Cryptographic Asynchronous Multi-Party Computation with Optimal Resilience
(extended abstract)*

Martin Hirt[1], Jesper Buus Nielsen[2**], and Bartosz Przydatek[1]

[1] Department of Computer Science, ETH Zurich
8092 Zurich, Switzerland
{hirt,przydatek}@inf.ethz.ch
[2] Department of Computer Science, University of Aarhus
DK-8200 Aarhus, Denmark
buus@daimi.au.dk

Abstract. We consider secure multi-party computation in the asynchronous model and present an efficient protocol with optimal resilience. For n parties, up to $t < n/3$ of them being corrupted, and security parameter κ, a circuit with c gates can be securely computed with communication complexity $\mathcal{O}(cn^3\kappa)$ bits. In contrast to all previous asynchronous protocols with optimal resilience, our protocol requires access to an expensive broadcast primitive only $\mathcal{O}(n)$ times — independently of the size c of the circuit. This results in a practical protocol with a very low communication overhead.

One major drawback of a purely asynchronous network is that the inputs of up to t honest parties cannot be considered for the evaluation of the circuit. Waiting for all inputs could take infinitely long when the missing inputs belong to corrupted parties. Our protocol can easily be extended to a hybrid model, in which we have one round of synchronicity at the end of the input stage, but are fully asynchronous afterwards. In this model, our protocol allows to evaluate the circuit on the inputs of every honest party.

1 Introduction

Secure multi-party computation. The goal of secure multi-party computation (MPC) is to allow a set of n players to evaluate an agreed function of their inputs in a secure way, where security means that an adversary corrupting some of the players cannot achieve more than controlling the inputs and outputs of these players. In particular, the adversary does not learn the inputs of the uncorrupted players, and furthermore, she cannot influence the outputs of the uncorrupted players except by selecting the inputs of the corrupted players.

---

* The full version of this paper is available at *Cryptology ePrint Archive* [HNP04].

** Supported by the Danish National Science Research Council grant No. 21-02-0093.

We consider a static active t-adversary who can corrupt up to t of the players and take full control over them. Furthermore, we focus on *asynchronous communication*, i.e., the messages in the network can be delayed for an arbitrary amount of time (but eventually, all messages are delivered). As a worst-case assumption, we give the ability of controlling the delay of messages to the adversary.

Asynchronous communication models real-world networks (like the Internet) much better than synchronous communication. However, it turns out that MPC protocols for asynchronous networks are significantly more involved than their synchronous counterparts. One reason for this is that in an asynchronous network, when a player does not receive an expected message, he cannot distinguish whether the sender is corrupted and did not send the message, or the message was sent but delayed in the network. This implies also that in a fully asynchronous setting it is impossible to consider the inputs of *all* uncorrupted players when evaluating the function. The inputs of up to t (potentially honest) players have to be ignored, because waiting for them could turn out to be endless [Bec54].

History and related work. The MPC problem was first proposed by Yao [Yao82] and solved by Goldreich, Micali, and Wigderson [GMW87] for computationally bounded adversaries and by Ben-Or, Goldwasser, and Wigderson [BGW88] and independently by Chaum, Crépeau, and Damgård [CCD88] for computationally unbounded adversaries. All these protocols considered a synchronous network with a global clock. The first MPC protocol for the asynchronous model (with unconditional security) was proposed by Ben-Or, Canetti, and Goldreich [BCG93]. Extensions and improvements, still in the unconditional model, were proposed in [BKR94,SR00,PSR02]. A great overview of asynchronous MPC with unconditional security is given in [Can95].

The most efficient asynchronous protocols up to date are the ones of Srinathan and Rangan [SR00] and of Prabhu, Srinathan and Rangan [PSR02]. The former protocol requires $\Omega(n^2)$ invocations to the broadcast primitive for every multiplication, which makes the protocol very inefficient when broadcast is realized with some asynchronous broadcast protocol. The latter protocol is rather efficient; it requires $\Omega(n^4\kappa)$ bits of communication per multiplication. However, it tolerates only $t < n/4$ corruptions, which is non-optimal.

Contributions. We present the first asynchronous MPC protocol for the cryptographic model. The protocol is secure with respect to an active adversary corrupting up to $t < n/3$ players; this is optimal in an asynchronous network.

The main achievement of the new protocol is its efficiency: Once the inputs are distributed, the protocol requires $\mathcal{O}(c_M n^3 \kappa)$ bits of communication to evaluate a circuit with c_M multiplication gates and with security parameter κ. This is the same communication complexity that is required by the most efficient known protocol for the synchronous model [CDN01], and improves on the communication complexity of the most efficient optimally-secure asynchronous MPC protocol [SR00] by a factor of $\Omega(n)$. In contrast to both the protocols of [CDN01] and [SR00], our protocol uses broadcast only in a very limited manner: the number of broadcast invocations is independent of the size of the circuit. This nice property is also achieved in [PSR02], but this protocol is non-optimal (it tolerates only $t < n/4$) and requires $\Omega(n)$ times more communication than ours.

In an asynchronous MPC, the agreed function can be evaluated only on a subset of the inputs, i.e., some (potentially honest) player cannot provide their input into the computation. However, the presented protocol can easily be extended to consider the input of each (honest) party, at the cost of one round of synchronization required at the end of the input stage.

2 Preliminaries, notation and tools

2.1 Formal model

We use the model of security of asynchronous protocols from [Can01]. Formally our model for running a protocol will be the hybrid model with a functionality for distributing some initial cryptographic keys between the parties using some function init. The ideal functionality that we wish to realize is given by a circuit Circ, or more precisely a family of circuits. Namely, the functionality allows the adversary to specify a set of at least $n - t$ parties, $W \subseteq [n]$ (where $[n]$ denotes the set $\{1, \ldots, n\}$), which are to supply the inputs to the computation. The circuit to be computed, $\mathsf{Circ} = \mathsf{Circ}(W)$, is then uniquely defined by the subset of parties providing the inputs. The informal proofs in this extended abstract do not require familiarity with specific details of the model in [Can01], and below we only recall the needed specificities.

ASYNCHRONOUS PROTOCOLS. An n-player protocol is a tuple $\pi = (P_1, \ldots, P_n, \mathsf{init})$, where each P_i is a probabilistic interactive Turing machine, and init is an *initialization function*, used for the usual set-up tasks (initialize the players, set up cryptographic keys, etc.). The parties (players) communicate over an *asynchronous* network, in which the delay between sending and delivery of a message is unbounded. More precisely, when a party sends a message, this message is added to the set of messages already sent but not yet delivered, $\mathsf{Msg} = \{(i, j, m)\}$, where (i, j, m) denotes a message m from P_i to P_j. The delivery of the messages is scheduled by the adversary (see below).

We assume that the function to be computed is given as a circuit consisting of input gates augmented by the party to supply the input, linear gates and multiplication gates, and output gates augmented by the party to see the output, all over some ring $\mathcal{M}$.

