Certificate Validation in Secure Computation and Its Use in Verifiable Linear Programming

Sebastiaan de Hoogh[1], Berry Schoenmakers[2], and Meilof Veeningen[1]

[1] Philips Research
[2] Eindhoven University of Technology

Abstract. For many applications of secure multiparty computation it is natural to demand that the output of the protocol is verifiable. Verifiability should ensure that incorrect outputs are always rejected, even if all parties executing the secure computation collude. Since the inputs to a secure computation are private, and potentially the outputs are private as well, adding verifiability is in general hard and costly.
In this paper we focus on privacy-preserving linear programming as a typical and practically relevant case for verifiable secure multiparty computation. We introduce certificate validation as an effective technique for achieving verifiable linear programming. Rather than verifying the computation proper, which involves many iterations of the simplex algorithm, we extend the output of the secure computation with a certificate. The certificate allows for efficient and direct validation of the correctness of the output. The overhead incurred by the computation of the certificate is marginal. For the validation of a certificate we design particularly efficient distributed-prover zero-knowledge proofs, fully exploiting the fact that we can use ElGamal encryption for this purpose, hence avoiding the use of more elaborate cryptosystems such as Paillier encryption.
We also formulate appropriate security definitions for our approach, and prove security for our protocols in this model, paying special attention to ensuring properties such as input independence. By means of several experiments performed in a real multi-cloud-provider environment, we show that the overall performance for verifiable linear programming is very competitive, incurring minimal overhead compared to protocols providing no correctness guarantees at all.

1 Introduction

When outsourcing a computation to the cloud, we want to be sure that the result is correct. But if the computation involves confidential inputs, e.g., of multiple mutually distrusting inputters, we also want to guarantee the privacy of the inputs. For instance, solving linear programs is useful for optimising global profits in supply chains [CdH10] or financial benchmarking [DDN+15]; confidentiality is important because the inputs are sensitive information from multiple companies but correctness is important because the outcome represents financial value.

Separately, privacy and correctness can each be achieved. Correctness can be achieved by replicating a computation and comparing the results [CL02] (but

this only protects against uncorrelated failure); or by relying on the use of trusted hardware by the worker [SZJVD04]. Alternatively, correctness can be achieved without assuming uncorrelated failure or trusted hardware, by instead producing cryptographic proofs of correctness (e.g., [PHGR13]).

Achieving privacy is hard when outsourcing to a single cloud worker, but feasible if the computation is distributed between several workers. Indeed, having a single worker perform arbitrary computations on encryptions requires fully homomorphic encryption, a cryptographic primitive that is still impractical for realistic applications. But distributing computations between multiple workers in a privacy-preserving way is possible, and getting more and more practical, using multiparty computation protocols (e.g., [BD09, DKL+13]). Such protocols guarantee privacy and correctness up to a certain threshold of corrupted workers. The inputters can pick workers run at different cloud providers, thereby reducing the risk that too many of them collude or are compromised.

Unfortunately, using such techniques has a major drawback: apart from inputters having to trust the choice of workers for privacy, also recipients have to trust the choice of workers for correctness of their result. However, requiring this trust by the recipients is undesirable: it means recipients (potentially anybody, if the computation result is public) need to be involved in assessing the trustworthiness of workers; and the result may simply have too much value to allow the possibility of incorrectness.

In theory, privacy and correctness can be achieved by producing cryptographic proofs of correctness in a multi-party way. Indeed, this is the basic idea behind recent *universally verifiable* [SV15] (or *publicly auditable* [BDO14]) multiparty computation protocols. (Correctness holds regardless of the workers, but privacy only holds up to a certain maximum of corruptions: we cannot hope to circumvent this in outsourcing scenarios without resorting to fully homomorphic encryption.) However, producing correctness proofs in a multi-party way is too expensive for realistic computations such as linear programming, requiring zero-knowledge proofs involving threshold Paillier encryptions [SV15] or somewhat homomorphic encryptions [BDO14]. Moreover, existing approaches need secure distributed set-up of these threshold cryptosystem, which is hard in practice.

1.1 Our Contribution

In this paper, we present certificate validation as a general technique for achieving verifiable secure computation, and we demonstrate this in detail for verifiable linear programming. While solving a linear program, e.g., by means of the simplex algorithm, is complex and time-consuming, we make the critical observation that the so-called "dual solution" of a linear program allows one to efficiently verify the optimality of the result, without redoing the computation of the result. Thus, we show how to use fast multiparty computation techniques for the computation itself, while limiting the use of slower verifiable techniques to prove the optimality of the result. We avoid the use of expensive encryption schemes such as Paillier's cryptosystem by combining the computation stage and the validation stage in a new way, enabling the use of ElGamal encryption (implemented

using elliptic curves). We show how to enforce inputters to choose their inputs independently, and we prove security in a rigorous security model.

Concretely, our instantiation is with $n = 3$ workers (but can be easily generalised to $n = 2t + 1$ workers). We distribute the computation between all three workers using protocols that guarantee privacy if they do not collude or act maliciously (i.e., deviate from the protocol). Then, two (in general, $t + 1$) of the workers perform certificate validation to guarantee to anybody that the found solution is correct. Hence, we reach a compromise between passive and active security. One the one hand, we provide more security than passively secure multiparty computation because we guarantee correctness (in the sense that the solution is valid with respect to the certificate) regardless of corruptions. On the other hand, we provide less security than actively secure multiparty computation because we do not guarantee privacy if workers collude or act maliciously.

With our new protocol, we are able to demonstrate, for the first time, that certificate validation leads to practically feasible performance. We have implemented our protocol and tested its performance in a linear programming case study, performed in a real multi-cloud-provider environment. As mentioned, our security is in between passive and active; our experiments show that our performance is in fact much closer to passive, adding only little overhead in cases where using active security would be much slower.

1.2 Related Work

Verifiable computation, i.e., the question of how to verify correctness of computations performed by untrusted parties (without privacy) has a long history in the literature (e.g., [DFK+92, AS98, GKR08]). Recently, major practical improvements in efficiency (e.g., [PHGR13]) have shown that in some cases it is possible in practice to verify a computation faster than performing it.

Combining verifiability with privacy has traditionally only been considered for particular applications such as e-voting [CF85,SK95], but recent works [dH12, BDO14, SV15] have also started studying the problem of verifiability for general multiparty computation. In essence (like in our work) the correctness proofs of these works rely on zero-knowledge proofs of correct multiplication and decryption: of Paillier encryptions in [dH12,SV15], and of somewhat homomorphic encryptions in [BDO14]. Compared to these works, this work proposes a private multiplier approach that enables the use of the much more efficient ElGamal encryption scheme (besides introducing the approach of certificate validation).

Another recent line of work has combined verifiability and privacy when outsourcing computations to a single worker. However, known constructions in this line of work are unfortunately inpractical due to their use of costly primitives, e.g., fully homomorphic encryption and verifiable computation [LTV12,FGP14]; or functional encryption and garbled circuits [GKP+13]. Indeed, because such constructions require a single party to compute on encrypted data, even without offering verifiability they are inherently much heavier than our approach.

A final line of work on outsourcing computation combines privacy with "partial" verifiability in the sense that also correctness is only guaranteed if not all

$\mathcal{I}/\mathcal{P}/\mathcal{R}$	inputters/workers/recipients
party/ies P **do** S	party/ies P concurrently perform S
$\mathsf{Enc}_{\mathsf{pk}}(x; r)$	ElGamal encryption of x with public key pk, randomness r
$\mathsf{Dec}_{\mathsf{sk}}(x)$	ElGamal decryption with key (share) sk
p	prime order for ElGamal
$\oplus, \otimes$	homomorphic addition/multiplication of ElGamal ciphertexts
$\mathsf{send}(v; \mathcal{P}); \mathsf{recv}(\mathcal{P})$	send/receive v over secure, private channel (no $\mathcal{P}$ means $\mathcal{P}_1/\mathcal{P}_2$)
$\mathsf{bcast}(v)$	share v on bulletin board
$\textsc{ZkVer}(\Sigma; v; \pi; a)$	Fiat-Shamir proof verification (p. 6)
$a \in_R S$	sample a uniformly random from S
$[x], [x']$	own/other party's additive share of x (for two workers)
$\mathcal{H}$	cryptographic hash function

Fig. 1. Notation in algorithms and protocols

workers collude. This is the case for normal multiparty computation protocols applied in an outsourcing setting (e.g., [JNO14, DDN+15]), but also for specialised outsourcing protocols like [KMR11, ACG+14]. Compared to these works, we *do* offer correctness if all workers collude (without relying on a trusted set-up).

Finally, using short certificates to prove correctness of a larger operation has been proposed before. For instance, it was used to prove the correct execution of graph algorithms in [ZPK14]; see also [TT10] and its references. As far as we know, we are the first to propose the use of certificates for verifiability of multiparty computation. The certificates mentioned in these works may be useful for achieving verifiability of these computations in a privacy-preserving setting.

Outline. We first present a protocol for proving and verifying that a set of encryptions satisfies some given polynomial relations (Section 2). We then show how to combine this protocol with fast, non-verifiable multiparty computation (Section 3). We show with experiments that this gives rise to practical verifiable secure linear programming (Section 4). We finish with a discussion of related and future work (Section 5). Figure 1 shows notation used in this paper.

2 Proving Relations on ElGamal Encryptions

The main idea of our approach is compute a function using multiparty computation; and then prove correctness of the result by proving that the input and result satisfy a number of polynomial relations with a private multiplier-based protocol on ElGamal encryptions. Suppose that in the computation, a collection $X_1, \ldots, X_n$ of ElGamal encryptions has been produced representing the inputs and outputs of the computation, whose correctness we now want to prove. Say these ElGamal encryptions are encrypted under a private key s that is additively shared with threshold t (usually, $t = 1$), i.e, $t + 1$ workers have shares $[s]_1, \ldots, [s]_{t+1}$ such that $s = [s]_1 + \ldots + [s]_{t+1}$. Suppose the $t + 1$ workers have also additively shared the plaintexts and randomness of these encryptions. Then

the workers will together prove in zero knowledge that the encryptions satisfy certain relations, without learning any information about the encrypted values.

The overall approach for producing this proof is the following. For each polynomial relation $r(x_1, \ldots, x_n) = 0$ in values $x_1, \ldots, x_n$, the workers produce an encryption R of the left-hand side value. This requires additions, multiplications by a constant, and multiplications of two encryptions. The first two can be computed locally using homomorphic properties of ElGamal. Multiplications of an encryption Y by an encryption X_i of a shared plaintext x_i can be performed verifiably by letting the workers verifiably multiply their shares, and combining correctness proofs on the shares into an overall correctness proof using the multiparty Fiat-Shamir transform [SV15]. Finally, a proof that R decrypts to zero is made by homomorphically combining decryption proofs using shares $[s]_i$.

In the remainder of this section, we review the threshold homomorphic ElGamal cryptosystem and associated proofs of correct multiplication and decryption, and the multiparty variants from [SV15]. We then discuss how to use these multiplication and decryption proofs to obtain an overall proof that the polynomial relations hold. In Appendix A, we give an explicit two-party protocol.

2.1 Threshold ElGamal and Zero-Knowledge Proofs

First, recall the additively homomorphic ElGamal cryptosystem [El 85]. Given a generator g of a discrete logarithm group of size p (p prime), public keys are group elements h such that $s = \log_g h$ is unknown; the private key is s; encryption of $m \in \mathbb{Z}_p$ with randomness $r \in \mathbb{Z}_p$ is $(g^r, g^m h^r)$; and decryption of (a, b) is $g^m = ba^{-s}$. This cryptosystem is indeed additively homomorphic: if (a, b) encrypts m and (a', b') encrypts m', then $(a \cdot a', b \cdot b')$, denoted $(a, b) \oplus (a', b')$, encrypts $m + m'$. Moreover, if (a, b) encrypts m, then (a^α, b^α), denoted $(a, b) \otimes \alpha$, encrypts $m\alpha$; and $(a^\alpha g^r, b^\alpha h^r)$ is a random encryption of $m\alpha$. Because ElGamal decrypts to g^m and not to m, it is only possible to decrypt small values for which the discrete logarithm problem with respect to g is feasible. Suitable discrete logarithm groups include groups of points on elliptic curves, e.g., [Nat99]. ElGamal is turned into a threshold cryptosystem [Ped91] in which two parties together can perform decryption, by sharing the private key s as $s = s_1 + s_2$: parties can publish their decryptions $D_1 = ba^{-s_1}$, $D_2 = ba^{-s_2}$, from which the overall decryption is computed as $g^m = b(b^{-1}D_1)(b^{-1}D_2)$.[3] Public key shares $h_i = g^{s_i}$ are published that are used to prove correctness of decryption shares.

