Asynchronous Multi-Party Computation With Quadratic Communication

Martin Hirt[1], Jesper Buus Nielsen[2], and Bartosz Przydatek[3⋆]

[1] Dept. of Computer Science, ETH Zurich, Switzerland
[2] Dept. of Computer Science, University of Aarhus, Denmark
[3] Google Switzerland, Zurich, Switzerland

Abstract. We present an efficient protocol for secure multi-party computation in the asynchronous model 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 $\mathcal{O}(cn^2\kappa)$ bits, which improves on the previously known solutions by a factor of $\Omega(n)$. The construction of the protocol follows the approach introduced by Franklin and Haber (Crypto'93), based on a public-key encryption scheme with threshold decryption. To achieve the quadratic complexity, we employ several techniques, including circuit randomization due to Beaver (Crypto'91), and an abstraction of *certificates*, which can be of independent interest.

1 Introduction

Secure multi-party computation. Secure multi-party computation (MPC) allows a set of n parties (players) to evaluate an agreed function of their inputs in a secure way, where security means that an adversary corrupting some of the parties, cannot achieve more than controlling their inputs and outputs. In particular, the adversary does not learn the inputs of the uncorrupted parties, and she cannot influence the outputs of the uncorrupted parties, except by selecting the inputs of the corrupted players. 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 a player in an asynchronous network waiting for a message 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 — inputs of up to t (potentially honest) players have to be ignored.

History and related work. The MPC problem was first proposed by Yao [26] and solved by Goldreich, Micali, and Wigderson [18] for computationally bounded adversaries and

⋆ Work done in part at ETH Zurich.

L. Aceto et al. (Eds.): ICALP 2008, Part II, LNCS 5126, pp. 473–485, 2008.

by Ben-Or, Goldwasser, and Wigderson [5] and independently by Chaum, Crépeau, and Damgård [12] 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 [4]. Extensions and improvements, still in the unconditional model, were proposed in [6, 24]. A great overview of asynchronous MPC with unconditional security is given in [8]. The most efficient asynchronous protocol up to date [19] communicates $\mathcal{O}(n^3\kappa)$ bits per multiplication gate, where κ is a security parameter.

Contributions. We present an asynchronous MPC protocol, cryptographically secure with respect to an active adversary corrupting up to $t < n/3$ players (this is optimal in an asynchronous network). Once the inputs are distributed, the protocol requires $\mathcal{O}(c_M n^2 \kappa)$ bits of communication to evaluate a circuit with c_M multiplication gates and with security parameter κ. This improves on the communication complexity of the most efficient optimally-secure asynchronous MPC protocol by a factor of $\Omega(n)$. The new protocol, similarly as [19], uses the approach based on threshold encryption [17, 13], but introduces several modifications, which result in both conceptual simplification and improved efficiency. In particular, we use a notion of *certificates*, which greatly simplify the description of the protocol on an abstract level.

2 Formal model and preliminaries

Notation. We use n to denote the number of players (i.e., parties) participating in the MPC protocol, we use $P_1, \ldots, P_n$ to denote the players, and we use $\mathcal{P}$ to denote the set of all players. For an integer $m > 0$ we write $[m]$ to denote the set $\{1, \ldots, m\}$. Our constructions are parametrized by a security parameter κ.

Communication model. We consider an *asynchronous* communication network, with point-to-point secure channels, but without guaranteed delivery of messages. 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, like initialization, setting up cryptographic keys, etc. The players communicate over a network in which the delay between sending and delivery of a message is unbounded. We measure the communication complexity by the worst case number of bits sent by the honest parties.

Security model. We use the model of asynchronous protocols proposed by Canetti [9]. Formally our model for running a protocol is a hybrid model with a functionality init for distributing initial cryptographic keys among the parties. We consider a poly-time adversary, which can corrupt up to $t < n/3$ parties *before* the execution of the protocol, i.e., we consider a static adversary, and corrupted parties are under full control of the adversary. The adversary schedules the delivery of the messages arbitrarily, except that it must eventually deliver all message sent be honest parties.

The security of a protocol is defined relative to an ideal evaluation of the circuit: for any adversary attacking the protocol must exist a simulator which simulates the attack of the adversary to any environment, given only an ideal process for evaluating the circuit. The simulator has very restricted capabilities: It sees the inputs of the corrupted parties.