ADVERSARY. We consider a polynomially bounded adversary, and our constructions are parametrized by a security parameter κ. The adversary controls the delivery of messages and can corrupt up to t parties. A corrupted party is under full control of the adversary, which sees all incoming messages, and determines all outgoing messages. The adversary schedules the delivery of the messages arbitrarily, by picking a message $(i, j, m) \in \mathsf{Msg}$ and delivering it to the recipient. The adversary doesn't see the contents of messages exchanged between honest (i.e., not corrupted) parties, and any message from an honest party to an honest party is *eventually* delivered. In most cases we require that $t < n/3$, but will sometimes consider other thresholds. The set of parties to be corrupted is specified by the adversary *before* the execution of the protocol, i.e., we consider static security.

EXECUTION OF A PROTOCOL. Before the protocol starts, an initialization function init is evaluated on the security parameter 1^κ and on random input $r \in \{0,1\}^*$, to generate a tuple $(\mathrm{sv}_1, \ldots, \mathrm{sv}_n, \mathrm{pv}) = \mathsf{init}(\kappa, r)$ of secret values sv_i and a public value pv. Each party P_i is initialized with $(1^\kappa, \mathrm{sv}_i, \mathrm{pv})$. At the beginning of the protocol execution, every party P_i receives its input value x_i from the environment, and produces some initial messages $(i, \cdot, \cdot)$ which are added to the set Msg. The adversary is given the public value pv, the values (x_j, sv_j) for each corrupted party P_j, and the control over the set Msg. Subsequently the protocol is executed in a sequence of *activations*. In each activation the adversary picks a message $(i, j, m) \in \mathsf{Msg}$ and delivers it to P_j. Upon delivery of a message, party P_j performs some computation based on its current state, updates its state and produces some messages of the form $(j, \cdot, \cdot)$, which are added to the set Msg. In some activation the parties can produce the output to the environment and terminate. The adversary determines the inputs x_i and all messages of corrupted parties. The adversary and the environment can communicate with each other.

SECURITY. The security of a protocol is defined relative to an ideal evaluation of the circuit by requiring that for any adversary attacking the execution of the protocol there exists a simulator which can simulate the attack of the adversary to any environment given only an ideal process for evaluating the circuit. In the ideal process the simulator has very restricted capabilities: It sees the inputs of the corrupted parties. Then it specifies a subset $W \subseteq [n]$ of the parties to be the input providers, under the restriction that $|W| \geq n - t$. The set W is used to pick the circuit $\mathsf{Circ} = \mathsf{Circ}(W)$ to be evaluated. The input gates of Circ are assigned the inputs of the corresponding parties (the adversary specifies the inputs of the corrupted parties), then Circ is evaluated and the outputs of the corrupted parties are shown to the simulator. Given these capabilities the simulator must then simulate to the environment the entire view of an execution of the protocol, including the messages sent and the possible communication between the environment and the adversary.

In the following subsections we briefly describe cryptographic tools needed in our constructions, and introduce a notation for their use in the rest of the paper.

2.2 Homomorphic public-key encryption with threshold decryption

We assume the existence of a semantically secure probabilistic public-key encryption scheme, which also is homomorphic and enables threshold decryption:

ENCRYPTION AND DECRYPTION. For an encryption key e and a decryption key d, let $\mathscr{E}_e : \mathcal{M} \times \mathcal{R} \to \mathcal{C}$ denote the encryption function mapping (plaintext, randomness) pair $(c, r) \in \mathcal{M} \times \mathcal{R}$ to a ciphertext $C \in \mathcal{C}$, and let $\mathscr{D}_d : \mathcal{C} \to \mathcal{M}$ denote the corresponding decryption function. We require that $\mathcal{M}$ is a ring $\mathbb{Z}_M$ for some $M > 1$, and we use $\cdot$ to denote multiplication in $\mathcal{M}$. We often use capital letters to denote an encryption of the corresponding lower-case letter. When keys are understood, we write $\mathscr{E}$ and $\mathscr{D}$ instead of $\mathscr{E}_e$ resp. $\mathscr{D}_d$, and we frequently omit the explicit mention of the randomness in encryption function $\mathscr{E}$.

Homomorphic property. We require that there exist (efficiently computable) binary operations $+$, $*$, and $\oplus$, such that $(\mathcal{M}, +)$, $(\mathcal{R}, *)$, and $(\mathcal{C}, \oplus)$ are algebraic groups, and that $\mathscr{E}_e$ is a group homomorphism, i.e. that

$$\mathscr{E}(a, r_a) \oplus \mathscr{E}(b, r_b) = \mathscr{E}(a + b, r_a * r_b) \ .$$

We use $A \ominus B$ to denote $A \oplus (-B)$, where $-B$ denotes the inverse of B in the group $\mathcal{C}$. For an integer a and $B \in \mathcal{C}$ we use $a \cdot B$ to denote the sum of B with itself a times in $\mathcal{C}$.

Ciphertext re-randomization. For $C \in \mathcal{C}$ and $r \in \mathcal{R}$ we let $\mathscr{R}_e(C, r) = C \oplus \mathscr{E}_e(0, r)$. We use $C' = \mathscr{R}_e(C)$ to denote $C' = \mathscr{R}_e(C, r)$ for uniformly random $r \in \mathcal{R}$. We call $C' = \mathscr{R}_e(C)$ a re-randomization of C. Note that C' is a uniformly random encryption of $\mathscr{D}_d(C)$.

Threshold decryption. We require that there exists a threshold function sharing of $\mathscr{D}_d$ among n parties, i.e., for some construction threshold $1 < t_{\mathsf{D}} \leq n$ there exists a sharing $(d_1, \ldots, d_n)$ of the decryption key d (where d_i is intended for party P_i), such that given decryption shares $c_i = \mathscr{D}_{i,d_i}(C)$ for t_{D} distinct decryption-key shares d_i, it is possible to efficiently compute c such that $c = \mathscr{D}_d(C)$. Furthermore, the encryption scheme should be still semantically secure against chosen plaintext attack when the adversary is given $t_{\mathsf{D}} - 1$ decryption-key shares. Finally, we require that given a ciphertext C, plaintext $c = \mathscr{D}_d(C)$, and a set of $t_{\mathsf{D}} - 1$ decryption-key shares $\{d_i\}$, it is possible to compute *all* n decryption shares $c_j = \mathscr{D}_{j,d_j}(C)$, $j = 1, \ldots, n$. We will always have $t_{\mathsf{D}} = t + 1$. When keys are understood, we write $\mathscr{D}_i(C)$ to denote the function computing decryption share of party P_i for ciphertext C, and $c = \mathscr{D}(C, \{c_i\})$ to denote the process of combining the decryption shares $\{c_i\}$ to a plaintext c.

Robustness. To efficiently protect against cheating servers we require that there exists an efficient two-party zero-knowledge protocol for proving the correctness of a decryption share $c_i = \mathscr{D}_{i,d_i}(C)$ given (e, C, c_i) as instance, and given (i, d_i) and randomness r as witness. We require also that there exists an efficient two-party zero-knowledge protocol for proving the knowledge of a plaintext, given (e, C) as instance and the corresponding plaintext c and randomness r as witness. We require that these protocols communicate $\mathcal{O}(\kappa)$ bits per proof.

