in a later protocol (for instance, a later execution of the outsourcing protocol itself), this might cause some of the honest clients to start the later protocol while some other honest clients will never start the later protocol. This might lead to problems with functionality, deadlock and also security problems, as it might wash out the fraction of honest clients participating in the ensuring computation.

We will describe a generic and efficient way to achieve a property which we call *eventual output consensus*, meaning that *either* no honest clients receive an output *or* all honest clients will eventually receive their output. This will, e.g., ensure that either no honest clients will continue with a later protocol, or all honest clients will eventually start executing the later protocol.

We now describe our solution. As a first modification, all workers will send $\boldsymbol{c}$ to all clients, the workers will sign the value $\boldsymbol{c}$, and the clients will accept only if they receive the same $\boldsymbol{c}$ from all workers along with valid signatures from all workers. If so, the client C_j will retrieve c_j from $\boldsymbol{c}$ and output $z_j = c_j - r_j$ as usual. Whenever a client C_j outputs a value c_j, it will forward $\boldsymbol{c}$ to all other clients, along with the signatures of all workers. If a client C_j has not yet given output and receives such a $\boldsymbol{c}$ signed by all workers, it will in turn retrieve c_j from $\boldsymbol{c}$, output $z_j = c_j - r_j$, and forwards $\boldsymbol{c}$ to all clients along with the signatures. It is easy to see that this is secure under the assumptions that at least one worker is honest, as all values used to determine outputs are signed by all workers and therefore also the honest worker.

One might wonder if we can do better than eventual agreement on the output. It turns out that we can not. In our model all but one worker can be corrupted and any number of clients can be corrupted. This means that all-in-all more than half of the participants might be corrupted. It is well-known that in a setting without honest majority, generic MPC cannot guarantee termination or fairness. I.e., we cannot ensure that all honest clients will learn their outputs and we cannot even ensure that it does not happen that the corrupted clients learn their outputs and no honest clients learn their outputs. The best we can hope for is therefore that at least the honest clients have consensus on whether outputs were gotten or not. Since we consider asynchronous communication, where we assume that all messages between honest clients are eventually delivered but has no upper bound on the communication delay, we can only hope for this consensus to eventually arise, which is exactly what our protocol achieves.

5 Acknowledgements

The authors are partially supported by the European Research Commission Starting Grant 279447 and The National Science Foundation of China (grant 61061130540) for the Sino-Danish Center for the Theory of Interactive Computation. The first author is supported by The Danish Council for Independent Research Starting Grant 10-081612. The third author is supported by The Danish Council for Independent Research (DFF) Grant 11-116416/FTP.

[6] [0] can be made by first letting any worker W_i do $[c] \leftarrow \mathsf{Input}(W_i, c)$. Then run $c \leftarrow \mathsf{Open}([c])$ and have all workers verify that $c = 0$.

References

[BCD+09] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In *Financial Cryptography*, pages 325–343, 2009.

[BDO14] Carsten Baum, Ivan Damgård, and Claudio Orlandi. Publicly auditable secure multi-party computation. SCN 2014. Available as Cryptology ePrint Archive, Report 2014/075, 2014. `http://eprint.iacr.org/`.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT*, pages 169–188, 2011.

[BFR13] Michael Backes, Dario Fiore, and Raphael M. Reischuk. Verifiable delegation of computation on outsourced data. In *ACM Conference on Computer and Communications Security*, pages 863–874, 2013.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, pages 192–206, 2008.

[BSCG+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *CRYPTO (2)*, pages 90–108, 2013.

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. In *FOCS*, pages 97–106, 2011.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

[CDF+08] Ronald Cramer, Yevgeniy Dodis, Serge Fehr, Carles Padró, and Daniel Wichs. Detection of algebraic manipulation with applications to robust secret sharing and fuzzy extractors. In *EUROCRYPT*, pages 471–488, 2008.

[CLT14] Henry Carter, Charles Lever, and Patrick Traynor. Whitewash: Outsourcing garbled circuit generation for mobile devices. Cryptology ePrint Archive, Report 2014/224, 2014. `http://eprint.iacr.org/`.

[CMF+14] Koji Chida, Gembu Morohashi, Hitoshi Fuji, Fumihiko Magata, Akiko Fujimura, Koki Hamada, Dai Ikarashi, and Ryuichi Yamamoto. Implementation and evaluation of an efficient secure computation system using 'r' for healthcare statistics. *Journal of the American Medical Informatics Association*, pages amiajnl–2014, 2014.

[CMTB13] Henry Carter, Benjamin Mood, Patrick Traynor, and Kevin Butler. Secure outsourced garbled circuit evaluation for mobile devices. In *Proceedings of the 22nd USENIX conference on Security*, pages 289–304. USENIX Association, 2013.

[DKL+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure mpc for dishonest majority - or: Breaking the spdz limits. In *ESORICS*, pages 1–18, 2013.

[DO10] Ivan Damgård and Claudio Orlandi. Multiparty computation for dishonest majority: From passive to active security at low cost. In *CRYPTO*, pages 558–576, 2010.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, pages 643–662, 2012.

[DS83] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM J. Comput.*, 12(4):656–666, 1983.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.

[GGP10] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, pages 465–482, 2010.

[GHRW14] Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs. Outsourcing private RAM computation. *IACR Cryptology ePrint Archive*, 2014:148, 2014.

[HL10] Carmit Hazay and Yehuda Lindell. *Efficient Secure Two-Party Protocols: Techniques and Constructions*. Springer-Verlag, 2010.

[HLP11] Shai Halevi, Yehuda Lindell, and Benny Pinkas. Secure computation on the web: Computing without simultaneous interaction. In *CRYPTO*, pages 132–150, 2011.

[KMR11] Seny Kamara, Payman Mohassel, and Mariana Raykova. Outsourcing multi-party computation. *IACR Cryptology ePrint Archive*, 2011:272, 2011.

[KMR12] Seny Kamara, Payman Mohassel, and Ben Riva. Salus: a system for server-aided secure function evaluation. In *ACM Conference on Computer and Communications Security*, pages 797–808, 2012.

[KMRS14] Seny Kamara, Payman Mohassel, Mariana Raykova, and Saeed Sadeghian. Scaling private set intersection to billion-element sets. In *Financial Crypto*, 2014.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. In *ICALP (2)*, pages 486–498, 2008.

[LTV12] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *STOC*, pages 1219–1234, 2012.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *CRYPTO*, pages 681–700, 2012.

[PHGR13] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 238–252, 2013.

[PTK13] Andreas Peter, Erik Tews, and Stefan Katzenbeisser. Efficiently outsourcing multiparty computation under multiple keys. *IEEE Transactions on Information Forensics and Security*, 8(12):2046–2058, 2013.