Then it picks a subset $\mathcal{W} \subseteq [n]$ of the parties to be the input providers, s.t. $|\mathcal{W}| \geq n - t$. The adversary determines the inputs of the corrupted parties. The input gates of Circ belonging to the parties from $\mathcal{W}$ are assigned the inputs of the corresponding parties, and the remaining input gates are assigned default values. Then Circ is evaluated and the outputs of the corrupted parties are shown to the simulator, which must then simulate the entire view of an execution of the protocol.

2.1 Cryptographic primitives and protocols

In the proposed MPC protocols we employ a number of standard primitives and sub-protocols. We introduce the required notation and tools with their essential properties, and then we point to the literature to example implementations.

Homomorphic encryption with threshold decryption. We assume the existence of a semantically secure public-key encryption scheme, which additionally is homomorphic and enables threshold decryption, as specified below.

Encryption and decryption. For an encryption key e and a decryption key d, let $\mathscr{E}_e : \mathbb{M} \times \mathbb{R} \to \mathbb{C}$ denote the encryption function mapping a plaintext $x \in \mathbb{M}$ and a randomness $r \in \mathbb{R}$ to a ciphertext $X \in \mathbb{C}$, and let $\mathscr{D}_d : \mathbb{C} \to \mathbb{M}$ denote the corresponding decryption function, where $\mathbb{M}, \mathbb{R}, \mathbb{C}$ are algebraic structures, as specified below. We require that $\mathbb{M}$ is a ring $\mathbb{Z}_M$ for some $M > 1$, and we use "$\cdot$" to denote multiplication in $\mathbb{M}$. We often use capital letters to denote encryptions of the plaintexts denoted by the corresponding lower-case letters. When keys are understood, we write $\mathscr{E}$, $\mathscr{D}$ instead of $\mathscr{E}_e$, $\mathscr{D}_d$, and we often omit the explicit mentioning of the randomness in the encryption function $\mathscr{E}$.

Homomorphic property. We require that there exist (efficiently computable) binary operations $+$, $*$, $\oplus$, such that $(\mathbb{M}, +)$, $(\mathbb{R}, *)$, $(\mathbb{C}, \oplus)$ are algebraic groups, and that $\mathscr{E}_e$ is a group homomorphism, i.e. $\mathscr{E}(a, r_a) \oplus \mathscr{E}(b, r_b) = \mathscr{E}(a + b, r_a * r_b)$. We use $A \ominus B$ to denote $A \oplus (-B)$, where $-B$ denotes the inverse of B in the group $\mathbb{C}$. For an integer a and $B \in \mathbb{C}$ we use $a \star B$ to denote the sum of B with itself a times in $\mathbb{C}$.

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

Threshold decryption. We require a threshold function sharing of decryption $\mathscr{D}_d$ among n parties, i.e. that for a construction threshold $t_{\mathsf{D}} = t+1$, there is a sharing $(d_1, \ldots, d_n)$ of the decryption key d (where d_i is intended for party P_i), satisfying the following conditions. Given the decryption shares $x_i = \mathscr{D}_{i,d_i}(X)$ for t_{D} distinct decryption-key shares d_i, it is possible to efficiently compute x such that $x = \mathscr{D}_d(X)$. When keys are understood, we write $\mathscr{D}_i(X)$ to denote the function computing decryption share of party P_i for ciphertext X, and $x = \mathscr{D}(X, \{x_i\}_{i \in I})$ to denote the process of combining the decryption shares $\{x_i\}_{i \in I}$ to a plaintext x.

Security. We require the usual security of the threshold cryptosystem, cf. [13], and in particular require that there exists an efficient two-party zero-knowledge protocol for proving the correctness of decryption shares.

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 verification 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 verification function, where $\mathsf{Ver}_v(x,\sigma) = 1$ indicates that σ is a valid signature on the message x. We write Sign_i/Ver_i to denote the signing/verification operation of party P_i.