2.3 Digital signatures

We assume the existence of a digital signature scheme unforgeable against an adaptive chosen message attack. For a signing key s and a verifying key v, let $\mathsf{Sign}_s : \{0,1\}^* \to \{0,1\}^\kappa$ denote the signing function, and let $\mathsf{Ver}_v : \{0,1\}^* \times \{0,1\}^\kappa \to \{0,1\}$ denote the verifying function, where $\mathsf{Ver}_v(m, \sigma) = 1$ indicates that σ is a valid signature on m. We write $\mathsf{Sign}_i/\mathsf{Ver}_i$ to denote the signing/verification operation of party P_i.

2.4 Threshold signatures

We assume the existence of a *threshold* signature scheme unforgeable against an adaptive chosen message attack. For a signing key s and a verifying key v, let $\mathscr{S}_s : \{0,1\}^* \to \{0,1\}^\kappa$ denote the signing function, and let $\mathscr{V}_v : \{0,1\}^* \times \{0,1\}^\kappa \to \{0,1\}$ denote the verifying function, where $\mathscr{V}_v(m,\sigma) = 1$ indicates that σ is a valid signature on m.

THRESHOLD SIGNING. We require that there exists a threshold function sharing of $\mathscr{S}_s$ among n parties, i.e., for some signing threshold $1 < t_\mathsf{S} \leq n$ there exists a sharing $(s_1, \dots, s_n)$ of the signing key s (where s_i is intended for party P_i), such that given signature shares $\sigma_i = \mathscr{S}_{i,s_i}(m)$ for t_S distinct signing-key shares s_i, it is possible to efficiently compute σ such that $\mathscr{V}_v(m,\sigma) = 1$. Furthermore, the threshold signature scheme should be still unforgeable against adaptive chosen message attack when the adversary is given $t_\mathsf{S} - 1$ signing-key shares. Finally, we require that given a signature σ on m, and $t_\mathsf{S} - 1$ signing-key shares $\{s_i\}$, it is possible to compute *all* n signature shares $\sigma_j = \mathscr{S}_{j,s_j}(m)$, $j = 1, \dots, n$. We will always have $t_\mathsf{S} = n - t$. When keys are understood, we write $\mathscr{S}_i(m)$ to denote the function computing signature share of party P_i for message m, and $\sigma = \mathscr{S}(m, \{\sigma_i\})$ to denote the process of combining the signature shares $\{\sigma_i\}$ to a signature σ.

ROBUST THRESHOLD SIGNING. To efficiently protect against cheating servers we require that there exists an efficient two-party zero-knowledge protocol for proving the correctness of a signature share $\sigma_i = \mathscr{S}_{i,s_i}(m)$, given (v, m, σ_i) as instance and given (i, s_i) as witness. We require that this protocol communicates $\mathcal{O}(\kappa)$ bits per proof.

2.5 Byzantine agreement

We assume the existence of a Byzantine agreement (BA) protocol, i.e., a protocol with the following properties: The input of party P_i is a bit $v_i \in \{0,1\}$ and the output of the BA is a bit $w \in \{0,1\}$. If all honest parties enter the BA, then the BA eventually terminates. Furthermore, if the BA terminates with output w, then some honest party entered the BA with input $v_i = w$. In particular, if all honest parties have the same input $v_i = v$, then the output of the BA is $w = v$.

2.6 Cryptographic assumptions and instantiations of tools

The security of our constructions is based on *decisional composite residuosity assumption* (DCRA) [Pai99]. Alternatively, it could be based also on QRA and strong RSA. We stress, that our constructions are in the plain model. In particular, our constructions *do not* make use of random oracles.

HOMOMORPHIC ENCRYPTION WITH THRESHOLD DECRYPTION. An example of a scheme satisfying all required properties is Paillier's cryptosystem [Pai99] enhanced by threshold decryption as in [FPS00,DJ01]. In this scheme $\mathcal{M} = \mathbb{Z}_N$ for an RSA modulus N. Another example can be based on the QR assumption and the strong RSA assumption. [CDN01].

Digital signatures. As our digital signature scheme we use standard RSA signatures [RSA78].

Threshold signatures. As an example we can use the threshold signature scheme by Shoup [Sho00]. The security of the threshold signature scheme in [Sho00] is based on the assumption that standard RSA signatures are secure. As presented in [Sho00] the zero-knowledge proofs are non-interactive but for the random-oracle model. The protocol can be modified to be secure in the plain (random oracle devoid) model by using interactive proofs (cf. [Nie02]).

Byzantine agreement. In our protocols we employ the efficient Byzantine agreement protocol of Cachin *et al.* [CKS00], which has expected constant round complexity, and expected bit complexity of $\mathcal{O}(n^2\kappa)$. As presented in [CKS00] the security proof of the protocol needs the random-oracle methodology (for the above mentioned threshold signature scheme). This protocol also can be modified to be secure in the plain model [Nie02].

3 The new protocol

Our protocol follows the paradigm of secure multi-party computation based on a threshold homomorphic encryption scheme, as introduced by Franklin and Haber [FH96], and made robust by Cramer, Damgård and Nielsen [CDN01]. However, both protocols use synchrony in an essential manner.

A high-level overview. The protocol proceeds in three stages, an *input stage*, an *evaluation stage*, and a *termination stage*. In the following, we briefly summarize the goal of each stage:

- *Input stage*: Every player provides an encryption of his input to every other player, and the players jointly agree on a subset of players who have correctly provided their inputs.
- *Evaluation stage*: Every player independently evaluates the circuit on a gate-by-gate basis, with help of the other players. The circuit consists of linear gates, multiplication gates, and output gates. The circuit may depend on the selected subset of players that have provided input.
- *Termination stage*: As soon as a player has completed the circuit evaluation, he moves into the termination stage, where the players jointly agree that the circuit evaluation is completed, every player has received (or will eventually receive) the output(s), and hence every player who is still in the evaluation stage can safely abort it.

By having *every* player evaluate the circuit on his own, we bypass the inherent synchronicity problems of the asynchronous model. We denote the player that evaluates the circuit as the *king*, and all other players (who support the king) the *slaves*. Note that every player acts (in parallel) once as king, and n times as slave, once for every king. The kings are not synchronized among each other; it can happen that one king has almost completed the evaluation of the circuit, while another king is still at the very beginning. However, each slave is synchronized with his king. As soon as the first king completes the evaluation and provably reveals all outputs, all other kings (and their slaves) can safely truncate their own evaluation.

In order to achieve robustness, we must require every party to prove (in zero-knowledge) the correctness of essentially every value she provides during the protocol execution. These proofs could easily be constructed in the random-oracle model (by using Fiat-Shamir heuristics), but this would be at the costs of a non-standard model. We therefore follow another approach: A party who is to prove some claim, proves this claim interactively to every other party. The verifying party then certifies that she has correctly verified the claim. Once the prover has collected enough such certificates, she can convince any third party non-interactively of the validity of the claim. Technically, we use threshold signatures for the certificates, which allows the prover to compute one short certificate that proves that t_{S} parties have verified his proof (recall that t_{S} denotes the threshold of the threshold signature scheme, and we set $t_{\mathsf{S}} = n - t$). Formally, we will say that "a party P_i constructs and sends a proof π_i of «*some claim*»", denoted as $\pi_i = \mathsf{proof}(\text{«}\textit{some claim}\text{»})$, when we mean that P_i bilaterally proves the claim in zero-knowledge to every party P_j, who then, upon successful completion of the proof, sends to P_i a signature share $\mathscr{S}_j(\text{«}\textit{some claim}\text{»})$. After obtaining t_{S} correct signature shares[3] $\{\pi_{i,j}\}_{j\in I}$, party P_i computes π_i as $\mathscr{S}(\text{«}\textit{some claim}\text{»}, \{\pi_{i,j}\}_{j\in I})$. Since this construction is standard, we omit the details from the description of the protocols. Naturally, we do include the corresponding subprotocols and their bit complexities in the analysis of the proposed solution (cf. Section 3.8).