The correctness of decryption shares and multiplications can be proven using Σ-protocols [CDS94]. Recall that a Σ-protocol for a binary relation R is a three-move protocol in which a potentially malicious prover convinces a honest verifier that he knows a *witness* w for *statement* v such that $(v, w) \in R$. First, the prover sends an *announcement* to the verifier; the verifier responds with a uniformly random *challenge*; and the prover sends his *response*, which the verifier verifies. For our purposes, we need three standard Σ-protocols: proof

[3] Of course, parties can alternatively share a^{-s_1}, a^{-s_2}; we prefer our description because it treats decryption and decryption shares uniformly.

of plaintext knowledge Σ_{PK}, proof of correct multiplication Σ_{CM}, and proof of correct decryption Σ_{CD}. Σ_{PK} proves knowledge of plaintext y and randomness r used in the statement $(a, b) = (g^r, h^r g^y)$. Σ_{CM} proves the following: given a statement consisting of encryptions (a_1, b_1), (a_2, b_2), and (a_3, b_3), the prover knows (y, r, s) such that $a_2 = g^r$ and $b_2 = h^r g^y$ (i.e., (a_2, b_2) encrypts plaintext y with randomness r); and $a_3 = a_1^y g^s$ and $b_3 = b_1^y h^s$ (i.e., (a_3, b_3) encrypts the product encryption, randomised with s). For Σ_{CD}, recall that the decryption of plaintext (a, b) with private key (share) s is $D = ba^{-s}$. Correctness of D with respect to public key (share) h is proven by proving knowledge of the value s such that $h = g^s$ and $b = Da^{-s}$ using a standard equality proof.

Σ-protocols can be used to obtain non-interactive zero-knowledge proofs using the well-known *Fiat-Shamir heuristic* [FS86]. Namely, a party proves knowledge of a witness for statement v by generating announcement a; setting challenge $c = \mathcal{H}(v||a||aux)$ with some auxiliary information aux; and using this to computing response r. The proof (a, c, r) can be verified by checking that (a, c, r) verify, and $\mathcal{H}(v||a||aux)$ gives c. If a follows from c and r, then the proof can be shortened to (c, r), which is accept if $\textsc{ZkVer}(\Sigma; v; c, r; aux)$ holds, where $\textsc{ZkVer}(\Sigma; v; c, r; aux) := (\mathcal{H}(v||\Sigma.\mathsf{rea}(v; c; r)||aux) \stackrel{?}{=} c)$. Security is in the random oracle model, an idealised model of hash functions. To prove multiple statements v_i, the same challenge can be used for all the proofs by computing announcements a_i and setting $c = \mathcal{H}(v_1||a_1||v_2||a_2||\ldots||aux)$.

For our Σ-protocols Σ_{PK}, Σ_{CD} and Σ_{CM}, *homomorphisms* [SV15] exist that allow provers to combine proofs for different statements into one single proof. Suppose we have an encryption X and a series of encryptions Y_i, Z_i such that Z_i is an encryption of the product of the plaintexts of X and Y_i. Then separate instances of Σ_{CM} can be used to prove that the Z_i are indeed product encryptions. However, if the transcripts (a_i, c, r_i) of these proofs all share the same challenge, then these transcript can be "homomorphically combined" into one transcript that proves that $\oplus Z_i$ is the product encryption of X and $\oplus Y_i$. The combination function simply takes the product of all elements of the announcement, and the sum of all elements of the response. If a verifier is just interested in X, $\oplus Y_i$ and $\oplus Z_i$ then the verifier no longer needs to verify the individual proofs. Similarly, a homomorphism for Σ_{PK} combines proofs of plaintext knowledge for (a_i, b_i) into a proof of knowledge for $(\prod a_i, \prod b_i)$. That is, it combines proofs of knowledge of the plaintexts of X_i into one proof of (collective) knowledge of the plaintext of $\oplus X_i$. A homomorphism for Σ_{CD} combines proofs of correct decryption of (a, b) to shares D_i with respect to public key shares h_i into a proof of correct decryption of (a, b) to $b\prod(b^{-1}d_i)$ with respect to public key $h = \prod h_i$. Also these homomorphisms also take the announcements' product and the responses' sum.

Using the above homomorphisms, it is possible to obtain non-interactive zero-knowledge proofs of combined statements. Suppose parties with statements $v_1, \ldots, v_t$ want to produce a series of proofs for combined statement v. They exchange announcements $a_1, \ldots, a_t$ for their shares $v_1, \ldots, v_t$; compute combined announcement a; take challenge $h = \mathcal{H}(v||a||aux)$; and exchange responses $r_1, \ldots, r_t$. The combined r with the challenge h proves collective knowledge of the

witness corresponding to statement v. For security reasons, parties should not not be able to choose a_i based those of others. To ensure this, before exchanging a_i the parties should first exchange commitments to these values. As above, it is possible to use the same challenge for multiple combined proofs. [SV15] proves the desirable notions of soundness and zero knowledge.

2.2 Proving and Verifying Polynomial Relations

We now present an overview of our PolyProve protocol for producing a proof that ElGamal encryptions $X_1, \ldots, X_n$ satisfy a given set of polynomial relations. $\text{PolyProve}^{\mathcal{E},\mathcal{G}}(\mathsf{pk}; [\mathsf{pk}]; [\mathsf{sk}]; X_1, \ldots, X_n; [x_1], \ldots, [x_n]; [r_1], \ldots, [r_n])$ has two sets of inputs. First, the ElGamal public key pk and secret-shares $[\mathsf{pk}]$, $[\mathsf{sk}]$ of this key and the corresponding private key. Second, encryptions $X_1, \ldots, X_n$, and secret-shares of the respective plaintexts $[x_i]$ and randomness $[r_i]$. The set of relations to be proven is formalised by structures $\mathcal{E}$ and $\mathcal{G}$. $\mathcal{E}$ is a set of equations $x_j = 0$ ($1 \leq j \leq N$ for some $N \geq n$). $\mathcal{G}$ is an arithmetic circuit to compute values x_j for $j > n$. Specifically, $\mathcal{G}$ consists of gates $x_k = v$, $x_k = x_i + x_j$, $x_k = x_i \cdot v$, and $x_k = x_i \cdot x_j$ (v any constant). For multiplication $x_k = x_i \cdot x_j$, we require $1 \leq j \leq n$: for these encryptions the workers have shared the randomness, which we will need to produce the proof. (Clearly, any set of polynomial relations can be described by such $\mathcal{E}$ and $\mathcal{G}$.) The protocol proceeds in the following steps:

- The first step of the protocol is to evaluate the circuit to obtain encryptions $X_{n+1}, \ldots, X_N$. All gates except $x_k = x_i \cdot x_j$ can be evaluated locally; for $x_k = x_i \cdot x_j$, the parties use their additive shares of the plaintext of X_j to obtain shares of X_k, randomised using randomness $[s_k]$. The parties exchange these shares so that, at the end of this step, all parties know all encryptions $X_1, \ldots, X_N$.
- Then, the parties use the multiparty Fiat-Shamir transform to produce combined proofs of correctness of the multiplications in the arithmetic circuit $\mathcal{G}$ for $X_{n+1}, \ldots, X_N$.
- After verifying the correctness of all multiplication proofs, the parties can now safely decrypt encryptions X_j for all equations $x_j = 0$: first, they produce decryption shares with associated proofs of correctness, and then they use the multiparty Fiat-Shamir transform to produce a proof that the combination of the decryption shares produces zero. (Note that it is not necessary to exchange the decryption shares since the result is zero by assumption.)

We remark that in the case of two parties, a slight optimization to the multiparty Fiat-Shamir transform is possible. Namely, instead of each party having to commit to each announcement before opening it, it is sufficient for the first worker to commit to its announcement; the second party to provide its announcement; and the first party to open its commitment. In Appendix A we explicitly give the above PolyProve algorithm which includes this optimization.

The corresponding algorithm $\text{PolyVer}^{\mathcal{E},\mathcal{G}}(\mathsf{pk}; X_1, \ldots, X_n; \pi)$ checks if the proof π produced by PolyProve is correct. The algorithm takes as arguments

the public key pk, encryptions $X_1, \ldots, X_n$, and proof π as above. First, it computes missing encryptions in $\{X_{n+1}, \ldots, X_N\}$, i.e., of gates that are not inputs or multiplication results, using the homomorphic properties of ElGamal. Then, it verifies all multiplication and decryption proofs. Using the above techniques, this check corresponds to recomputing the announcements and checking if everything hashes to the correct multiplication proof and decryption proof challenges. Details appear in Appendix A.

3 Combining Computation and Validation

We now present our main protocol for privacy-preserving outsourcing with correctness guarantees. We compute a solution and a so-called "certificate" using normal multiparty computation, and then produce a proof that the solution is valid with respect to the certificate using the above ElGamal-based proofs.

3.1 Certificates and Validating Functions

To efficiently validate a computation result, we use certificates. In complexity theory, a certificate is a proof that a value lies in a certain set, that can be verified in polynomial time (see [Hro01]):

Definition 1. *Let $\mathcal{S}_1, \mathcal{S}_2$ be sets and $\mathcal{X} \subseteq \mathcal{S}_1$. A polynomial time computable predicate $\phi \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ is called a* validating function *for $\mathcal{X}$ if $\mathcal{X} = \{w \in \mathcal{S}_1 \mid \exists c \in \mathcal{S}_2 \ : \ \phi(w, c)\}$. If $\phi(w, c)$, then c is a* certificate *of the fact that $w \in \mathcal{X}$.*

E.g., let $S_1 = \{x \in \mathbb{N} \mid \exists y \in \mathbb{Z} \ : \ y^2 = x\}$ be the squares, then $\phi(x, y) := x \stackrel{?}{=} y^2$ is a validating function, and, ± 2 are certificates of the fact that $4 \in \mathcal{S}_1$.

In our case, a computation is given by a computation function $(\boldsymbol{a}, \boldsymbol{r}) = f(\boldsymbol{x})$ and a validating function $\phi(\boldsymbol{x}, \boldsymbol{a}, \boldsymbol{r})$. Here, on input $\boldsymbol{x}$, function f computes function output $\boldsymbol{r}$ and certificate $\boldsymbol{a}$; validating function ϕ checks that $\boldsymbol{r}$ is a valid output with respect to $\boldsymbol{x}$ and $\boldsymbol{a}$. We require that if $(\boldsymbol{a}, \boldsymbol{r}) = f(\boldsymbol{x})$, then $\phi(\boldsymbol{x}, \boldsymbol{a}, \boldsymbol{r})$, but we do not demand the converse: the outcome of the computation might not be unique, and ϕ might merely check that *some* correct solution was found, not that it was produced according to algorithm f. (For instance, a square root finder may return the positive square root while negative square root is also valid.) In our case study, we use that the optimality of a solution to a linear program can be efficiently validated using a certificate, but this concept is more generally applicable: for instance, see [TT10, ZPK14] and references therein.

3.2 Security Properties

We now provide a high-level overview of our setting and the relevant security properties. We consider a setting in which m inputters $\mathcal{I}_1, \ldots, \mathcal{I}_m$ want to perform a computation on their respective inputs $\boldsymbol{x} = x_1, \ldots, x_m$. As above, the computation is given by a function $(\boldsymbol{a}, \boldsymbol{r}) = f(\boldsymbol{x})$ and validating function

Table 1. Security properties and conditions on workers

Property	**Satisfied if...**
Correctness	Always
Input independence	Always
Privacy	No malicious and $\leq t$ colluding workers
Independence of robustness	No malicious and $\leq t$ colluding workers

$\phi(\boldsymbol{x}, \boldsymbol{a}, \boldsymbol{r})$, where $\boldsymbol{r}$ is the outcome of the computation and $\boldsymbol{a}$ is the certificate. We assume that ϕ is given as a set of polynomial relations. The computation is distributed among n workers $\mathcal{P}_1, \ldots, \mathcal{P}_n$, using (t, n) Shamir sharing with $n = 2t + 1$. One single recipient $\mathcal{R}$ obtains the result (we later discuss changes when multiple parties need to get the result).

We will guarantee different security properties in different situations. We require *correctness* of the computation result, in the sense that it satisfies ϕ, regardless of which parties are corrupted.[4] *Privacy* means that nobody learns information about the honest parties' inputs (apart from the recipient learning the function result); we guarantee it if the workers are non-malicious (i.e., they do not deviate from the protocol) and do not collude with each other (they may collude with inputters or the recipient). *Input independence* means that corrupted inputters cannot choose their input depending on honest inputs (note that this is not implied by privacy as we also want to prevent corrupted inputters from copying honest inputs); we guarantee this property regardless of which parties are corrupted. A final property often considered in this setting is *robustness*, i.e., the guarantee that parties cannot stop the computation from reaching a result; we do not aim for this property, and in fact, any inputter can make the computation break down by providing incorrect inputs. However, we do guarantee *independence of robustness* in the sense that parties cannot decide to make the computation break down depending on the inputs of honest inputters, if none of the workers collude or act maliciously.

Our security guarantees indeed (as said before) lie strictly between active and passive security for multiparty computation. Indeed, passively secure protocols do not guarantee correctness or input independence if there are malicious workers (which we do); but actively secure protocols guarantee correctness, privacy, and independence of robustness also with malicious workers (which we do not). We summarise our security properties, and the conditions on the workers under which they are satisfied, in Table 1. In Section 3.4, we will formalise these properties and state a security theorem for our protocol.

3.3 The VerMPC Protocol

We now present our VerMPC protocol providing the above security guarantees.

To compute $(\boldsymbol{a}, \boldsymbol{r}) = f(\boldsymbol{x})$, we use passively secure multiparty computation protocols based on (t, n)-Shamir sharing with $n = 2t + 1$. In these protocols,

[4] As mentioned, depending on ϕ this may not imply the result is computed using f.