Threshold signatures. We assume the existence of a *threshold* signature scheme, which is unforgeable against an adaptive chosen message attack. For a signing key s and a verification 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 verification 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 sharing of $\mathscr{S}_s$ among n parties, i.e. that for a given signing threshold t_S, $1 < t_\mathsf{S} \leq n$, there exists a sharing $(s_1, \ldots, s_n)$ of the signing key s (where s_i is intended for P_i), such that given signature shares $\sigma_i = \mathscr{S}_{i,s_i}(x)$ for t_S distinct signing-key shares s_i, it is possible to efficiently compute a signature σ satisfying $\mathscr{V}_v(x,\sigma) = 1$. We will always have $t_\mathsf{S} = n - t$. When keys are understood, we use $\mathscr{S}_i(x)$ to denote the function computing P_i's signature share for the message x, and $\sigma = \mathscr{S}(x, \{\sigma_i\}_{i\in I})$ to denote the process of combining the signature shares $\{\sigma_i\}_{i\in I}$ to a signature σ.

Security. The scheme should be unforgeable against adaptive chosen message attack when the adversary is given $(t_\mathsf{S} - 1)$ signing-key shares, and we require that there exists an efficient two-party zero-knowledge protocol for proving the correctness of signature shares.

Byzantine Agreement. We require a Byzantine Agreement (BA) protocol: Each P_i has input $v_i \in \{0,1\}$ and output $w_i \in \{0,1\}$, where: *Termination*: If all honest parties enter the BA, then the BA eventually terminates. *Consistency:* Upon termination the outputs of all honest players are equal, i.e. $w_i = w$ for some $w \in \{0,1\}$. *Validity:* If all honest parties have input $v_i = w$, then the output is w.

Cryptographic assumptions & instantiations of tools. All the above tools can be instantiated in the standard (random oracle devoid) model using known results from [23, 16, 14, 13, 3, 25, 22, 7]. For details see [20].

3 Certificates

In order to achieve robustness we require every party to prove (in zero-knowledge) the correctness of essentially every value she provides during the protocol execution. To implement this process efficiently we introduce *certificates*, which are used for certifying the truth of claims. Any party can verify the correctness of a certificate locally, without any interaction. Moreover, a certificate should provide no other information than the truth of the claim. Finally, a party can convince any other party about the truth of the corresponding claim by sending the certificate. More formally, we say that a bit-string α *is a certificate for claim* m if there exits a publicly known, efficiently computable verification procedure V, such that the following conditions are satisfied, except with

negligible probability: if $V(\alpha, m) = 1$ then claim m is true (soundness), and α gives no other information than the truth of the claim m (zero-knowledge). Moreover we require completeness, i.e. the ability of generating certificates for true claims needed in our protocols, like for example:

(i) «P_i *knows the plaintext of* X_i» (iii) «*the plaintext of* X_i *is in the set* $\{0, 1, 2\}$»
(ii) «X_i *is the unique input of* P_i» (iv) «*at least* $n - t$ *parties have received* X_i»

If X is some value, and α is a certificate for some claim m about X (e.g., claim (iii) above), then we say that X *is a value certified (by* α*) for claim* m.

We often require also that the certificates for correctness/validity of some data X imply also *uniqueness* of the data, i.e. that it is not possible to obtain two valid certificates for two different values for the same claim. This can be achieved by assigning unique identifiers to every gate, every wire and every step in the protocols, and requiring that the identifiers are parts of the claims, e.g. «X_i *is input of* P_i *for wire* id», and that parties participate in construction of at most one certificate for a particular claim. Occasionally, to clarify the issues, we explicitly specify the identifiers, but for simplicity the use of identifiers is usually implict.

Constructing certificates. Certificates can be implemented in a simple way using any signature scheme $(\mathsf{Sign}, \mathsf{Ver})$: a certificate α for claim m is just a set of at least $n - t$ correct signatures: $\alpha := \{\sigma_i\}_{i \in I}$, where $|I| \geq n - t$ and each σ_i is a signature of party P_i on message m. To create *short* certificates we employ a *threshold* signature scheme $(\mathscr{S}, \mathscr{V})$ with a threshold $t_{\mathsf{S}} = n - t$ (cf. Sect. 2.1). To construct a certificate α valid for «*some claim*» a party collects t_{S} correct *signature shares* $\sigma_j = \mathscr{S}_j$(«*some claim*») from different parties, and combines them to a signature $\alpha = \mathscr{S}$(«*some claim*», $\{\sigma_j\}_{j \in J}$), where $|J| \geq t_{\mathsf{S}}$. Any party knowing the corresponding public verification key v can verify α using the algorithm $\mathscr{V}$. Depending on the context, we use different methods for creating certificates:

bilateral proofs: if P_i needs to certify knowledge of some value, or validity of some NP-statement (cf. examples (i) and (iii), respectively), we will use 2-party zero-knowledge proofs: P_i bilaterally proves a claim m in zero-knowledge to every P_j, who then, upon successful completion of the proof, sends to P_i a signature share $\sigma_j = \mathscr{S}_j(m)$ with a proof of correctness of the share, and P_i combines the correct shares to get a certificate α_i. We say then that "P_i *constructs certificate* α_i *for «some claim» by bilateral, zero-knowledge proofs*", denoted as $\alpha_i := \mathsf{certify}_{\mathsf{zkp}}$(«*some claim*»).

protocol-driven: For other claims, like (iv) and (ii), P_i also constructs a certificate α_i from a set of $n - t$ signature shares σ_j, but this time P_j sends σ_j not in response to a bilateral proof, but based on the current context of execution, as required by the protocol. In this case we just say "P_i *constructs certificate* α_i *for «some claim»*" and write $\alpha_i := \mathsf{certify}$(«*some claim*»). [1]

[1] Note that the signed messages can be different from the actual claim being certified, e.g, each P_j could provide a signature share for the message «*I have seen* X_i», and a complete signature on such a message can be interpreted as a certificate for (iv).

An adversary corrupting up to t players can never obtain sufficiently many signature shares: In the case of bilateral proofs an honest party never signs an incorrect claim, hence the adversary can collect at most $t < n - t$ shares. In the protocol-driven case, the soundness depends on the actual claim being certified, but it will be clear from the context. Note that the threshold $n - t$ implies that $n - 2t$ honest parties must sign to create a certificate, which ensures uniqueness.

4 The new protocol

Our protocol needs that the encryption key of a public-key encryption scheme is publicly known, while the corresponding decryption key is shared among all the players. Given such a setup the evaluation of a circuit proceeds as follows. First the parties provide their inputs as ciphertexts of the encryption scheme. Then they cooperate to evaluate the circuit gate-by-gate: given encryptions of inputs of a gate, parties compute an encryption of the corresponding output of the gate, while maintaining privacy of the intermediate values. Finally, after an encryption of the output gate is computed, parties decrypt this encryption to learn the output. The robustness against corrupted parties is achieved with help of *certificates*, which are used to certify the correct execution of the protocol.

Intuitively, the efficiency gain stems from a combination of a ballanced distribution of work, with the so-called *circuit-randomization technique* due to Beaver [1]. In this technique the multiplication of two encrypted values is performed using a *pre-generated* random triple, which in our case consists of three ciphertexts (U, V, W) containing secret random plaintexts $u, v, w \in \mathbb{M}$, satisfying $u \cdot v = w$. Due to homomorphic encryption, given such a triple and two ciphertexts A, B containing plaintexts a, b, we compute a ciphertext C of $c = a \cdot b$ by *publicly decrypting* $A + U$ and $B + V$, and by using the following identity

$$a \cdot b = (a + u) \cdot (b + v) - (a + u) \cdot v - u \cdot (b + v) + w \; . \tag{1}$$

Main protocol — a high-level overview. The protocol proceeds in four stages, a *precomputation stage*, an *input stage*, an *evaluation stage*, and a *termination stage*. We briefly summarize the goal of each stage:

- *Precomputation stage*: Players generate random triples.
- *Input stage*: Each player provides an encryption of his input to every other player, and the players agree on a set of *input providers*.
- *Evaluation stage*: Players evaluate the circuit gate-by-gate, by executing concurrently subprotocols for every gate of the circuit.
- *Termination stage*: Executed concurrently to the evaluation stage, this stage ensures that every player eventually receives the output(s) and terminates.

Strictly speaking, the presented protocol is limited to the evaluation of *deterministic* circuits, but can be easily extended also to *randomized* circuits [20].