Finally we note that we make use of threshold signatures also explicitly, as specified in the descriptions of the protocols. Their use there has similar purpose, namely as certificates for the validity of certain claims.

3.1 Main protocol

The main protocol first invokes the input stage, then the evaluation stage, and finally the termination stage. At the end of the input stage, the circuit Circ is determined by the set of parties that provide input. In the evaluation stage, every party starts one instance of the king protocol, and n instances of the slave protocol — one for every king.

In order to precisely describe the new protocol, we first formally model the circuit to be computed, and then give the invariant that is satisfied during the whole computation. For the clarity of presentation we assume in the following that every party provides at most one input value and that there is only one final output value to be disclosed to all parties (i.e., the final output is to be public). This is without loss of generality for the case with public outputs, and protocols for the general case with multiple input/output values can be derived by straightforward modifications. However, the issue of providing *private* outputs is more involved, and we discuss it in Section 4.1.

[3] The correctness of a signature share is proved to P_i by P_j, using another efficient zero-knowledge protocol (cf. Sect. 2.4).

THE CIRCUIT. We assume that the function to be computed is given as a circuit Circ over the plaintext space $\mathcal{M}$ of the homomorphic encryption scheme in use. The circuit is a set of labeled gates, where each label is a unique bit-string $G \in \{0,1\}^*$. We use $G(\mathsf{Circ})$ to denote the set of all labels of the circuit Circ. In the following we let $v : G(\mathsf{Circ}) \to \mathcal{M} \cup \{\perp\}$ be a map from the labels into the the plaintext space, where $v(G)$ denotes the value of the gate G. Each gate is represented by a tuple $(G, \ldots)$, and can have one of the following types:

input gate: (G), consisting only of its label $G = (P_i, \mathsf{input})$, where $v(G)$ is equal to x_i, the input value provided by player P_i.

linear gate: $(G, \mathsf{linear}, a_0, a_1, G_1, \ldots, a_l, G_l)$, where $l \geq 0$, $a_0, \ldots, a_l \in \mathcal{M}$ are constants, and $v(G) = a_0 + \sum_{j=1}^{l} a_j \cdot v(G_j)$.

multiplication gate: $(G, \mathsf{mul}, G_1, G_2)$, where $v(G) = v(G_1) \cdot v(G_2)$.

output gate: (G, output, G'), where $v(G) = v(G')$ is the output value of the circuit.

CORRECTNESS INVARIANT. Throughout the computation each party P_i maintains a data structure containing the views of each party P_j on the intermediate values in the circuit. More precisely, P_i holds a dictionary Γ_i, which for each party P_j maps labels G to encryptions computed by the King P_j,

$$\Gamma_i : [n] \times G(\mathsf{Circ}) \to \mathcal{C} \cup \{\perp\} ,$$

where initially $\Gamma_i(j, G) =\perp$ for all labels and all $j \in [n]$. If $\Gamma_i(j, G) = C \neq\perp$, then from P_i's point of view gate G was completed by P_j, and C is a ciphertext of the homomorphic encryption scheme of the value $v(G)$. We say that C is the encryption of $v(G)$ *reported* by P_j to P_i, and that P_i has *accepted* C from P_j.

The protocol guarantees, that if an honest party P_i accepts a ciphertext C reported by P_j, then C is an encryption of a correct value for gate G. Moreover, any two honest parties who accept an encryption of any party P_l for a gate G agree on the encryption. Formally, we have the following definition.

Definition 1 (Correctness Invariant). *The following properties hold at any point in the protocol:*

1. (Agreement on circuit) *There exists a set $W \subseteq [n]$ such that $|W| \geq n - t$ and such that for all honest parties having defined the circuit Circ_i it holds that $\mathsf{Circ}_i = \mathsf{Circ}(W)$.*
2. (Agreement on input encryptions) *For all pairs of honest parties P_i, P_j, and all $k, l, m \in W$ it holds that if $\Gamma_i(k, (P_l, \mathsf{input})) \neq\perp$ and $\Gamma_j(m, (P_l, \mathsf{input})) \neq\perp$, then*
$$\Gamma_i(k, (P_l, \mathsf{input})) = \Gamma_j(m, (P_l, \mathsf{input})) := X_l \ .$$
Furthermore, if P_l is honest then $\mathscr{D}(X_l)$ is the initial input x_l of P_l.
3. (Correct gate encryption) *For every honest party P_i, if for any $G \in \mathsf{Circ}$ we have that $\Gamma_i(i, G) = C \neq\perp$, then $\mathscr{D}(C)$ is identical to the value of gate G obtained by decrypting the input encryptions held by P_i and evaluating the circuit on the plaintexts.*

4. (Agreement on encryptions of gates by same king) *For every two honest parties P_i and P_j, for any $l \in [n]$ and any $G \in \mathsf{Circ}$, if $\Gamma_i(l, G) = C \neq \perp$ and $\Gamma_j(l, G) = C' \neq \perp$, then $C = C'$.*

This invariant is propagated from the initial input stage until the output stage is reached. Hence, a threshold decryption of the encrypted output value is guaranteed to yield correct computation results.

THE BASIS OF THE CORRECTNESS INVARIANT. The correctness invariant is established in the input stage, which determines the values of all input gates. Due to the security properties of the input-stage protocol, these values are guaranteed to be *correct* in the sense that the each party providing input knows the actual value hidden in the encryption, and that this value is a valid input to the function to be computed.

3.2 Input stage

The goal of the input stage is to define an encryption of the input of each party. To ensure independence of the inputs, the parties are required to prove plaintext knowledge for their encryptions. In a synchronous network we could simply let the parties broadcast their encryptions. However, in an asynchronous setting with an active adversary we cannot guarantee that each party contributes an input value, since it is impossible to distinguish between an honest slow party and a corrupted party. Therefore a protocol is used which selects $(n-t)$ so-called *input providers.*

First, each party P_i encrypts its input value x_i to obtain a ciphertext $X_i = \mathscr{E}(x_i)$, and constructs a proof $\pi_i = \mathsf{proof}(\text{«}P_i \textit{ knows the plaintext in } X_i\text{»})$ (using bilateral zero-knowledge proofs and threshold signatures), which serves as a certificate that P_i knows the encrypted value, and that X_i is P_i's unique possible input encryption to the circuit. Afterwards P_i distributes (X_i, π) to all parties, and then constructs and distributes another certificate, a *certificate of distribution* cert_i, stating that P_i has distributed (X_i, π_i) to at least $n-t$ parties (recall that $n-t$ is the threshold of the signature scheme).