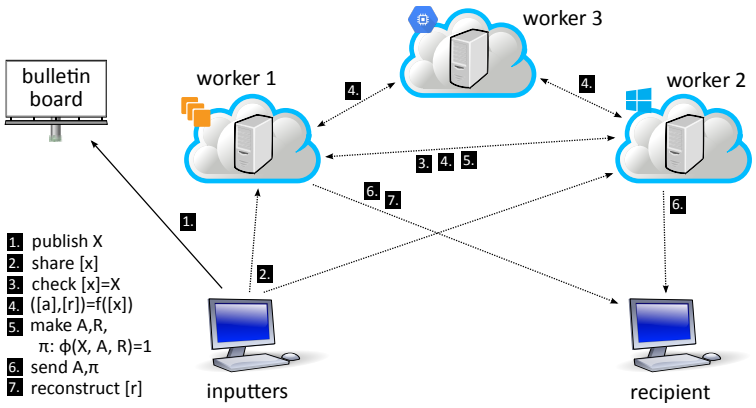

Fig. 2. VerMPC protocol with three workers (dotted lines are private, secure channels)

private values are information-theoretically shared between the n parties such that $t+1$ parties are needed to recover the value. Protocols exist to, e.g., multiply, bit-decompose, compare, and open these shared values (see [dH12] for an overview); these protocols are secure against passively corrupted, non-colluding workers. Note that the computation of f involves n parties and uses Shamir shares, whereas PolyProve involves $t+1$ parties and uses additive shares. It is easy to switch between the two: $t+1$ parties holding additive shares can Shamir-share them among all n; and $t+1$ of the n parties holding Shamir shares can locally convert them to additive shares by Lagrange interpolation.

Given a multiparty computation protocol to compute $([a_1],\ldots,[r_1],\ldots,[r_l]) \leftarrow f([x_1],\ldots,[x_m])$ and the protocol $\text{PolyProve}^{\mathcal{E}_\phi,\mathcal{G}_\phi}(X_1,\ldots;[x_1],\ldots;[r_{x,1}],\ldots)$ to prove that this result is correct, the question is how to combine them in a secure way. Figures 2, 3 show our VerMPC protocol, for concreteness instantiated with three workers ($t=1$, $n=3$). It consists of the following steps:

Step 1. First, the inputters announce their inputs. Each inputter encrypts its input (line 3), and makes a proof of knowledge of the corresponding plaintext (lines 4). These encryptions and proofs are posted on a bulletin board. To prevent corrupted inputters from adaptively choosing their input based on the inputs of others, this happens in two rounds: first, each inputter provides a hash as commitment to its input; having received the commitments of the other inputters, it then reveals the actual encrypted input and proof (line 6). If anybody provides an incorrect input or proof, the protocol is terminated (line 7).

Step 2. Next, the inputters provide additive secret shares of the plaintext x and randomness s of the encryption to the $t+1$ workers who will later perform the PolyProve protocol (line 8).

Step 3. The $t+1$ workers check if the provided sharing of the input is consistent with the encryptions that were posted in step 1. (Without this check, the recipient could learn information of the function output both on the encrypted and the secret-shared inputs, which should not be possible.) They do this by simply

Require: pk/sk ElGamal public/private keys shared by $\mathcal{P}_1$, $\mathcal{P}_2$; $\boldsymbol{x} = x_1, \ldots, x_m$ inputs
Ensure: Recipient $\mathcal{R}$ returns either $\boldsymbol{r}$ with $\phi(\boldsymbol{x}, \boldsymbol{a}, \boldsymbol{r})$ for some $\boldsymbol{a}$, or $\perp$

1: **protocol** VERMPC$^{f,\phi}(\mathsf{pk}; [\mathsf{pk}]; [\mathsf{sk}]; \{x_i\}_{i\in\mathcal{I}})$
2: **parties** $\mathcal{I}_1, \ldots, \mathcal{I}_m$ **do** ▷ step 1
3: $r_{\mathrm{x},i} \in_R \mathbb{Z}_p$; $X_i \leftarrow \mathsf{Enc}_{\mathsf{pk}}(x_i; r_{\mathrm{x},i})$
4: $(a_i, s_i) \leftarrow \Sigma_{\mathrm{PK}}.\mathsf{ann}(X_i; x_i, r_{\mathrm{x},i})$; $c_i \leftarrow \mathcal{H}(X_i||a_i||i)$
5: $r_i \leftarrow \Sigma_{\mathrm{PK}}.\mathsf{res}(X_i; x_i, r_{\mathrm{x},i}; a_i; s_i; c_i)$; $\pi_{\mathrm{x},i} \leftarrow (c_i, r_i)$
6: $h_i \leftarrow \mathcal{H}(i||X_i||\pi_{\mathrm{x},i})$; $\mathsf{bcast}(h_i)$; $\mathsf{bcast}(X_i, \pi_{\mathrm{x},i})$
7: **if** $\exists j : h_j \neq \mathcal{H}(j||X_j||\pi_{\mathrm{x},j}) \vee \neg$ZKVER$(\Sigma_{\mathrm{PK}}; X_j; \pi_{\mathrm{x},j}; j)$ **then return** $\perp$
8: $x'_i \in_R \mathbb{Z}_p$; $r'_{\mathrm{x},i} \in_R \mathbb{Z}_p$; $\mathsf{send}(x'_i, r'_{\mathrm{x},i}; \mathcal{P}_1)$; $\mathsf{send}(x_i - x'_i, r_{\mathrm{x},i} - r'_{\mathrm{x},i}; \mathcal{P}_2)$ ▷ st 2
9: **parties** $\{\mathcal{P}_1, \mathcal{P}_2\}$ **do**
10: **for all** $1 \leq i \leq m$ **do**
11: $[x_i], [r_{\mathrm{x},i}] \leftarrow \mathsf{recv}(\mathcal{I}_i)$; $[X_i] \leftarrow \mathsf{Enc}_{\mathsf{pk}}([x_i]; [r_{\mathrm{x},i}])$; $\mathsf{send}([X_i])$ ▷ step 3
12: $[X'_i] \leftarrow \mathsf{recv}()$; **if** $X_i \neq [X_i] \oplus [X'_i]$ **then return** $\perp$
13: **parties** $\{\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3\}$ **do** $([a_1], \ldots, [a_k], [r_1], \ldots, [r_l]) \leftarrow f([x_1], \ldots, [x_m])$ ▷ st 4
14: **parties** $\{\mathcal{P}_1, \mathcal{P}_2\}$ **do** ▷ step 5
15: **for all** $1 \leq i \leq k$ **do**
16: $[r_{\mathrm{a},i}] \in_R \mathbb{Z}_p$; $[A_i] \leftarrow \mathsf{Enc}_{\mathsf{pk}}([a_i]; [r_{\mathrm{a},i}])$; $\mathsf{send}([A_i])$; $[A'_i] \leftarrow \mathsf{recv}()$; $A_i \leftarrow [A_i] \oplus [A'_i]$
17: **for all** $1 \leq i \leq l$ **do**
18: $[r_{\mathrm{r},i}] \in_R \mathbb{Z}_p$; $[R_i] \leftarrow \mathsf{Enc}_{\mathsf{pk}}([r_i]; [r_{\mathrm{r},i}])$; $\mathsf{send}([R_i])$; $[R'_i] \leftarrow \mathsf{recv}()$; $R_i \leftarrow [R_i] \oplus [R'_i]$
19: $\pi \leftarrow$ POLYPROVE$^{\mathcal{E}_\phi, \mathcal{G}_\phi}(\mathsf{pk}; [\mathsf{pk}]; [\mathsf{sk}]; X_1, \ldots, R_l; [x_1], \ldots; [r_{\mathrm{x},1}], \ldots)$
20: $\mathsf{send}(\{[r_i], [r_{\mathrm{r},i}]\}_{i=1,\ldots,l}; \mathcal{R})$ ▷ step 6
21: **party** $\mathcal{P}_1$ **do** $\mathsf{send}(A_1, \ldots, A_k, \pi; \mathcal{R})$ ▷ step 7
22: **party** $\mathcal{R}$ **do**
23: $\{[r_i]^{(1)}, [r_{\mathrm{r},i}]^{(1)}\}_{i=1,\ldots,l} \leftarrow \mathsf{recv}(\mathcal{P}_1)$; $\{[r_i]^{(2)}, [r_{\mathrm{r},i}]^{(2)}\}_{i=1,\ldots,l} \leftarrow \mathsf{recv}(\mathcal{P}_2)$
24: $(A_1, \ldots, A_k, \pi) \leftarrow \mathsf{recv}(\mathcal{P}_1)$
25: **for all** $1 \leq i \leq m$ **do if** $\neg$ZKVER$(\Sigma_{\mathrm{PK}}; X_i; \pi_{\mathrm{x},i}; j)$ **then return** $\perp$
26: **for all** $1 \leq i \leq l$ **do** $R_i \leftarrow \mathsf{Enc}_{\mathsf{pk}}([r_i]^{(1)} + [r_i]^{(2)}; [r_{\mathrm{r},i}]^{(1)} + [r_{\mathrm{r},i}]^{(2)})$
27: **if** $\neg$POLYVER$^{\mathcal{E}_\phi, \mathcal{G}_\phi}(\mathsf{pk}; X_1, \ldots, R_l; \pi)$ **then ret** $(r_1, \ldots, r_l)$ **else ret** $\perp$

Fig. 3. VERMPC protocol with three workers

encrypting their shares of the inputs using their shares of the randomness; exchanging the result; and checking correctness using the homomorphic property of the cryptosystem (lines 11–12). (Note that this works because ElGamal is not only homomorphic in the plaintext but also in the randomness.)

Step 4. Then, the actual computation takes place (line 13). This is the only step that involves the additional workers. The $t + 1$ workers holding additive shares of the input Shamir-share them between all n workers; then the computation is performed between the n workers; and finally, $\mathcal{P}_1, \ldots, \mathcal{P}_{t+1}$ locally convert their Shamir shares to additive shares $[a_i], [r_i]$.

Step 5. $t + 1$ workers produce the encrypted result and prove its correctness: They exchange encryptions of their respective additive shares of the certificate and result (line 15–18). They run the POLYPROVE protocol from Section 2.2 to obtain a proof that $\phi(X, A, R) = 1$ (line 19). The arithmetic circuit for ϕ should be such that each certificate value A_i and result value R_i occurs at least once as

1: **function** IVERMPC$^{f,\phi}$
2: **for all** honest inputters $\mathcal{I}_i$ **do** get x_i from party $\mathcal{I}_i$ ▷ input phase
3: **for all** corrupted inputters $\mathcal{I}_i$ **do** get x_i from adversary $\mathcal{S}$
4: **if** $< t$ passively corrupted workers **then** ▷ computation phase
5: compute result, certificate $\boldsymbol{r}; \boldsymbol{a} \leftarrow f(\boldsymbol{x})$
6: **else if** $\geq t$ passively corrupted workers **then**
7: send honest inputs $\{x_i\}_{i \in \mathcal{I} \setminus C}$ to adversary $\mathcal{S}$
8: compute result, certificate $\boldsymbol{r}; \boldsymbol{a} \leftarrow f(\boldsymbol{x})$
9: **if** active inputter, $\mathcal{S}$ sends $\perp$ **then** $\boldsymbol{r} \leftarrow \perp, \ldots, \perp$
10: **else** ▷ actively corrupted workers
11: send honest inputs $\{x_i\}_{i \in \mathcal{I} \setminus C}$ to adversary $\mathcal{S}$
12: get result, certificate $\boldsymbol{r}; \boldsymbol{a}$ from adversary $\mathcal{S}$
13: **if** any x_i is $\perp$ or $\phi(\boldsymbol{x}; \boldsymbol{a}; \boldsymbol{r})$ does not hold **then** set result $\boldsymbol{r} \leftarrow \perp, \ldots, \perp$
14: send result $\boldsymbol{r}$ to recipient $\mathcal{R}$ ▷ result phase

Fig. 4. Security guarantees captured by ideal-world trusted party

right-hand side of a multiplication: because the workers prove knowledge of these right-hand sides, this guarantees that they know the corresponding plaintexts.
Step 6. The workers send their additive shares of the result and the randomness of their encryption shares $[R_i]$ to the recipient (line 20).
Step 7. One worker sends the encryptions of the certificate and proof of correctness (line 21). The recipient checks the proofs of knowledge provided by the inputters (read from the bulletin board) (line 25); computes the encrypted result $R_1, \ldots, R_l$ from its shares (line 26); and calls PolyVer to verify correctness (line 27): if the proof verifies, plaintext $r_1, \ldots, r_l$ is the computation outcome.

3.4 Formal Security Model and Theorem

To formally state and prove the security of our protocol, we use the standard formalism used for multiparty computation: the ideal/real world paradigm [Can98]. We demand that the outputs of the recipient and the adversary in a protocol execution are distributed similarly to those outputs in an ideal world where the function is computed by an incorruptible trusted party. Because we provide different security guarantees under different conditions (Table 1), depending on the number and type of corruptions, the trusted party gives the adversary the chance to learn inputs or manipulate outputs (cf. [SV15, BDO14]). In the ideal world, the adversary has no chances to break privacy or correctness apart from those explicitly given to it by the trusted party. If for every real-world adversary $\mathcal{A}$ there is an ideal-world adversary $\mathcal{S}_\mathcal{A}$ such that the real-world outputs are distributed the same as in the ideal world, then also real-world adversaries cannot learn or influence more than allowed by the ideal-world trusted party.