When a party collects $n-t$ certificates of distributions it knows that at least $n-t$ parties have their certified inputs distributed to at least $n-t$ parties. So, at least $n-t$ parties had its certified input distributed to at least $(n-t)-t \geq t+1$ *honest* parties. So, if all honest parties echo the certified inputs they saw and collect $n-t$ echoes, then all honest parties will end up holding the certified input of the $n-t$ parties which had their certified inputs distributed to at least $t+1$ honest parties. These $n-t$ parties will eventually be the input providers. To determine who they are, n Byzantine agreements are run. The protocol for selecting input providers is given in more detail in Figure 1.

3.3 Computing linear gates

Due to the homomorphic property of encryption linear gates can be computed locally, without interaction. That is, if a party P_i has accepted P_j's encryptions

To define an initial set of inputs, P_i with initial input $x_i \in \mathcal{M}$ proceeds as follows:
Initialize empty sets $A_i, \boldsymbol{\mathcal{A}}_i, B_i, \boldsymbol{\mathcal{B}}_i, C_i$ and execute the following rules concurrently:

DOUBLE DISTRIBUTION:

1. compute $X_i := \mathscr{E}(x_i)$ and $\pi_i := \mathsf{proof}(\text{«}P_i \textit{ knows the plaintext in } X_i\text{»})$.
2. send (X_i, π_i) to all parties.
3. collect $n-t$ signature shares $\{\sigma_j\}$ on «X_i *is* P_i*'s input*»
(i.e., $\sigma_j = \mathscr{S}_j(\text{«}X_i \textit{ is } P_i\textit{'s input}\text{»})$)
4. compute $\mathsf{cert}_i = \mathscr{S}(\text{«}X_i \textit{ is } P_i\textit{'s input}\text{»}, \{\sigma_j\})$; send (X_i, cert_i) to all parties.
5. collect $n-t$ signature shares $\{\sigma'_j\}$ on «*I hold* P_i*'s input*»
6. compute $\mathsf{cert}'_i = \mathscr{S}(\text{«}I \textit{ hold } P_i\textit{'s input}\text{»}, \{\sigma'_j\})$; send cert'_i to all parties.

GRANT CERTIFICATE OF UNIQUENESS:

on the first msg. (X_j, π_j) from P_j, with $\mathscr{V}(\text{«}P_j \textit{ knows the plaintext in } X_j\text{»}, \pi_j)=1$, return $\sigma_i = \mathscr{S}_i(\text{«}X_j \textit{ is } P_j\textit{'s input}\text{»})$ to P_j.

GRANT CERTIFICATE OF DISTRIBUTION:

on the first message (X_j, cert_j) from P_j, with $\mathscr{V}(\text{«}X_j \textit{ is } P_j\textit{'s input}\text{»}, \mathsf{cert}_j) = 1$, add j to A_i, add (X_j, cert_j) to $\boldsymbol{\mathcal{A}}_i$, and return $\sigma'_i := \mathscr{S}_i(\text{«}I \textit{ hold } P_j\textit{'s input}\text{»})$ to P_j.

ECHO CERTIFICATE OF DISTRIBUTION:

on a message cert'_j, where $\mathscr{V}(\text{«}I \textit{ hold } P_j\textit{'s input}\text{»}, \mathsf{cert}'_j) = 1$ and $j \notin C_i$, add j to C_i, and send cert'_j to all parties.

SELECT INPUT PROVIDERS:

When $|C_i| \geq n-t$, stop executing the above rules and proceed as follows:

1. send $(A_i, \boldsymbol{\mathcal{A}}_i)$ to all parties.
2. collect a set $\{(A_j, \boldsymbol{\mathcal{A}}_j)\}_{j\in J}$ of $(n-t)$ incoming, well-formed $(A_j, \boldsymbol{\mathcal{A}}_j)$.
3. let $B_i := \bigcup_{j\in J} A_j$ and $\boldsymbol{\mathcal{B}}_i := \bigcup_{j\in J} \boldsymbol{\mathcal{A}}_j$
4. enter n Byzantine Agreements (BAs) with inputs $v_1, \ldots, v_n \in \{0,1\}$,
where $v_j = 1$ iff $j \in B_i$.
5. let $w_1, \ldots, w_n$ denote the outputs of the BAs,
and let $W = \{j \in \{1, \ldots, n\} | w_j = 1\}$.
6. use W to generate a circuit $\mathsf{Circ} = \mathsf{Circ}(W)$.
7. for each $j \in B_i \cap W$, send $(X_j, \mathsf{cert}_j) \in \boldsymbol{\mathcal{B}}_i$ to all parties.
8. for each $j \in W$ wait to receive (X_j, cert_j) with $\mathscr{V}(\text{«}X_j \textit{ is } P_j\textit{'s input}\text{»}, \mathsf{cert}_j)=1$.
9. for all $j \in W$ and $l \in \{1, \ldots, n\}$, let $\Gamma_i(l, (P_j, \mathsf{input})) = X_j$.

Fig. 1: The input stage code for P_i.

of inputs to a linear gate $(G, \mathsf{linear}, a_0, G_1, a_1, \ldots, G_l, a_l)$, i.e. when $\Gamma_i(j, G_u) \neq \perp$, for $u = 1 \ldots l$, then P_i computes locally $\Gamma_i(j, G) := A_0 \oplus \left(\bigoplus_{u=1}^{l}(a_j \cdot \Gamma_i(j, G_u))\right)$, where A_0 is a *"dummy"* encryption of a_0, computed using a publicly-known, fixed random string.

Wait until input stage is completed, resulting in a circuit Circ *and an initialized dictionary* Γ_k. *Then concurrently execute for each linear, multiplication, or output gate:*

LINEAR GATE $(G, \mathsf{linear}, a_0, a_1, G_1, \ldots, a_l, G_l)$:

1. wait until $\Gamma_k(k, G_u) \neq \perp$, for all $u = 1 \ldots l$.
2. compute $\Gamma_k(k, G) := A_0 \oplus \left(\bigoplus_{u=1}^{l} (a_u \cdot \Gamma_k(k, G_u)) \right)$.