Figure 4 shows the algorithm IVERMPC$^{f,\phi}$ of the ideal-world trusted party that captures the privacy and correctness guarantees discussed in Section 3.2. In the *input phase*, the trusted party obtains the inputs from the honest inputters (line 2) and then asks the adversary to provide the inputs on behalf of the

corrupted inputters (line 3). In particular, regardless of corruptions, corrupted inputters cannot choose their inputs depending on those of honest inputters: this captures input independence. (However, they can provide $\perp$ in which case the whole computation will fail, capturing that we do not guarantee robustness.)

In the *computation phase*, we distinguish three different cases. The first, simplest case is when there are fewer than t passively corrupted workers: in this case, the trusted party simply evaluates the function f (line 5), capturing correctness. In the second case, if there are at least t corrupted workers but they are all passive, then we can no longer guarantee privacy, so the trusted party sends the inputs to the adversary (line 7). The trusted party then computes f (line 8). If there are any actively corrupted inputters, then we do not guarantee independence of robustness. Namely, the $\geq t$ corrupted workers learn the honest inputs before the corrupted inputters provide their input shares, so the corrupted inputters can stop participating (but not change their inputs) depending on the honest inputs. We capture this by letting the trusted party ask $\mathcal{S}$ whether it wants to send $\perp$, in which case it sets all inputs to $\perp$ (line 9). In the third case, if there are actively corrupted workers, then the passively secure protocols we use guarantee neither privacy nor correctness, so the trusted party provides the inputs to the adversary (line 11) and asks it to provide the computation result (line 12). Finally, in the *result phase*, the trusted party checks if the computation result satisfies ϕ, and otherwise sets the result to $\perp$, capturing correctness (line 13). The result is then sent to $\mathcal{R}$ (line 14).

We now precisely define the real-world and ideal-world execution models. Let C be a set of corrupted parties, of which A are actively corrupted. Let k be a security parameter. Let adversary $\mathcal{A}$ be a probabilistic polynomial time Turing machine. Define real-world execution

$$\textsc{Real}^{C,A}_{\textsc{VerMPC}^{f,\phi},\mathcal{A}}(k; x_1, \ldots, x_m)$$

as the distribution consisting of the output of the recipient $\mathcal{R}$ and the adversary $\mathcal{A}$ in a protocol run. This run consists of a secure set-up of the threshold ElGamal cryptosystem, returning public key pk shared as threshold public/private keys $[\mathsf{pk}]$, $[\mathsf{sk}]$; followed by an execution of the protocol $\textsc{VerMPC}^{f,\phi}(\mathsf{pk}; [\mathsf{pk}]; [\mathsf{sk}]; \{x_i\}_{i\in\mathcal{I}})$ with adversary $\mathcal{A}$. We assume the communication model of [Can98], i.e., a fully connected, synchronous network with rushing; parties can use private channels and a bulletin board, and all communication is ideally authenticated (see [Can98, SV15] for details).

Similarly, the ideal-world execution given set C of corrupted parties of which A active, adversary $\mathcal{S}$, security parameter k, and inputs $x_1, \ldots, x_m$ is called

$$\textsc{Ideal}^{C,A}_{\textsc{IVerMPC}^{f,\phi},\mathcal{S}}(k; x_1, \ldots, x_m);$$

it is defined as the distribution consisting of the outputs of the recipient $\mathcal{R}$ and the adversary $\mathcal{S}$ in an ideal-world protocol execution. In this execution, all parties communicate securely with an incorruptible trusted party $\mathcal{T}$ executing algorithm $\textsc{IVerMPC}^{f,\phi}$ (Figure 4). Honest inputters send their inputs to $\mathcal{T}$; a

honest recipient gets its output from $\mathcal{T}$; and the adversary $\mathcal{S}$ can send arbitrary messages to $\mathcal{T}$ and return an arbitrary value.

Definition 2. *Protocol Π is a* t-passively secure multiparty computation protocol with certificate validation *if, for all probabilistic polynomial time adversaries corrupting set C of parties and actively corrupting $A \subseteq C$, there exists a probabilistic polynomial time adversary $\mathcal{S}$ such that for all possible inputs $\boldsymbol{x}$:*

$$\textsc{Real}^{C,A}_{\Pi,\mathcal{A}}(k;\boldsymbol{x}) \approx \textsc{Ideal}^{C,A}_{\textsc{IVerMPC}^{f,\phi},\mathcal{S}}(k;\boldsymbol{x}),$$

where $\approx$ denotes computational indistinguishability in security parameter k.

Theorem 1. *Protocol* VerMPC *is a* 2*-passively secure multiparty computation protocol with certificate validation in the random oracle model assuming the decisional Diffie-Hellman problem in the group used for ElGamal encryption is hard.*

We prove this theorem in Appendix B.

Because we use the Fiat-Shamir heuristic for non-interactive zero-knowledge proofs, our construction is only secure in the random oracle model. In this model, evaluations of the hash function $\mathcal{H}$ are modelled as queries to a "random oracle" $\mathcal{O}$ that evaluates a perfectly random function. Although security in the random oracle model does not generally imply security in the standard model, the model is commonly used to devise simple and efficient protocols, and no security problems due to its use are known. In particular, our variant of the model [SV15] assumes that the random oracle has not been used before the protocol starts: in practice, it should be instantiated with a keyed hash function, with every computation using a fresh random key.

3.5 Extensions

Input range checking. The multiparty computation protocols used to compute f may only guarantee correctness and privacy if their inputs x are bounded, e.g., $-2^k \leq x \leq 2^k$. To guarantee that the inputs of corrupted parties lie in this range, it is possible to use statistically secure additive shares over the integers in line 8 of the protocol, i.e., by choosing x'_i at random from $[-2^{k-1}, \ldots, 2^{k-1}]$. The workers check if the shares they receive in line 11 lie in this range. Privacy of honest inputs is guaranteed if they are smaller than 2^{k-1} by a statistical security parameter.

Multiple recipients and universal verifiability. In our model, only one party learns the result. If multiple parties need to learn the result, then the encrypted outputs $R_1, \ldots, R_l$ should be posted on a bulletin board to ensure consistency. Note that we cannot guarantee fairness as the workers can always choose to send their shares of the result to some recipients but not others.

At the end of the protocol, the recipient obtains not only the result; but also a non-interactive zero-knowledge proof that this result is correct. In particular, the recipient can also convince third parties that the encrypted outputs $R_1, \ldots, R_l$

are correct. In effect, this protocol achieves what is known as "universal" verifiability [dH12,SV15], although some small changes are needed to obtain security in a [SV15]-like model.

Basing it on Commitments. Verifiability by certificate validation can be based on Pedersen commitments instead of ElGamal encryptions. This requires a few changes; in particular, to prove that a commitments is zero, one needs to know the randomness, hence the randomness of product commitments needs to be computed in a multiparty way. Using Pedersen commitments likely leads to smaller proofs and quicker verification. Also, it is no longer needed to distribute decryption keys to the workers, hence a computation can be outsourced to anybody without preparation. On the other hand, when using Pedersen commitments, whoever knows the trapdoor $\log_g h$ used to set up the commitment scheme, can produce correctness proofs of incorrect computation results. (In the present construction, knowing the trapdoor breaks privacy but not correctness.)

Load Balancing of the 2PC. In the present protocol, two of the three workers produce the proof in line 19 while the third worker does nothing. If it is important to balance the computation load, then it is possible to let the three pairs of workers each produce one third of this proof.

Reducing Memory Load with Less Batching. In the present setup, all multiplication proofs and all decryption proofs share the same challenge. Although this gives the smallest proofs and fastest computation, it also means that the announcements for all those proofs need to be in memory at the same time. Memory usage can be reduced at the expense of a slight increase in proof and computation time by splitting the set of all equations into "blocks" and executing PolyProve and PolyVer for each block.

4 Secure and Verifiable Linear Programming

To demonstrate the feasibility of our approach, we apply it to linear programming. Linear programming is a broad class of optimisation problems occurring in many applications; for instance, it can be used for optimising global profits in supply chains [CdH10] or balancing risks in financial portfolios. Precisely, the problem is to minimise the output of a linear function, subject to linear constraints on its variables. One instance of this problem is called a linear program (LP); it is given by a matrix $\mathbf{A}$ and vectors $\boldsymbol{b}$ and $\boldsymbol{c}$. The vector $\boldsymbol{c} = (c_1, \ldots, c_n)$ gives the linear function $\boldsymbol{c} \cdot \boldsymbol{x} = c_1 \cdot x_1 + \ldots + c_n \cdot x_n$ in variables $\boldsymbol{x} = (x_1, \ldots, x_n)$ that needs to be minimised. The matrix $\mathbf{A}$ and vector $\boldsymbol{b}$ give the constraints $\mathbf{A} \cdot \boldsymbol{x} \leq \boldsymbol{b}$ that need to be satisfied. $\mathbf{A}$ has n columns, and $\mathbf{A}$ and $\boldsymbol{b}$ have m rows, where m is the number of constraints. In addition to these constraints, we require $/x_i \geq 0$. For instance, the LP

$$/\mathbf{A} = \begin{pmatrix} 1 & 2 & 1 \\ 1 & -1 & 2 \end{pmatrix}, \boldsymbol{b} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \boldsymbol{c} = \begin{pmatrix} -10 \\ 3 \\ -4 \end{pmatrix}$$

represents the problem to find x_1, x_2, x_3 satisfying $x_1 + 2x_2 + x_3 \leq 2$, $x_1 - x_2 + 2x_3 \leq 1$, and $x_1, x_2, x_3 \geq 0$, such that $-10x_1 + 3x_2 - 4x_3$ is minimal.

To find the optimal solution of a linear program, typically an iterative algorithm called the *simplex algorithm* is used. Each iteration involves several comparisons and a Gaussian elimination step, making it quite heavy for multiparty computation. For relatively small instances, passively secure linear programming is feasible [BD09, CdH10]; but actively secure MPC much less so when including preprocessing (as we discuss later). Fortunately, given a solution $\boldsymbol{x}$ to an LP, there is an easy way to prove that it is optimal using the optimal solution $\boldsymbol{p}$ of the so-called *dual LP* "maximise $\boldsymbol{b} \cdot \boldsymbol{p}$ such that $\mathbf{A} \cdot \boldsymbol{p} \leq \boldsymbol{c}, \boldsymbol{p} \leq 0$". Namely, it is well known that solutions $(\frac{x_1}{q}, \ldots, \frac{x_n}{q})$ and $(\frac{p_1}{q}, \ldots, \frac{p_m}{q})$ $(\boldsymbol{x} \in \mathbb{Z}^n, \boldsymbol{p} \in \mathbb{Z}^m, q \in \mathbb{N}^+)$ are both optimal if the following conditions hold:

$$q \geq 1; \quad \boldsymbol{p} \cdot \boldsymbol{b} = \boldsymbol{c} \cdot \boldsymbol{x}; \quad \mathbf{A} \cdot \boldsymbol{x} \leq q \cdot \boldsymbol{b}; \quad \boldsymbol{x} \geq 0;$$
$$\mathbf{A}^T \cdot \boldsymbol{p} \leq q \cdot \boldsymbol{c}; \quad \boldsymbol{p} \leq 0.$$

Also, the simplex algorithm for finding $\boldsymbol{x}$ turns out to also directly give $\boldsymbol{p}$. To turn the above criterion into a set of polynomial equations, we define the certificate to consist of bit decompositions of $(q \cdot \boldsymbol{b} - \mathbf{A} \cdot \boldsymbol{x})_i$, $\boldsymbol{x}_i$, $(q \cdot \boldsymbol{c} - \mathbf{A}^T \cdot \boldsymbol{p})_i$, and $-\boldsymbol{p}_i$, and prove that each bit decomposition $b_0, b_1, \ldots$ sums up to the correct value v (with equation $v = b_0 + 2 \cdot b_1 + \ldots$) and contains only bits (with equations $b_i \cdot (1 - b_i) = 0$).

4.1 Cloud Experiments

To assess the performance of our solution, we have performed experiments in a realistic cloud outsourcing setting. Our experiments used a specially developed prototype implementation of our protocols. We took the simplex implementation from the TUeVIFF distribution of VIFF[5], and modified it to produce the certificate of correctness, i.e., the dual solution and required bit decompositions. We implemented the VerMPC protocol from Section 3.3 using SCAPI [EFLL12]. SCAPI is a high-level cryptographic library that supports ElGamal encryption, Σ-protocols Σ_{PK} and Σ_{CD}, and the Fiat-Shamir heuristic; to implement VerMPC, we needed to add threshold decryption, Σ_{CM}, and the PolyProve and PolyVer protocols from Section 2.2. For ElGamal we use the NIST P-224 elliptic curve, supported in SCAPI through the MIRACL library.

To obtain a realistic outsourcing setting, we have deployed the three workers on three different cloud instances from different providers on different continents. See Table 2 for their specifications. In this setup, the inputters and recipient do not have to rely on any single cloud provider or jurisdiction for their privacy: they simply have to assume that the different instances do not collude. All machines ran Ubuntu 14.04.2 LTS. The recipient ran Windows 7 on an Intel i5-5300 (2.30 GHz). The VIFF part of the computation (i.e., step 4) requires authenticated and private channels; these were implemented using SSL. We did not implement steps 1–3 as they contribute minimally to the overall performance of the protocol.

[5] Available at `http://www.win.tue.nl/~berry/TUeVIFF/`

Table 2. Specifications of the workers used in our cloud experiments

#	Location	Provider	Instance	Processor	Memory	€/hour
1	Ireland	Amazon EC2	m3.medium	Xeon @ 2.50GHz	3.75 GB	€0.067
2	Virginia	MS Azure	Standard_D1	Xeon @ 2.20GHz	3.5 GB	€0,070
3	Taiwan	Google GCE	n1-standard-1	Xeon @ 2.50GHz	3.75 GB	€0.050

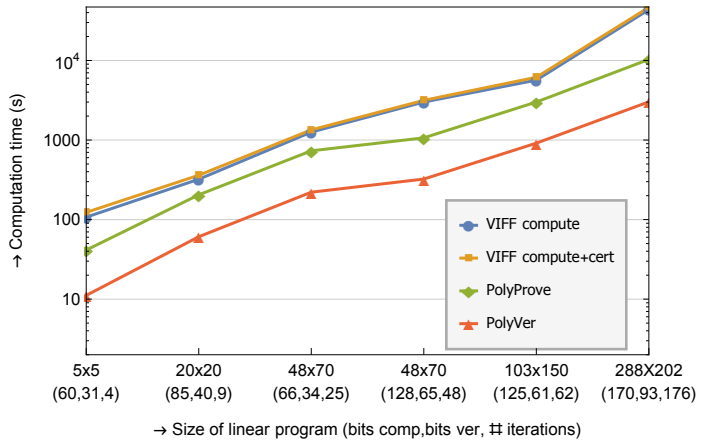

Fig. 5. Computation times of VerMPC on various LPs (the x-axis shows the LP size, bit length for VIFF, bit length for the certificate, and number of iterations)

We ran our experiments on several LPs: randomly-generated small LPs and larger LPs based on Netlib test programs[6]. We measured the time for VIFF to solve the LP and to compute the certificate (this depends on the LP size, number of iterations needed, and the bit length for internal computations); the time for PolyProve to produce a proof; and for PolyVer to verify it (this depends on the LP size and bit length for the proof).[7]

Figure 5 shows the performance numbers of our experiments. For the LPs in our experiments, we find that producing a proof adds little overhead to computing the solution, and that verifying the proof is much faster than participating in the computation. As a consequence, for the recipient, outsourcing both guarantees correctness and saves time compared to participating in the computation. Already in a party with three inputters/recipients, privacy-preserving outsourcing makes sense; with more inputters/recipients, this performance effect is even bigger because computation scales linearly in the number of parties involved.

[6] `http://www.netlib.org/lp/data/`; coefficients rounded for performance

[7] We took the minimal bitlengths needed for correctness. In practice, these are not known in advance: for VIFF, one takes a safe margin; for the proof, one can reveal and use the maximal bit length of all bit decompositions in the certificate.

In general, one expects the difference between computing the solution and proving its correctness to be more pronounced for larger problems: indeed, both the computation and the correctness verification scale in the size of the LP; but computation additionally scales in the number of iterations needed to reach the optimal solution. This number of iterations typically grows with the LP size. However, we only found this for the biggest linear program, where proving is over four times faster than computing; for the other programs, this factor was around two. An explanation for this is that also the bitlength of solutions (which influences proving time) typically grows with the number of iterations.

4.2 Certificate Validation versus Active Security

As discussed, the security guarantees of our model lie in between passive security (that does not guarantee correctness in case of active attacks) and active security (that also guarantees privacy in this case). Above we showed that the overhead of our approach compared to passive security is small; we now compare our performance to that of active security. To get an idea of the performance difference between our approach and active security, we have solved several of our LP instances with an LP solver based on the state-of-the-art SPDZ protocols [DKL$^+$13]. SPDZ combines a slow preprocessing phase, in which many random values are shared between workers, with a fast on-line phase with complexity comparable to passively secure protocols. Hence, after preprocessing has been performed, SPDZ can perform a computation with full privacy and correctness guarantees in about the same time as VIFF (in fact, due to a more efficient implementation, the tested implementation is even faster).

However, preprocessing is slow. No public implementation of the preprocessing phase is available, but it is possible to estimate the time it takes by measuring the amount of randomness needed for the on-line phase and combining this with available preprocessing performance figures [DKL$^+$13]. Even with estimates that are very generous to SPDZ, one finds that the SPDZ preprocessing time is at least 15 times more than the VIFF computation time. For instance, for the first 48-by-70 linear program, we estimate that preprocessing for an actively secure computation takes at least 13 hours; conversely, for out implementation, computation and proving time is close to 35 minutes and verification time is 3.7 minutes. Also, note that the SPDZ timings from [DKL$^+$13] were on a local network whereas our workers are spread over the world; and that the SPDZ timings were for two parties and SPDZ preprocessing scales linearly with the number of parties involved, including all inputters and recipients (whereas our performance does not depend on the number of inputters and recipients). Clearly, outsourcing with certificate validation has favourable performance compared to using SPDZ.

5 Concluding Remarks

In this paper, we have shown how to use certificate validation to obtain correctness guarantees for privacy-preserving outsourcing. In particular, we effi-

ciently instantiate this idea by combining passively secure three-party computation with ElGamal-based proofs. For linear programming, verifying results takes much less time than participating in an actively secure computation; in fact, it even takes less time than participating in a passively secure computation without any correctness guarantees. Hence, for computations on inputs of mutually distrusting parties, privacy-preserving outsourcing with correctness guarantees provides a compelling combination of correctness (always) and privacy (against semi-honest, non-collaborating cloud workers) guarantees.

We see several directions for improvement of our work. We have used passively secure protocols for computation; using protocols that guarantee privacy (but not correctness) also against active attacks would offer stronger protection, possibly at a low performance cost. Our implementation can be optimised, and our alternative construction using Pedersen commitment should have smaller proofs and faster verification. Much bigger speed-ups, however, (especially for linear programming) would come from using efficient zero-knowledge proofs for specific tasks, e.g., for showing that certain values are positive. In particular, range proofs are much faster to verify than our bit-wise proofs; the work of Keller et al. [KMR12] suggests ways of distributing these proofs that could be adapted to our setting. Alternatively, it may be possible to achieve even faster certificate validation by combining verifiable outsourcing techniques with the privacy guarantees of multiparty computation.

References

[ACG+14] P. Ananth, N. Chandran, V. Goyal, B. Kanukurthi, and R. Ostrovsky. Achieving Privacy in Verifiable Computation with Multiple Servers - Without FHE and without Pre-processing. In *Proceedings of PKC '14*, 2014.

[AS98] S. Arora and S. Safra. Probabilistic checking of proofs: A new characterization of NP. *J. ACM*, 45(1):70–122, 1998.

[BD09] P. Bogetoft and Dan Lund Christensen, et al. Secure Multiparty Computation Goes Live. In *Proceedings of Financial Crypto '09*, 2009.

[BDO14] C. Baum, I. Damgård, and C. Orlandi. Publicly Auditable Secure Multi-Party Computation. In *Proceedings of SCN '14*, 2014.

[Can98] R. Canetti. Security and Composition of Multi-party Cryptographic Protocols. *Journal of Cryptology*, 13:2000, 1998.

[CdH10] O. Catrina and S. de Hoogh. Secure multiparty linear programming using fixed-point arithmetic. In *Proc. of ESORICS '10*, pages 134–150, 2010.

[CDS94] R. Cramer, I. Damgård, and B. Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *Proceedings of CRYPTO '94*, 1994.

[CF85] J. Cohen and M. Fischer. A Robust and Verifiable Cryptographically Secure Election Scheme. In *Proceedings of FOCS '85*, pages 372–382. IEEE, 1985.

[CL02] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

[DDN+15] I. Damgård, K. Damgård, K. Nielsen, P. S. Nordholt, and T. Toft. Confidential benchmarking based on multiparty computation. *IACR Cryptology ePrint Archive*, 2015:1006, 2015.

[DFK+92] C. Dwork, U. Feige, J. Kilian, M. Naor, and S. Safra. Low communication 2-prover zero-knowledge proofs for NP. In *Proceedings of CRYPTO '92*, pages 215–227, 1992.

[dH12] S. de Hoogh. *Design of large scale applications of secure multiparty computation: secure linear programming.* PhD thesis, Eindhoven University of Technology, 2012.

[DKL+13] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits. In *Proceedings of ESORICS '13*, 2013.

[EFLL12] Y. Ejgenberg, M. Farbstein, M. Levy, and Y. Lindell. SCAPI: The Secure Computation Application Programming Interface. *Cryptology ePrint Archive, Report 2012/629*, 2012:629, 2012.

[El 85] T. El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[FGP14] D. Fiore, R. Gennaro, and V. Pastro. Efficiently Verifiable Computation on Encrypted Data. In *Proceedings of CCS '14*, 2014.

[FS86] A. Fiat and A. Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *Proc. of CRYPTO '86*, 1986.

[GKP+13] S. Goldwasser, Y. T. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich. Reusable garbled circuits and succinct functional encryption. In *Proc. of STOC '13*, 2013.

[GKR08] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: interactive proofs for muggles. In *Proc. of STOC '08*, pages 113–122, 2008.

[Hro01] J. Hromkovic. *Algorithmics for hard problems - introduction to combinatorial optimization, randomization, approximation, and heuristics.* Springer, 2001.

[JNO14] T. P. Jakobsen, J. B. Nielsen, and C. Orlandi. A framework for outsourcing of secure computation. In *Proceedings of CCSW '14*, pages 81–92, 2014.

[KMR11] S. Kamara, P. Mohassel, and M. Raykova. Outsourcing multi-party computation. *IACR Cryptology ePrint Archive*, 2011:272, 2011.

[KMR12] M. Keller, G. L. Mikkelsen, and A. Rupp. Efficient Threshold Zero-Knowledge with Applications to User-Centric Protocols. In *Proceedings of ICITS '12*, 2012.

[LTV12] A. López-Alt, E. Tromer, and V. Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Proceedings of STOC '12*, pages 1219–1234, 2012.

[Nat99] National Institute of Standards and Technology. Recommended elliptic curves for federal government use, 1999. Available at `http://csrc.nist.gov/encryption`.

[Ped91] T. P. Pedersen. A Threshold Cryptosystem without a Trusted Party (Extended Abstract). In *Proceedings of EUROCRYPT '91*, 1991.

[PHGR13] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly Practical Verifiable Computation. In *Proceedings of S&P '13*, 2013.

[SK95] K. Sako and J. Kilian. Receipt-Free Mix-Type Voting Scheme—A Practical Solution to the Implementation of a Voting Booth. In *Proceedings of EUROCRYPT '95*, volume 921 of *Lecture Notes in Computer Science*, pages 393–403. Springer, 1995.

[SV15] B. Schoenmakers and M. Veeningen. Universally Verifiable Multiparty Computation from Threshold Homomorphic Cryptosystems. Cryptology ePrint Archive, Report 2015/058, 2015.

1: ▷ Relation: $R = \{(a, b; y, r) \mid a = g^r \wedge b = h^r g^y\}$
2: **function** $\Sigma_{\mathrm{PK}}.\mathsf{ann}(a, b; y, r)$ ▷ Announcement
3: $u, v \in_R \mathbb{F}_q$; $c \leftarrow g^v$; $d \leftarrow h^v g^u$; **return** $(c, d; u, v)$
4: **function** $\Sigma_{\mathrm{PK}}.\mathsf{res}(a, b; y, r; c, d; u, v; e)$ ▷ Response
5: $k \leftarrow u + e \cdot y$; $l \leftarrow v + e \cdot r$; **return** (k, l)
6: **function** $\Sigma_{\mathrm{PK}}.\mathsf{sim}(a, b; e)$ ▷ Simulator
7: $k, l \in_R \mathbb{F}_q$; $a \leftarrow g^l a^{-e}$; $b \leftarrow h^l g^k b^{-e}$
8: **return** $(a, b; e; k, l)$
9: **function** $\Sigma_{\mathrm{PK}}.\mathsf{ext}(a, b; c, d; e; e'; k, l; k', l')$ ▷ Extractor
10: $y \leftarrow (k - k')/(e - e')$; $r \leftarrow (l - l')/(e - e')$
11: **return** (y, r)
12: **function** $\Sigma_{\mathrm{PK}}.\mathsf{rea}(a, b; e; k, l)$ ▷ Announcement recomputation
13: $c \leftarrow g^l a^{-e}$; $d \leftarrow h^l g^k b^{-e}$; **return** (c, d)
14: **function** $\Sigma_{\mathrm{PK}}.\mathsf{ver}(a, b; c, d; e; k, l)$ ▷ Verification
15: **return** $a^e \stackrel{?}{=} g^l c^{-1} \ \wedge \ b^e \stackrel{?}{=} h^l g^k d^{-1}$

Fig. 6. Σ_{PK}: Proof of plaintext knowledge

[SZJVD04] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of Usenix Security '04*, 2004.