MULTIPLICATION GATE $(G, \mathsf{mul}, G_1, G_2)$:

1. wait until $\Gamma_k(k, G_1) = C_1 \neq \perp$ and $\Gamma_k(k, G_2) = C_2 \neq \perp$
2. generate a randomizer (R, U, cert) for G, and send it to all parties:
   (a) collect a set $S_G := \{(R_i, U_i, \sigma_i, \pi_i)\}_{i \in I_G}$, with $|I_G| \geq t + 1$, where $\sigma_i = \mathsf{Sign}_i(«(R_i, U_i) : \textit{part of } P_k\textit{'s randomizer for } G»)$, and $\pi_i = \mathsf{proof}_i(«P_i \textit{ knows } r_i \textit{ in } R_i\textit{, and } U_i \textit{ is a randomization of } r_i C_1»)$
   (b) send S_G to all parties
   (c) compute $R := \bigoplus_{i \in I_G} R_i$, and $U := \bigoplus_{i \in I_G} U_i$
   (d) collect a set $\mathsf{cert} := \{\mathsf{cert}_i\}_{i \in I'_G}$, with $|I'_G| \geq t_\mathsf{S}$, where $\mathsf{cert}_i = \mathscr{S}_i(«(R, U) : P_k\textit{'s randomizer for } G»)$
3. collect a set $V_G = \{(z_i, \phi_i)\}_{i \in I''_G}$, with $|I''_G| \geq t_\mathsf{D}$, where each z_i is a decryption share of P_i for $Z = C_2 \oplus R$, and ϕ_i is the corresponding validity proof
4. send V_G to all parties
5. decrypt $z := \mathscr{D}(Z, V_G)$ and compute $\Gamma_k(k, G) := (z \cdot C_1) \ominus U$

OUTPUT GATE (G, output, G'):

1. wait until $\Gamma_k(k, G') = C \neq \perp$
2. collect a set $\{(c_i, \varpi_i)\}$ of t_D decryption shares for C, with corresponding validity proofs ϖ_i
3. compute and output $c := \mathscr{D}(C, \{c_i\})$; mark G as $\mathsf{decrypted}$

Fig. 2: The code for king P_k for evaluating the circuit.

3.4 Computing multiplication gates

Computation of multiplication gates is more involved. Each king P_k leads the computation of the encrypted product in *his* copy of the circuit, that is, given a gate $(G, \mathsf{mul}, G_1, G_2)$ such that $\Gamma_k(k, G) = \perp$, $\Gamma_k(k, G_1) = C_1$, and $\Gamma_k(k, G_2) = C_2$, with $C_1, C_2 \neq \perp$, the players proceed as follows. Let c_1, c_2 denote the values hidden in the ciphertexts C_1, C_2, respectively. First a randomizer (R, U, cert) is generated, where R is a threshold encryption of a random element $r \in \mathcal{M}$ (unknown to the parties and the adversary), $U = \mathscr{R}(rC_1)$, i.e., U is a random threshold encryption of rc_1, and cert is a certificate of the encryptions' correctness. Then P_k sends the randomizer to all parties, and waits until the parties answer with decryption shares of the ciphertext $Z = C_2 \oplus R$, which is an encryption of $z = c_2 + r$. Once sufficiently many (i.e., at least t_D) decryption shares arrive, P_k sends them to all parties, which allows each P_i to decrypt z, and compute an encryption of the product $c_1 c_2$, using the homomorphic property of

Wait until input stage is completed, resulting in a circuit Circ *and an initialized dictionary Γ_i. Then concurrently execute the following for each linear, multiplication, or output gate:*

LINEAR GATE $(G, \mathsf{linear}, a_0, a_1, G_1, \ldots, a_l, G_l)$ (only for $i \neq k$):

1. wait until $\Gamma_i(k, G_u) \neq \perp$, for all $u = 1 \ldots l$.
2. compute $\Gamma_i(k, G) := A_0 \oplus \left(\bigoplus_{u=1}^{l} (a_u \cdot \Gamma_i(k, G_u)) \right)$.

MULTIPLICATION GATE $(G, \mathsf{mul}, G_1, G_2)$:

1. wait until $\Gamma_i(k, G_1) = C_1 \neq \perp$ and $\Gamma_i(k, G_2) = C_2 \neq \perp$
2. help to generate a randomizer (R, U, cert) for G:
   (a) compute $R_i = \mathscr{E}(r_i)$ and $U_i = \mathscr{R}(r_i C_1)$ for a randomly picked $r_i \in \mathcal{M}$
   compute $\sigma_i := \mathsf{Sign}_i(«(R_i, U_i)$: *part of P_k's randomizer for G*»)
   construct a proof
   $\pi_i := \mathsf{proof}_i(«P_i$ *knows r_i in R_i, and U_i is a randomization of $r_i C_1$*»)
   (b) send $(R_i, U_i, \sigma_i, \pi_i)$ to king P_k
   (c) wait until received from P_k set
   $S_G := \{(R_l, U_l, \sigma_l, \pi_l)\}_{l \in I_G}$, with $|I_G| \geq t + 1$
   (d) compute $R := \bigoplus_{i \in I_G} R_i$, and $U := \bigoplus_{i \in I_G} U_i$
   compute $\mathsf{cert}_i = \mathscr{S}_i(«(R, U)$: *P_k's randomizer for G*»)
   (e) send cert_i to king P_k
3. wait until received (R, U, cert) from P_k,
   with $\mathscr{V}(«(R, U)$: *P_k's randomizer for G*», $\mathsf{cert}) = 1$
4. compute z_i, P_i's decryption share for $Z = C_2 \oplus R$, and $\phi_i = \mathsf{proof}(«z_i$ *is valid*»);
   send (z_i, ϕ_i) to P_k
5. wait until received V_G from P_k, with $|V_G| \geq t + 1$
6. decrypt $z := \mathscr{D}(Z, V_G)$ and compute $\Gamma_i(k, G) := (z \cdot C_1) \ominus U$

OUTPUT GATE (G, output, G'):

1. wait until $\Gamma_i(k, G') = C \neq \perp$
2. compute a decryption share $c_i := \mathscr{D}_i(C)$ and a proof $\varpi_i = \mathsf{proof}(«c_i$ *is valid*»);
   send (c_i, ϖ_i) to P_k

Fig. 3: The code for slave P_i helping king P_k to evaluate the circuit.

the encryption, and the fact that $c_1 c_2 = (c_2 + r)c_1 - rc_1$. That is, P_i computes $\Gamma_i(k, G) := (z \cdot C_1) \ominus U$.

3.5 Output stage

When P_i notices that the computation of an output gate (G, output, G') is completed by some king P_k (i.e. $\Gamma_i(k, G) = C \neq \perp$), but the gate has not been decrypted so far, then P_i sends a decryption share c_i of C to P_k along with a proof that the decryption share is correct. Then P_k collects enough decryption shares, and computes the value of the output gate.

During the protocol each party executes concurrently the following rules:

RULE 1:
1. wait until the output gate $G \in G(\mathcal{C})$ is marked decrypted
2. vote by sending the value of the gate to all parties

RULE 2:
1. wait until receiving $t+1$ identical votes for the value of the output gate
2. adopt the value receiving $t+1$ votes
3. mark the output gate $G \in G(\mathcal{C})$ as decrypted

RULE 3:
1. wait until receiving $n-t$ identical votes for the value of the output gate
2. terminate

Fig. 4: The code for terminating P_i.

3.6 Termination

As described above each king P_k will eventually learn the value of the output gate. This however requires that each slave P_i keeps running after king P_k learned the output values. To allow to also terminate the slaves, the parties execute a termination protocol. When king P_k learns the output of the circuit it outputs it and echos the result to all parties as its *vote* for the output (and does not yet terminate slave P_k). Since all honest parties compute identical outputs and there are at most t corrupted parties, if a party receives $t+1$ identical votes for some output value it can safely adopt this value as its own output, terminate its own king, and then echo the adopted output value. When a party receives $(n-t)$ identical votes for the output value it terminates the protocol. This is essentially a Bracha broadcast of the output value and allows all parties to eventually terminate.

3.7 Security analysis

Our protocol can be proved secure in the model described in Section 2. A formal proof that the protocol can be simulated can be given along the lines of the proof in [CDN01], using the following helping lemmas.

Lemma 1 (The correctness invariant). *The Properties 1, 2, 3 and 4 of Definition 1 hold at any point in the protocol if there are at most $t < n/3$ corrupted parties.*

Lemma 2 (Termination). *If all honest parties start running the protocol and there are at most $t < n/3$ corrupted parties, then all honest parties will eventually terminate the protocol.*

The proofs of the lemmas are given in the full version of the paper [HNP04]. Here we discuss how the lemmas allow to give a proof along the lines of [CDN01].

By property 1 a set of at least $n - t$ parties have their inputs considered, as required by the model in Section 2. Furthermore, by Property 3, the output v_i obtained by P_i when decrypting the output ciphertext in Step 3 in OUTPUT GATE in Fig. 2 will be correctly defined from the plaintexts of the input ciphertexts held by P_i. Since by Property 2 the parties agree on the input ciphertexts, all honest parties P_i will agree on the output v_i in Step 3 in OUTPUT GATE. This clearly implies that all honest parties terminate the protocol in Fig. 4 with the output being the common value v, as no other value can get $t + 1$ votes when there are at most t corrupted parties. Since v is the result of evaluating the circuit on the plaintexts of the input ciphertexts and, by Property 2, the input ciphertext X_l of honest party P_l contains the correct input x_l, the result v can indeed be obtained by restricting the set of input providers to a set of size at least $n - t$ and then changing only the inputs of the corrupted parties.

The privacy of the protocol (formally defined by the simulator only being given the inputs of the corrupted parties in the simulation) mainly follows from the fact that all inputs are encrypted using a semantic secure encryption scheme and that all proofs are zero-knowledge. So, the only knowledge leaked about the inputs of the honest parties is through decryptions of ciphertexts.

The decryptions take place only in Step 4 in MULTIPLICATION in Fig. 3 and in Step 2 in OUTPUT GATE in Fig. 3. By the correctness of the protocol the knowledge leaked in Step 2 in OUTPUT GATE is the result of the computation, which is allowed to leak by the model. So it remains to argue that no knowledge is leaked in Step 4 in MULTIPLICATION. To see this, observe that the value revealed by the decryption in Step 4 is $z = c_2 + \sum_{i \in I_G} r_i$, which holds the potential of leaking knowledge about c_2 (which is potentially to be kept secret). Since each term r_i from an honest party is chosen uniformly at random and all r_i are chosen independently (this is the purpose of having all parties, in particular the corrupted parties, prove plaintext knowledge of their r_i in Step 2(a)), it is sufficient to show that each revealed value $z = c_2 + \sum_{i \in I_G} r_i$ contains at least one honest value r_i *which did not enter another revealed value.*

Observe first of all that since $|I_G| \geq t+1$, at least one r_i came from an honest party. Observe then that each of the randomizers r_i are associated uniquely to one (P_k, G) by the signature σ_i (issued in Step 2(a) and checked in Step 2(c)). Therefore r_i only enters values $z = c_2 + \sum_{i \in I_G} r_i$ leaked in decryptions in Step 4 in MULTIPLICATION for the specific (P_k, G) in consideration. It is therefore sufficient to show that for each (P_k, G) there is only one value $z = c_2 + \sum_{i \in I_G} r_i$ for which knowledge is leaked. This follows from the uniqueness guaranteed by the threshold $t_S = n - t$ of the threshold signatures. More precisely, assume that for each king P_k and each gate G at most one value of the form «$\cdot$: P_k*'s randomizer for* G» is signed. I.e. there exists at most one value (R, U) for which there exists cert such that $\mathscr{V}($«(R, U) : P_k*'s randomizer for* G»$, \mathsf{cert}) = 1$ (this can be seen to be necessary for Property 4 to hold and thus follows from Lemma 1). Since the honest parties agree on the gate encryptions of P_k (by Property 4), this implies that there is at most one value $Z = C_2 \oplus R$ for which honest parties issue decryption shares in Step 4 for a given choice of (P_k, G).

Therefore each value $z = c_2 + \sum_{i \in I_G} r_i$ on which knowledge is leaked through decryption shares from honest parties, at least one r_i came from an honest party and did not enter another value on which knowledge was leaked, as desired.

3.8 Efficiency analysis

In this section we consider the communication complexity of the protocol. We omit computational complexity from the analysis, since it is clearly polynomial, and the bottleneck of distributed computing is in the communication overhead. For completeness, we consider the case where each party can have more than one input, and we denote by c_I the total number of input gates. For clarity we use $K = n$ to denote the number of kings and $S = n$ to denote the number of slaves.

In the protocol in Fig. 1, when each party has more than one input, X_i will simply be the vector of input encryptions. Assuming that all encryptions, all signature shares, all signatures and all pairwise proofs use communication $\mathcal{O}(\kappa)$ and that the communication complexity of a Byzantine agreement is $\mathcal{O}(n^2\kappa)$, it can be seen using simple counting that the communication complexity of the protocol in Fig. 1 is $\mathcal{O}(c_I n^2 \kappa + n^3 \kappa)$.

In king's protocol (Fig. 2) the only values sent are the sets S_G and V_G. These sets have size $\mathcal{O}(n\kappa)$ and are sent to all S slaves. This gives a communication complexity of $\mathcal{O}(Sn\kappa)$ each time a set is sent, or a total communication complexity of $\mathcal{O}((c_M + c_O)Sn\kappa)$ for running the protocol in Fig. 2, where c_M is the number of multiplication gates and c_O is the number of output gates. This is done by all K kings, yielding a total communication complexity of $\mathcal{O}((c_M + c_O)KSn\kappa)$.

In slave's protocol (Fig. 3) the sending of the values in Steps 2(b), 2(e) and 4 of MULTIPLICATION GATE, and Step 2 of OUTPUT GATE all use $\mathcal{O}(\kappa)$ bits of communication. Moreover, the constructions of the proofs in Steps 2(a) and 4 of MULTIPLICATION GATE, and in Step 2 of OUTPUT GATE takes $\mathcal{O}(n\kappa)$ bits of communication, and thus are the dominating instructions. Each construction of a proof is done at most once for each gate for each king being helped. This yields a total of $\mathcal{O}((c_M + c_O)Kn\kappa)$ for running the slave's protocol. Since the protocol is run by all S slaves, this yields a total communication complexity of $\mathcal{O}((c_M + c_O)KSn\kappa)$.

It is easy to verify that the total communication complexity of the protocol in Fig. 4 is $\mathcal{O}(c_O n^2 \kappa)$.

Summing the above terms we get a communication complexity of $\mathcal{O}((c_I + c_O)n^2\kappa + (c_M + c_O)KSn\kappa + n^3\kappa)$. Using $K = S = n$ and assuming that $c_O \geq 1$, this is $\mathcal{O}(c_I n^2 \kappa + (c_M + c_O)n^3\kappa)$, as claimed.