[TT10] R. Tamassia and N. Triandopoulos. Certification and authentication of data structures. In *Proceedings of AMW '10*, 2010.

[ZPK14] Y. Zhang, C. Papamanthou, and J. Katz. ALITHEIA: towards practical verifiable graph processing. In *Proceedings of CCS '14*, pages 856–867, 2014.

A Details of the PolyProve Protocol

In this Appendix, we present the details of the protocols presented in Section 2.2, specifically in the two-party setting that we have implemented.

A.1 Preliminaries

A Σ-protocol for a binary relation R is a three-move protocol in which a potentially malicious prover convinces a honest verifier that he knows a *witness* w for *statement* v such that $(v, w) \in R$. First, the prover sends an *announcement* (computed using algorithm $\Sigma.\mathsf{ann}$) to the verifier; the verifier responds with a uniformly random *challenge*; and the prover sends his *response* (computed using algorithm $\Sigma.\mathsf{res}$), which the verifier verifies (using predicate $\Sigma.\mathsf{ver}$).

Definition 3. *Let $R \subset V \times W$ be a binary relation with language $L_R = \{v \in V \mid \exists w \in W : (v, w) \in R\}$. Let Σ be a collection of probabilistic polynomial time algorithms $\Sigma.\mathsf{ann}$, $\Sigma.\mathsf{res}$, $\Sigma.\mathsf{sim}$, $\Sigma.\mathsf{ext}$, and polynomial time predicate $\Sigma.\mathsf{ver}$. Let C be a finite set called the* challenge space. *Σ is a* Σ-protocol *for relation R if:*

$\triangleright$ Relation: $R = \{(a_1, b_1, a_2, b_2, a_3, b_3; y, r, s) \mid$
$\triangleright$ $\qquad a_2 = g^r \wedge b_2 = h^r g^y \wedge a_3 = a_1^y g^s \wedge b_3 = b_1^y h^s\}$
function $\Sigma_{\mathrm{CM}}.\mathsf{ann}(a_1, b_1, a_2, b_2, a_3, b_3; y, r, s)$ $\triangleright$ Announcement
  $u, v, w \in_R \mathbb{F}_q$; $a \leftarrow g^v$; $b \leftarrow h^v g^u$; $c \leftarrow a_1^u g^w$
  $d = b_1^u h^w$
  return $(a, b, c, d; u, v, w)$
function $\Sigma_{\mathrm{CM}}.\mathsf{res}(\ldots; y, r, s; \ldots; u, v, w; e)$ $\triangleright$ Response
  $k \leftarrow u + e \cdot y$; $l \leftarrow v + e \cdot r$; $m \leftarrow w + e \cdot s$
  return (k, l, m)
function $\Sigma_{\mathrm{CM}}.\mathsf{sim}(a_1, b_1, a_2, b_2, a_3, b_3; e)$ $\triangleright$ Simulator
  $k, l, m \in_R \mathbb{F}_q$
  $a \leftarrow g^l a_2^{-e}$; $b \leftarrow h^l g^k b_2^{-e}$
  $c \leftarrow a_1^k g^m a_3^{-e}$; $d \leftarrow b_1^k h^m b_3^{-e}$
  return $(a, b, c, d; e; k, l, m)$
function $\Sigma_{\mathrm{CM}}.\mathsf{ext}(\ldots; \ldots; e; e'; k, l, m; k', l', m')$ $\triangleright$ Extractor
  $y \leftarrow (k - k')/(e - e')$; $r \leftarrow (l - l')/(e - e')$
  $s \leftarrow (m - m')/(e - e')$
  return (y, r, s)
function $\Sigma_{\mathrm{PK}}.\mathsf{rea}(a_1, b_1, a_2, b_2, a_3, b_3; e; k, l, m)$ $\triangleright$ Announcement recomputation
  $a \leftarrow g^l a_2^{-e}$; $b \leftarrow h^l g^k b_2^{-e}$
  $c \leftarrow a_1^k g^m a_3^{-e}$; $d \leftarrow b_1^k h^m b_3^{-e}$
  return (a, b, c, d)
function $\Sigma_{\mathrm{CM}}.\mathsf{ver}(a_1, b_1, a_2, b_2, a_3, b_3;$
$\qquad a, b, c, d; e; k, l, m)$ $\triangleright$ Verification
  return $a_2^e \stackrel{?}{=} g^l a^{-1} \ \wedge \ b_2^e \stackrel{?}{=} h^l g^k b^{-1} \ \wedge$
$\qquad a_3^e \stackrel{?}{=} a_1^k g^m c^{-1} \ \wedge \ b_3^e \stackrel{?}{=} b_1^k h^m d^{-1}$

Fig. 7. Σ_{CM}: Proof of correct multiplication

Completeness *If* $(a, s) \leftarrow \Sigma.\mathsf{ann}(v; w)$, $c \in C$, *and* $r \leftarrow \Sigma.\mathsf{res}(v; w; a; s; c)$, *then* $\Sigma.\mathsf{ver}(v; a; c; r)$.

Special soundness *If* $v \in V$, $c \neq c'$, *and* $\Sigma.\mathsf{ver}(v; a; c; r)$ *and* $\Sigma.\mathsf{ver}(v; a; c'; r')$ *both hold, then* $w \leftarrow \Sigma.\mathsf{ext}(v; a; c; c'; r; r')$ *satisfies* $(v, w) \in R$.

Special honest-verifier zero-knowledge *If* $v \in L_R$, $c \in C$, *then* $(a, r) \leftarrow \Sigma.\mathsf{sim}(v; c)$ *has the same probability distribution as* (a, r) *obtained by* $(a, s) \leftarrow \Sigma.\mathsf{ann}(v; w)$, $r \leftarrow \Sigma.\mathsf{res}(v; w; a; s; c)$. *If* $v \notin L_R$, *then* $(a, r) \leftarrow \Sigma.\mathsf{sim}(v; c)$ *satisfies* $\Sigma.\mathsf{ver}(v; a; c; r)$.

Completeness states that a protocol between a honest prover and verifier succeeds; special soundness essentially means that a successful prover must know the witness; special honest-verifier zero-knowledge essentially means that a honest verifier does not learn anything about a witness. For many Σ-protocols, the announcement a can be computed from the challenge c, statement v, the response r, denoted $\Sigma.\mathsf{rea}(v; c; r)$. (This will be useful for concisely storing non-interactive proofs.) Finally, we need that announcements are "non-trivial" [SV15] in the sense that they are random from a large space. Our Σ-protocols satisfy this.

1: ▷ Relation: $R = \{(a, b, D, h; s) \mid D = ba^{-s} \wedge h = g^s\}$
2: **function** $\Sigma_{\mathrm{CD}}.\mathsf{ann}(a, b, D, h; s)$ ▷ Announcement
3: $u \in_R \mathbb{F}_q$; $c \leftarrow a^u$; $d \leftarrow g^u$; **return** $(c, d; u)$
4: **function** $\Sigma_{\mathrm{CD}}.\mathsf{res}(a, b, D, h; s; c, d; u; e)$ ▷ Response
5: $f = u + e \cdot s$; **return** f
6: **function** $\Sigma_{\mathrm{CD}}.\mathsf{sim}(a, b, D, h; e)$ ▷ Simulator
7: $f \in_R \mathbb{F}_q$; $c \leftarrow a^f D^e b^{-e}$; $d \leftarrow g^f h^{-e}$
8: **return** $(b, c; e; f)$
9: **function** $\Sigma_{\mathrm{CD}}.\mathsf{ext}(a, b, D, h; c, d; e; e'; f; f')$ ▷ Extractor
10: $s \leftarrow (f - f')/(e - e')$; **return** s
11: **function** $\Sigma_{\mathrm{PK}}.\mathsf{rea}(a, b, D, h; e; f)$ ▷ Announcement recomputation
12: $c \leftarrow a^f D^e b^{-e}$; $d \leftarrow g^f h^{-e}$; **return** (c, d)
13: **function** $\Sigma_{\mathrm{CD}}.\mathsf{ver}(a, b, D, h; c, d; e; f)$ ▷ Verification
14: **return** $b^e D^{-e} \stackrel{?}{=} a^f c^{-1} \ \wedge \ h^e = g^f d^{-1}$

Fig. 8. Σ_{CD}: Proof of correct decryption (share)

For our purposes, we need three Σ-protocols: proof of plaintext knowledge Σ_{PK}, proof of correct multiplication Σ_{CM}, and proof of correct decryption Σ_{CD}. These protocols are standard; we give them here for completeness. Σ_{PK} (Figure 6) proves knowledge of plaintext y and randomness r used in encryption $(a, b) = (g^r, h^r g^y)$. Σ_{CM} (Figure 7) proves the following: given encryptions (a_1, b_1), (a_2, b_2), and (a_3, b_3), the prover knows (y, r, s) such that $a_2 = g^r$ and $b_2 = h^r g^y$ (i.e., (a_2, b_2) encrypts plaintext y with randomness r); and $a_3 = a_1^y g^s$ and $b_3 = b_1^y h^s$ (i.e., (a_3, b_3) encrypts the product encryption, randomised with s). For Σ_{CD}, recall that the decryption of plaintext (a, b) with private key (share) s is $D = ba^{-s}$. Correctness of D with respect to public key (share) h is proven by proving knowledge of the value s such that $h = g^s$ and $b = Da^{-s}$ using a standard equality proof (Figure 8).

We say that Σ_{CM} has a *homomorphism* [SV15] Φ_{CM} consisting of a statement combining function $\Phi_{\mathrm{CM}}.\mathsf{stmt}(\{(X, Y_i, Z_i)\}) = (X, \oplus Y_i, \oplus Z_i))$, an announcement combining function $\Phi_{\mathrm{CM}}.\mathsf{ann}(\{(a_i, b_i, c_i, d_i)\}) = (\prod a_i, \prod b_i, \prod c_i, \prod d_i)$, and a response combining function $\Phi_{\mathrm{CM}}.\mathsf{resp}(\{(k_i, l_i, m_i)\}) = (\sum k_i, \sum l_i, \sum m_i)$. Similarly, a homomorphism Φ_{PK} for Σ_{PK} combines proofs of plaintext knowledge for (a_i, b_i) into a proof of knowledge for $(\prod a_i, \prod b_i)$. That is, it combines proofs of knowledge of the plaintexts of X_i into one proof of (collective) knowledge of the plaintext of $\oplus X_i$. Homomorphism Φ_{CD} for Σ_{CD} combines proofs of correct decryption of (a, b) to shares D_i with respect to public key shares h_i into a proof of correct decryption of (a, b) to $b \prod (b^{-1} d_i)$ with respect to public key $h = \prod h_i$. Like Φ_{CM}, these homomorphisms also take the product of the announcements and the sum of the responses. (Homomorphisms need to satisfy two technical properties discussed in [SV15]; we mention without proof that these properties hold for the homomorphisms in this paper.)

Σ-protocols can be used to obtain non-interactive zero-knowledge proofs using the well-known *Fiat-Shamir heuristic* [FS86]. Namely, a party proves knowl-

edge of a witness for statement v by generating announcement a using $\Sigma.\mathsf{ann}$; setting challenge $c = \mathcal{H}(v||a||aux)$ with some auxiliary information aux; and computing response r with $\Sigma.\mathsf{res}$. The proof (a, c, r) can be verified by checking that

$$\mathcal{H}(v||a||aux) \stackrel{?}{=} c \wedge \Sigma.\mathsf{ver}(v; a; c; r).$$

If $\Sigma.\mathsf{rea}$ is defined, then the proof can be shortened to (c, r), which a verifier accepts if $\textsc{ZkVer}(\Sigma; v; c, r; aux)$ holds, where

$$\textsc{ZkVer}(\Sigma; v; c, r; aux) := (\mathcal{H}(v||\Sigma.\mathsf{rea}(v; c; r)||aux) \stackrel{?}{=} c).$$

Security holds in the random oracle model, an idealised model of hash functions. If a party needs to prove multiple statements v_i at the same time, then it is possible to use the same challenge for all the proofs by computing announcements a_i and setting $c = \mathcal{H}(v_1||a_1||v_2||a_2||\ldots||aux)$.

The above homomorphisms can be exploited to obtain non-interactive zero-knowledge proofs of combined statements. Suppose two parties want to produce a series of proofs for statements $v_i = \Phi.\mathsf{stmt}(\{v_i', v_i''\})$. They exchange announcements a_i', a_i'' for their shares v_i', v_i'' of v_i; compute $a_i = \Phi.\mathsf{ann}(\{a_i', a_i''\})$; take challenge $h = \mathcal{H}(v_1||a_1||v_2||a_2||\ldots||aux)$; and exchange responses r_i', r_i''. Taking $r_i = \Phi.\mathsf{resp}(\{r_i', r_i''\})$, the challenge h and responses r_i prove collective knowledge of witnesses corresponding to statements v_i. For security reasons, the second party should not be able to choose a_i'' based on a_i'. To ensure this, the first party can first provide a hash of its announcements; the second party then provides its announcements, after which the first party opens the hash. This construction satisfies the desirable notions of soundness and zero knowledge [SV15].

A.2 Details of PolyProve and PolyVer

Figure 9 shows our PolyProve protocol for producing a proof that ElGamal encryptions $X_1, \ldots, X_n$ satisfy a given set of polynomial relations in the two-party setting. The protocol has two sets of inputs. First, the ElGamal public key pk and secret-shares $[\mathsf{pk}]$, $[\mathsf{sk}]$ of this key and the corresponding private key. Second, encryptions $X_1, \ldots, X_n$, and secret-shares of the respective plaintexts $[x_i]$ and randomness $[r_i]$. The set of relations to be proven is formalised by structures $\mathcal{E}$ and $\mathcal{G}$. $\mathcal{E}$ is a set of equations $x_j = 0$ ($1 \leq j \leq N$ for some $N \geq n$). $\mathcal{G}$ is an arithmetic circuit to compute values x_j for $j > n$. Specifically, $\mathcal{G}$ consists of gates $x_k = v$, $x_k = x_i + x_j$, $x_k = x_i \cdot v$, and $x_k = x_i \cdot x_j$ (v any constant). For multiplication $x_k = x_i \cdot x_j$, we require $1 \leq j \leq n$: for these encryptions the workers have shared the randomness, which we will need to produce the proof. (Clearly, any set of polynomial relations can be described by such $\mathcal{E}$ and $\mathcal{G}$.)

The first step of the protocol is to evaluate the circuit (lines 3–10) to obtain encryptions $X_{n+1}, \ldots, X_N$. All gates except $x_k = x_i \cdot x_j$ can be evaluated locally; for $x_k = x_i \cdot x_j$, the parties use their additive shares of the plaintext of X_j to obtain shares of X_k, randomised using randomness $[s_k]$. Then, the parties compute announcements for the proofs of correctness of their multiplications

Require: pk/sk ElGamal public/private keys shared between parties $\mathcal{P}_1$, $\mathcal{P}_2$; $X_i = \mathsf{Enc}_{\mathsf{pk}}(x_i; r_i)$ a set of ElGamal encryptions; $\mathcal{G}$ an arithmetic circuit for $x_{n+1}, \ldots, x_N$ with multiplication gates $\mathcal{M} \subset \mathcal{G}$; $\mathcal{E}$ a set of equations $x_k = 0$

Ensure: return proof that equations $\mathcal{E}$ hold for $X_1, \ldots, X_N$ computed according to $\mathcal{G}$

1: **protocol** PolyProve$^{\mathcal{E},\mathcal{G}}(\mathsf{pk}; [\mathsf{pk}]; [\mathsf{sk}]; X_1, \ldots, X_n; [x_1], \ldots, [x_n]; [r_1], \ldots, [r_n])$
2: **parties** $\{\mathcal{P}_1, \mathcal{P}_2\}$ **do**
3: **for all** gates $\in \mathcal{G}$ **do** ▷ evaluate circuit and exchange multiplication proofs
4: **if** $\langle$constant gate $x_k = v\rangle$ **then** $X_k \leftarrow \mathsf{Enc}_{\mathsf{pk}}(v; 0)$
5: **if** $\langle$addition gate $x_k = x_i + x_j\rangle$ **then** $X_k \leftarrow X_i \oplus X_j$
6: **if** $\langle$multiplication gate $x_k = x_i \cdot v\rangle$ **then** $X_k \leftarrow X_i \otimes v$
7: **if** $\langle$multiplication gate $x_k = x_i \cdot x_j,\ 1 \le j \le n\rangle$ **then**
8: $[r_k] \in_r \mathbb{Z}_p$; $[X_k] \leftarrow (X_i \otimes [x_j]) \oplus \mathsf{Enc}_{\mathsf{pk}}(0; [r_k])$
9: $\mathsf{send}([X_k])$; $[X'_k] \leftarrow \mathsf{recv}()$; $X_k \leftarrow [X_k] \oplus [X'_k]$
10: $([a_k]; s_k) \leftarrow \Sigma_{\mathrm{CM}}.\mathsf{ann}(X_i, [X_j], [X_k]; [x_j], [r_j], [r_k])$
11: **parties** $\mathcal{P}_1$ **do**
12: $h \leftarrow \mathcal{H}(\{[a_k]\}_{k\in\mathcal{M}})$; $\mathsf{send}(h)$; $\{[a'_k]\}_{k\in\mathcal{M}} \leftarrow \mathsf{recv}()$; $\mathsf{send}(\{[a_k]\}_{k\in\mathcal{M}})$
13: **parties** $\mathcal{P}_2$ **do**
14: $h \leftarrow \mathsf{recv}()$; $\mathsf{send}(\{[a_k]\}_{k\in\mathcal{M}})$; $\{[a'_k]\}_{k\in\mathcal{M}} \leftarrow \mathsf{recv}()$; **if** $h \neq \mathcal{H}(\{[a'_k]\}_{k\in\mathcal{M}})$ **then** fail
15: **parties** $\{\mathcal{P}_1, \mathcal{P}_2\}$ **do**
16: **for all** mult. gates $x_k = x_i \cdot x_j,\ 1 \le j \le n$ in $\mathcal{M}$ **do** $a_k \leftarrow \Phi_{\mathrm{CM}}.\mathsf{ann}([a_k], [a'_k])$
17: $h_1 \leftarrow \mathcal{H}(\{X_i || X_j || X_k || a_k\}_{\text{gates } x_k = x_i \cdot x_j} || \text{mul})$
18: **for all** mult. gates $x_k = x_i \cdot x_j,\ 1 \le j \le n$ in $\mathcal{M}$ **do**
19: $[r_k] \leftarrow \Sigma_{\mathrm{CM}}.\mathsf{res}(X_i, [X_j], [X_k]; [x_j], [r_j], [r_k]; [a_k]; s_k; h_1)$
20: $\mathsf{send}(\{[r_k]\}_{k\in\mathcal{M}})$; $\{[r'_k]\}_{k\in\mathcal{M}} \leftarrow \mathsf{recv}()$
21: **for all** $k \in \mathcal{M}$ **do**
22: $r_k \leftarrow \Phi_{\mathrm{CM}}.\mathsf{resp}([r_k]; [r'_k])$; **if** $\neg\Sigma_{\mathrm{CM}}.\mathsf{ver}(X_i, X_j, X_k; a_k; h_1; r_k)$ **then** fail
23: **for all** $(x_k = 0) \in \mathcal{E}$ **do** ▷ decrypt eqs + prove
24: $[d_k] \leftarrow \mathsf{Dec}_{[\mathsf{sk}]}(X_k)$; $([A_k]; S_k) \leftarrow \Sigma_{\mathrm{CD}}.\mathsf{ann}(X_k, [d_k], [\mathsf{pk}]; [\mathsf{sk}])$
25: **party** $\mathcal{P}_1$ **do** $h \leftarrow \mathcal{H}(\{[A_k]\}_{k\in\mathcal{E}})$; $\mathsf{send}(h)$; $\{[A'_k]\}_{k\in\mathcal{E}} \leftarrow \mathsf{recv}()$; $\mathsf{send}(\{[A_k]\}_{k\in\mathcal{E}})$
26: **parties** $\mathcal{P}_2$ **do**
27: $h \leftarrow \mathsf{recv}()$; $\mathsf{send}(\{[A_k]\}_{k\in\mathcal{E}})$; $\{[A'_k]\}_{k\in\mathcal{E}} \leftarrow \mathsf{recv}()$; **if** $h \neq \mathcal{H}(\{[A'_k]\}_{k\in\mathcal{E}})$ **then** fail
28: **parties** $\{\mathcal{P}_1, \mathcal{P}_2\}$ **do**
29: **for all** eq. $(x_k = 0) \in \mathcal{E}$ **do** $A_k \leftarrow \Phi_{\mathrm{CD}}.\mathsf{ann}([A_k], [A'_k])$
30: $h_2 \leftarrow \mathcal{H}(\{X_k || g^0 || \mathsf{pk} || A_k\}_{\text{equation } x_k = 0} || \text{dec})$
31: **for all** eq. $(x_k = 0) \in \mathcal{E}$ **do** $[R_k] \leftarrow \Sigma_{\mathrm{CD}}.\mathsf{res}(X_k, [d_k], [\mathsf{pk}]; [\mathsf{sk}]; [A_k]; S_k; h_2)$
32: $\mathsf{send}(\{[R_k]\}_{k\in\mathcal{E}})$; $\{[R'_k]\}_{k\in\mathcal{E}} \leftarrow \mathsf{recv}()$
33: **for all** equations $(x_k = 0) \in \mathcal{E}$ **do**
34: $R_k \leftarrow \Phi_{\mathrm{CD}}.\mathsf{resp}([R_k], [R'_k])$; **if** $\neg\Sigma_{\mathrm{CD}}.\mathsf{ver}(X_k, g^0, \mathsf{pk}; A_k; h_2; R_k)$ **then** fail
35: **return** $(h_1, \{X_k, r_k\}_{k\in\mathcal{M}}, h_2, \{R_k\}_{k\in\mathcal{E}})$ ▷ return mult. and decr. proofs

Fig. 9. PolyProve: Prove polynomial equations over ElGamal ciphertexts

(line 10). They use these announcements to make combined multiplication proofs as described above: they exchange (line 12–14) and combine (line 17) their announcements; compute one overall challenge (line 17); and compute (line 18), exchange (line 20), and combine (line 22) the responses.

Require: $\mathcal{G}$ is an arithmetic circuit for $x_{n+1}, \ldots, x_N$; $\mathcal{M} \subset \mathcal{G}$ are multiplication gates; $\mathcal{E}$ is a set of equations $x_k = 0$
Ensure: all equations in $\mathcal{E}$ hold for $X_1, \ldots, X_N$

1: **function** PolyVer$^{\mathcal{E},\mathcal{G}}(\mathsf{pk}; X_1, \ldots, X_n; \pi)$
2: $(h_1, \{X_k, r_k\}_{k\in\mathcal{M}}, h_2, \{R_k\}_{k\in\mathcal{E}}) \leftarrow \pi$ ▷ unpack
3: **for all** gates $\in \mathcal{G}$ **do** ▷ determine encrypted gates
4: **if** $\langle$constant gate $x_k = v\rangle$ **then** $X_k \leftarrow \mathsf{Enc}_{\mathsf{pk}}(c; 0)$
5: **if** $\langle$addition gate $x_k = x_i + x_j\rangle$ **then** $X_k \leftarrow X_i \oplus X_j$
6: **if** $\langle$multiplication gate $x_k = x_i \cdot v\rangle$ **then** $X_k \leftarrow X_i \otimes v$
7: **for all** multiplications $x_k = x_i \cdot x_j$ in $\mathcal{M}$ **do** $a_k \leftarrow \Sigma_{\mathrm{CM}}.\mathsf{rea}(X_i, X_j, X_k; h_1; r_k)$
8: **for all** equations $x_k = 0$ in $\mathcal{E}$ **do** $A_k \leftarrow \Sigma_{\mathrm{CD}}.\mathsf{rea}(X_k, g^0, \mathsf{pk}; h_2; R_k)$
9: **return** $\mathcal{H}(\{X_i||X_j||X_k||a_k\}_{x_k=x_i\cdot x_j}||\mathrm{mul}) \stackrel{?}{=} h_1 \wedge \mathcal{H}(\{X_k||g^0||\mathsf{pk}||A_k\}_{x_k=0}||\mathrm{dec}) \stackrel{?}{=} h_2$

Fig. 10. PolyVer: Verify polynomial equations over ElGamal ciphertexts

The second step is to prove that, for each equation $x_j = 0$, X_j is an encryption of zero. The parties compute decryption shares $[d_k]$ (line 23) and produce a combined proof that decryption is to zero. These proofs are produced similarly to the multiplication proofs. (Note that the multiplication and decryption proofs cannot use the same challenge: for security, values X_k can be decrypted only after the multiplication proofs have been verified.) The overall proof $\pi = (h_1, \{X_k, r_k\}, h_2, \{R_k\})$ consists of the encrypted products X_k, challenge h_1, h_2, and responses r_k, R_k.

Algorithm PolyVer shown in Figure 10 shows how to check if the proof π produced by PolyProve is correct. Specifically, the algorithm takes as arguments the public key pk, encryptions $X_1, \ldots, X_n$, and proof π as above. First, it computes missing encryptions $\in \{X_{n+1}, \ldots, X_N\}$, i.e., of gates that are not inputs or multiplication results (line 3–6). Then, it computes the announcements for all multiplication (line 7) and decryption (line 8) proofs. The overall proof π is correct when these announcements with their respective statements hash to challenges h_1, h_2 (line 9).

B Security Proof

We now prove Theorem 1. As discussed, security is defined by demanding the existence, for every adversary $\mathcal{A}$ for the real protocol, of an adversary $\mathcal{S}_\mathcal{A}$, called the *simulator*, for the ideal-world execution. This $\mathcal{S}_\mathcal{A}$ should have the property that running the real protocol with $\mathcal{A}$ gives output of the recipient and adversary that are indistinguishable from performing the ideal-world execution with $\mathcal{S}_\mathcal{A}$. In the ideal world, we control the information that $\mathcal{S}_\mathcal{A}$ gets to work with and we control the output for the recipient. Hence, if these are the same in the real world, then privacy, correctness and other modelled security requirements must hold.