4 Extensions and applications

4.1 Computing functions with private outputs

The description of the new protocol in Section 3 only considers public outputs, i.e., every party learns the output(s) of the circuit. In the following, we present an extension that allows for outputs that are delivered only to an authorized party, say P_j.

The intuition of the protocol is that the decryption shares are not sent to the king, but rather directly to P_j. Every decryption share must go along with a proof of validity. This proof must not be interactive with P_j (the parties cannot wait for messages of P_j), and the proof must not be given to other parties (this would violate the privacy of the output protocol). Therefore, we have every slave P_i blind his decryption share c_i with a random value r_i, i.e., $c_i' = c_i + r_i$, encrypt r_i with randomness ρ_i, i.e., $R_i = \mathscr{E}(r_i, \rho_i)$, and prove interactively towards every auxiliary player P_j knowledge of r_i such that r_i encrypts to R_i and $c_i' - r_i$ is a valid decryption share. Upon accepting the proof, every auxiliary player hands a signature share for «(c_i', R_i) *is a good decryption share for slave* P_i» to P_i, who then sends c_i', r_i, ρ_i to P_j. Given this information from at least $n-t$ players, P_j picks the valid decryption shares and decrypts his private output.

We note that a similar technique has been recently used by Schoenmakers and Tuyls [ST04].

4.2 A hybrid model: asynchronous network with few synchronization rounds

A fully asynchronous MPC protocol inherently cannot consider the input of every honest party; once $n-t$ inputs are ready, the protocol must start. This is a serious drawback which makes the fully asynchronous model unusable for many real-world applications. We show that with a single round of synchronization, we can consider the input of *every* honest party. This model seems very reasonable in real-world; the parties would wait for other parties to have their input ready, and if not, use other means of communication (email, phone, fax, etc) to synchronize. However, the MPC protocol itself should run asynchronously to comply with the properties of existing networks, namely that the delay of messages is hard to predict. Note that asynchronous protocols can be looked as "best effort" protocols where the progress in the protocol is as fast as possible with the available network, in contrast to synchronous protocols whose progress is limited by the assumed worst-case delay of the network.

The necessary changes in the input protocol (cf. Fig. 1) are minimal: Every player P_i moves to the last stage (Select Input Providers) only when either $|C_i| = n$, or the synchronization round elapsed.

4.3 Non-robust computations

In the proposed protocol, robustness is guaranteed by having each party act as king, who evaluates the whole circuit on his own (with help of his slaves). This means that the computation and communication overhead for achieving robustness is a factor of n.

Goldwasser and Lindell have proposed a model for secure MPC in which output delivery is not guaranteed [GL02], unless some a priori specified party is honest. In this model, we can improve the communication complexity of our protocol by a factor of n, simply be letting this designated party act as king, and all other parties act as his slaves. We stress that the protocol still guarantees privacy and correctness of the computation, but termination (with output delivery) can only be guaranteed when the king is honest. This simplified protocol achieves an overall communication complexity of $\mathcal{O}(cn^2\kappa)$ for a circuit of size c and a security parameter κ.

5 Conclusions

We have proposed a secure multi-party computation protocol which substantially puts forward both theory and practice in this field. From a theoretical point of view, the protocol is optimally resilient, fully asynchronous, and has an asymptotically lower communication complexity than any previous asynchronous MPC protocol. Indeed, the protocol is as efficient as the most efficient known protocol for synchronous communication. Furthermore, the protocol requires very few invocations of the broadcast primitive (independent of the size of the computed circuit).

From a practical point of view, the new protocol is designed for real-world networks with unknown message delay, allows every party to provide his input under a very reasonable assumption (one round of synchronization), and achieves best-possible resilience against cheating (up to a third of the parties may misbehave). Furthermore, the protocol is very efficient, the constant communication overhead is minimal. The effective computation of the circuit takes less than $10n^3\kappa$ bits of communication per multiplication, which makes the protocol applicable for reasonably sized circuits among small sets of parties. The key set-up (for the threshold decryption and threshold signatures) is more communication-intensive; however, this can be performed long in advance.

Acknowledgements

We would like to thank anonymous referees for helpful comments.

References

[BCG93] Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In *Proc. 25th ACM STOC*, pages 52–61, 1993.

[Bec54] Samuel Beckett. *Waiting for Godot*. New York: Grove Press, 1954.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proc. 20th ACM STOC*, pages 1–10, 1988.

[BKR94] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience. In *Proc. 13th ACM PODC*, pages 183–192, 1994.

[Can95] Ran Canetti. *Studies in Secure Multiparty Computation and Applications*. PhD thesis, Weizmann Institute of Science, Rehovot 76100, Israel, June 1995.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proc. 42nd IEEE FOCS*, pages 136–145, 2001.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *Proc. 20th ACM STOC*, pages 11–19, 1988.

[CDN01] Ronald Cramer, Ivan Damgård, and Jesper B. Nielsen. Multiparty computation from threshold homomorphic encryption. In *Proc. EUROCRYPT '01*, pages 280–300, 2001.

[CKS00] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in Constantinopole: Practical asynchronous Byzantine agreement using cryptography. In *Proc. 19th ACM PODC*, pages 123–132, 2000.

[DJ01] Ivan Damgård and Mads Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In *Proc. 4th PKC*, pages 110–136, 2001.

[FH96] Matt Franklin and Stuart Haber. Joint encryption and message-efficient secure computation. *Journal of Cryptology*, 9(4):217–232, 1996.

[FPS00] Pierre-Alain Fouque, Gouillaume Poupard, and Jacques Stern. Sharing decryption in the context of voting or lotteries. In *Proc. Financial Cryptography '00*, 2000.

[GL02] S. Goldwasser and Y. Lindell. Secure computation without agreement. In *DISC '02*, pages 17–32, 2002.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game — a completeness theorem for protocols with honest majority. In *Proc. 19th ACM STOC*, pages 218–229, 1987.

[HNP04] Martin Hirt, Jesper Buus Nielsen, and Bartosz Przydatek. Cryptographic asynchronous multi-party computation with optimal resilience, 2004. *Cryptology ePrint Archive*, eprint.iacr.org, report no. 2004/368.

[Nie02] Jesper B. Nielsen. A threshold pseudorandom function construction and its applications. In *Proc. CRYPTO '02*, pages 401–416, 2002.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proc. EUROCRYPT '99*, pages 223–238, 1999.

[PSR02] B. Prabhu, K. Srinathan, and C. Pandu Rangan. Asynchronous unconditionally secure computation: An efficiency improvement. In *Proc. Indocrypt 2002*, pages 93–107, 2002.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[Sho00] Victor Shoup. Practical threshold signatures. In *Proc. EUROCRYPT '00*, pages 207–220, 2000.

[SR00] K. Srinathan and C. Pandu Rangan. Efficient asynchronous secure multiparty distributed computation. In *Proc. Indocrypt 2000*, pages 117–129, 2000.

[ST04] Berry Schoenmakers and Pim Tuyls. Practical two-party computation based on the conditional gate. In *Proc. ASIACRYPT '04*, pages 119–136, 2004.

[Yao82] Andrew C. Yao. Protocols for secure computations. In *Proc. 23rd IEEE FOCS*, pages 160–164, 1982.