We now explicitly construct simulator $\mathcal{S}_\mathcal{A}$ given $\mathcal{A}$. Overall, $\mathcal{S}_\mathcal{A}$ internally runs a copy of $\mathcal{A}$. $\mathcal{S}_\mathcal{A}$ computes messages for $\mathcal{A}$ on behalf of the honest parties

based on the information it gets from the trusted party, and it uses the messages it gets back from $\mathcal{A}$ to provide information to the trusted party to determine the ideal-world output of the recipient. At the end of the protocol, $\mathcal{S}_\mathcal{A}$ outputs whatever the simulated $\mathcal{A}$ would output in a real protocol output. Hence, to show that the ideal- and real-world executions are indistinguishable, we need to show both that the simulated $\mathcal{A}$ gets messages from $\mathcal{S}_\mathcal{A}$ that are indistinguishable from a real protocol execution (implying that its output is indistinguishable as well), and that the information that $\mathcal{S}_\mathcal{A}$ provides to $\mathcal{T}$ leads to an output for the recipient that is the same as in the protocol execution simulated with $\mathcal{A}$.

To prove our theorem, we build two different simulators. One works in the "private" case when at most one worker is passively corrupted, in which case we guarantee privacy. The other works in the "correct" case when either at least two workers are passively corrupted, or some worker is actively corrupted, in which case we do not guarantee privacy but only correctness. Our simulators, like our protocol, are based on [SV15]; we assume familiarity with the proof techniques used there. In particular, we refer to that work for details on the use of the random oracle model and the use of witness extended emulation techniques to extract witnesses from zero-knowledge proofs.

B.1 The Private Case

We now provide our simulator for the "private" case of at most one passively corrupted worker. We will show that, assuming the decisional Diffie-Hellman problem is hard in the group used for ElGamal encryption, the real and ideal protocol executions are computationally indistinguishable. Note that in this case, communication between the ideal world trusted party and simulator is as follows: the simulator sends the corrupted inputs to the trusted party $\mathcal{T}$; and if the recipient $\mathcal{R}$ is corrupted, the simulator gets the result of the computation. Simulation is as follows.

First, simulate step 1 of VerMPC using encryptions X_i of zero for the honest inputters. For the honest inputters, note that we can perform the required proofs of knowledge for X_i without actually knowing the plaintext and randomness by programming the random oracle [SV15]; this will be important later on. For the corrupted inputters, extract their plaintext x_i from the provided proof of knowledge $\pi_{\mathrm{x},i}$ using witness-extended emulation. If inputter i fails to provide a correct opening of hash h_i or a correct proof of knowledge, set $x_i = \perp$. Now, simulate steps 2–4 of VerMPC with respect to $\mathcal{A}$. In step 2, send random values to $\mathcal{P}_1$ or $\mathcal{P}_2$, if corrupted. In step 3, for corrupted inputters run the protocol; if any of them provides shares that are inconsistent with the encryptions X_i, set $x_i = \perp$. For honest inputters, use encryption shares that add up to the values X_i above. Provide corrupted x_i to the trusted party. Simulate step 4 by using the simulator of the underlying multiparty computation protocol, using zero as input for the honest parties.

To simulate step 5, handle encryptions A_i line 16 as in the protocol. (I.e., they represent the output of the computation as performed on zero inputs of the honest parties.) For encryptions R_i, do the same if the recipient is honest.

Otherwise, receive the results r_i of the computation from the trusted party, compute honest $[r_{\mathrm{r},i}]$ from the corrupted shares, and use these sharings to produce the encryption in line 18. Finally, simulate PolyProve. Here, the Σ-protocols can be simulated using the techniques from [SV15]. In particular, note that the simulator does not know the witness for the honest decryption proofs (this being the private key share of the honest worker), but it does know that each encryption X_k should decrypt to zero. Hence, it can compute the decryption shares the honest worker based on the decryption share of the corrupt worker and simulate its proof of correctness. Finally, if $\mathcal{R}$ is corrupted, provide the relevant values for steps 6 and 7.

Proof (Proof of Theorem 1, private case). We now prove for the above simulator $\mathcal{S}_{\mathcal{A}}$, and in case at most one worker is passively corrupted, that

$$\textsc{Real}^{C,A}_{\Pi,\mathcal{A}}(k;\boldsymbol{x}) \approx_c \textsc{Ideal}^{C,A}_{\mathrm{IVERMPC}^{f,\phi},\mathcal{S}_{\mathcal{A}}}(k;\boldsymbol{x}).$$

(Here, $\approx_c$ denotes computational indistinguishability. Below, we use $\approx_s$ for statistical indistinguishability, which of course implies computational indistinguishability.) In our proof, we move from Ideal to Real in steps. Note that correctness of the output by a honest recipient is always guaranteed by construction.

Ideal $\approx_s$ $\mathbf{YAD}_0$ YAD_0 starts by computing a random encryption B of 0. For every encryption X that Ideal computes based on zero inputs of the honest inputters, YAD_0 produces both an encryption X' based on the zero inputs and X'' based on the actual inputs of the honest inputters. (These encryption X are the encryptions X_i of honest parties inputs; A_i of the certificate; R_i of the result if the recipient is honest; and multiplication results X_k from PolyProve.) Then, YAD_0 computes X as $((1 \ominus B) \otimes X') \oplus (B \otimes X'')$. Note that we can indeed use X instead of X' without even knowing its randomness: the encryptions are only used to perform zero-knowledge proofs on, which we can simulate without knowing the plaintext and randomness.

Because in both Ideal and YAD_0, X is a random encryption of the same value, Ideal is clearly statistically indistinguishable from YAD_0.

$\mathbf{YAD}_0$ $\approx_c$ $\mathbf{YAD}_1$ YAD_1 is exactly the same as YAD_0, except that it sets B to be a random encryption of 1. In particular, all encryptions now represent the actual values while the secret sharing computation still uses zero inputs. Note that this does not affect the protocol, e.g., the decryptions in PolyProve are now performed on the outputs of the real computation instead of on the zero-based computation, but we already forced decryption return zero anyway. Clearly, $\mathrm{YAD}_0 \approx_c \mathrm{YAD}_1$ follows directly from semantic security of the ElGamal cryptosystem, i.e., from the decisional Diffie-Hellman problem in the underlying group: if they were not, then executing $\mathrm{YAD}_0/\mathrm{YAD}_1$ on a *given* encryption B would give a distinguisher between random encryptions of zero and one.

$\mathbf{YAD}_1 \approx_s \mathbf{YAD}_2$ YAD_2 is the same as YAD_1, except that it no longer uses random encryption B to compute encryptions as $X = ((1 \ominus B) \otimes X') \oplus (B \otimes X'')$; instead, it directly uses X''. Statistical indistinguishability is clear.

$\mathbf{YAD}_2 \approx_s \mathbf{YAD}_3$ YAD_3 is the same as YAD_3, except that also in the multiparty computation, the actual inputs of the honest parties are used. Statistical indistinguishability follows by the security of the multiparty computation protocols used.

$\mathbf{YAD}_3 \approx_s \mathbf{Real}$ The only remaining difference between YAD_3 and Real is that YAD_3 uses simulated zero-knowledge proofs whereas in Real, honest workers produce zero-knowledge proofs in the regular way. However, as shown in [SV15], this simulation can be done in a way that is statistically indistinguishable by the adversary.

Overall, the above sequence implies Ideal $\approx_c$ Real, as we wanted to show.

B.2 The Correct Case

We now consider the "correct" case in which there are at least t passively corrupted workers, or at least one actively corrupted worker. We build a simulator $\mathcal{S}_{\mathcal{A}}$ that gives statistical indistinguishability between the real-world and ideal-world protocol executions. Recall that the ideal-world simulator communicates with the trusted party as follows. First, it provides the corrupted inputs, after which it obtains the honest inputs. If there are any actively corrupted workers, then they choose the output $\boldsymbol{r}, \boldsymbol{a}$ of the multiparty computation. Otherwise, any corrupted inputter can block the computation (but not change its output). Finally, if the recipient is corrupted then it obtains the computation result. The simulator works as follows.

First, simulate step 1 of VerMPC. The main difficulty is that the simulator needs to provide the inputs of the corrupted parties to the trusted party before it learns the inputs of the honest parties. Conversely, the protocol requires providing the hash of encryptions of the honest inputs right at the beginning. The simulation simulates providing the hash by "lazily" programming the random oracle (cf. [SV15]). Namely, for each honest inputter, generate random hash value h_i, and broadcast it. Then, for each corrupted inputter i, receive hash value h_i. If this hash value has not been queried to the random oracle or the pre-image is not a correct proof of knowledge for an encryption X_i, set $x_i = \perp$. Otherwise, extract plaintext x_i from the extracted proof with witness-extended emulation. Now, send the corrupted inputs x_i to the trusted party and receive the honest x_i. For each honest inputter i, produce encrypted input X_i and proof of knowledge $\pi_{\mathrm{x},i}$, and program the random oracle such that $i||X_i||\pi_{\mathrm{x},i}$ hashes to h_i. Because the adversary cannot guess this pre-image, it has likely not queried the random oracle on it, so it cannot detect this deviation from the protocol. Broadcast the plaintexts and proofs. Finally, for each corrupted inputter i, receive the encryption and proof of knowledge; if they are different from the ones extracted above, set $x_i = \perp$.

From step 2 onwards, the simulator knows the inputs to the computation, so it can simply perform the full protocol with respect to the adversary.

If there are no actively corrupted workers, then the simulator observes whether the corrupted inputters deliver inconsistent sharings of their encrypted inputs, and based on that decides whether to send $\top$ or $\bot$ to the trusted party (cf. line 9 of $\text{IVERMPC}^{f,\phi}$).

If there is at most one corrupted worker, it is actually not directly clear that the simulator can perform the full protocol with respect to the adversary. Namely, it does not know the honest shares of the decryption key but needs these to run lines 23–31 of PolyProve. However, note that in our protocol, the simulator only has to decrypt ciphertexts of which it already knows the plaintext. Namely, ciphertexts are built from the X_i, A_i, and R_i. It has made the X_i for honest parties itself, and it has extracted the plaintexts of corrupted inputters from their proofs of knowledge. For the A_i and R_i, we have assumed that they occur at least once as the right-hand-side of a multiplication in the PolyProve circuit. Hence, the corrupted parties have had to prove knowledge of their contributions in lines 12–22 of PolyProve, from which the simulator can extract their plaintexts. Now, because the simulator knows the plaintext and the decryption keys of the corrupted workers, it can easily simulate the decryption shares of the honest workers. Moreover, it can simulate the proofs of correct decryption without actually knowing the decryption key as in [SV15]. If there are at least two corrupted workers, then the simulator really has all information and it can easily simulate the protocol.

Finally, if the recipient is honest, then the simulator needs to provide the recipient's output $\boldsymbol{r}$ and the certificate $\boldsymbol{a}$ to the trusted party. For this, it sees what values $A_1, \ldots, A_k, \pi, r_1 \ldots, r_m, r_{\mathrm{r},1}, \ldots, r_{\mathrm{r},m}$ the recipient gets as a result of the protocol run with $\mathcal{A}$. If π is a valid proof for result $r_1, \ldots, r_m$, then $A_1, \ldots, A_k$ must occur in proofs of knowledge made by the adversary, from which the simulator can extract $a_1, \ldots, a_k$. Hence, it provides $\boldsymbol{r}, \boldsymbol{a}$ to the trusted party.

Proof (Proof of Theorem 1, correct case). We now prove for the above simulator $\mathcal{S}_{\mathcal{A}}$, and in case there are ≥ 1 actively or ≥ 2 passively corrupted workers, that:

$$\text{Real}^{C,A}_{\Pi,\mathcal{A}}(k;\boldsymbol{x}) \approx_s \text{Ideal}^{C,A}_{\text{IVERMPC}^{f,\phi},\mathcal{S}_{\mathcal{A}}}(k;\boldsymbol{x}).$$

(Here, $\approx_s$ is statistical indistinguishability.) The two distributions consist of the output of the adversary and recipient.

First consider the output of the adversary. There are only two differences in the real protocol execution in Real and the protocol execution that the simulator has performed with $\mathcal{A}$ in Ideal. The first difference is is that in the input phase, the simulator programs the random oracle to be able to provide a hash image h_i without actually having determined the pre-image yet. As in [SV15], this succeeds (in which case the view of the adversary is identical to the real protocol) except with negligible probability. The second difference is that, in case there is only one corrupted worker, the simulator simulates decryption proofs instead of computing them. Also this simulation succeeds except with

negligible probability; and if it does, it gives a view for the adversary that is identical to the real protocol. Hence, the adversary has statistically no way to distinguish the real-world and ideal-world executions.

Now consider the output of the recipient. As described above, the simulator simply runs the protocol, and checks if the adversary supplies a correct proof to decide if the trusted party should provide the computation result to the ideal-world recipient. However, the trusted party passes on these results to the ideal-world recipient only after verifying that ϕ holds, so if ϕ does not hold but the proof nonetheless verifies, then this gives a difference between the real-world and ideal-world outputs. However, in this case, the adversary has managed to produce at least one non-interactive zero-knowledge proof of an incorrect statement. The chance of the adversary succeeding in doing this is only negligible, so with overwhelming probability, ϕ does hold and the ideal-world and real-world outputs of the recipient are in fact the same. This concludes the proof.