The circuit. For the clarity of presentation we assume that every party provides exactly one input, and that the outputs are public, but this is without loss of generality [19]. The function to be computed is given as a circuit Circ over the plaintext space $\mathbb{M}$ of the homomorphic encryption scheme in use. The circuit is a set of labeled gates, where a label G uniquely identifies the gate. The full description of a gate is a tuple $(G, \ldots)$, where the parameters after the label depend on the type and the position of the gate. We denote by $\mathcal{G}$ the set of all gate labels of Circ, and we use $v : \mathcal{G} \to \mathbb{M} \cup \{\perp\}$ to refer to the values of gates, i.e., $v(G)$ denotes the value of gate G. Each gate has 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 $l \geq 0$, $a_0, \ldots, a_l \in \mathbb{M}$ 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: $(G, \mathsf{output}, G_1)$, where $v(G) = v(G_1)$ is an output value of Circ.

Dictionary. Throughout the computation each party P_i maintains a *dictionary* $\Gamma_i : \mathcal{G} \to \mathbb{C} \cup \{\perp\}$, containing P_i's view on the intermediate values (encryptions) in the circuit. Initially $\Gamma_i(G) = \perp$ for all labels from $\mathcal{G}$. If $\Gamma_i(G) = X \neq \perp$, then from P_i's point of view evaluation of gate G was completed, and X is a ciphertext encrypting the value $v(G)$. We say then that P_i *has accepted* X *for* G. Honest parties will agree on accepted ciphertexts, allowing us to define a common map Γ. Furthermore, for all input gates $\Gamma(G) = X$ will be an encryption of the input that the party supplying input to that gate intended to deliver, except for at most t parties, where X might be an encryption of a default value. This is allowed by the security model.

Random Triples. Each party P_i maintains also a mapping Δ_i assigning to each multiplication gate a random triple generated during the precomputation stage, $\Delta_i : \mathcal{G} \to \mathbb{C} \times \mathbb{C} \times \mathbb{C} \cup \{\perp\}$. Initially $\Delta_i(G) = \perp$, for all gates G and all $j \in [n]$. If $\Delta_i(G) = (U, V, W) \neq \perp$, then P_i will use this triple for evaluating gate G. The honest parties P_i and P_j will have $\Delta_i = \Delta_j$.

5 Subprotocols used by the main protocol

Below we present subprotocols of the main protocol. First we present a protocol SELECT, which is a basic subprotocol used both in the precomputation and input stages. Then we describe the subprotocols for the main stages.

Selecting values. Protocol SELECT is used for selecting values provided by the players during the computation. It is parametrized by a condition φ (like e.g. «X_i *is* P_i*'s valid input*»), which has to be satisfied for each input to the protocol, and certified by an appropriate certificate. We require that φ *implies uniqueness*, i.e., that every party can obtain a corresponding certificate valid for φ for at most one input value used in any execution of SELECT.

The protocol proceeds as follows (cf. Fig. 1): First P_i distributes its input (X_i, α_i) to all parties, and then constructs and distributes a *certificate of distribution* β_i, which

Protocol SELECT(φ)*, code for* P_i: given input X_i with a certificate α_i valid for condition $\varphi(i)$ initialize sets $A_i, \boldsymbol{\mathcal{A}}_i, C_i$ as empty, then execute the following rules concurrently:

DISTRIBUTION:

1. send (X_i, α_i) to all parties.
2. construct and send to all parties $\beta_i := \mathsf{certify}(\text{«}\textit{we hold } P_i\textit{'s input } X_i\text{»})$

GRANT CERTIFICATE OF DISTRIBUTION: Upon first (X_j, α_j) from P_j with α_j valid for $\varphi(j)$: add j to A_i, add (X_j, α_j) to $\boldsymbol{\mathcal{A}}_i$, and send $\sigma_i := \mathsf{Sign}_i(\text{«}\textit{we hold } P_j\textit{'s input } X_j\text{»})$ to P_j.

ECHO CERTIFICATE OF DISTRIBUTION: Upon (X_j, β_j) with β_j valid for «*we hold* P_j*'s input* X_j» and $j \notin C_i$: add j to C_i and send (X_j, β_j) to all parties.

SELECTION: If $|C_i| \geq n-t$, stop executing all 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 $(n-t)$ well-formed $(A_j, \boldsymbol{\mathcal{A}}_j)$; let $B_i := \bigcup_{j \in J} A_j$ and $\boldsymbol{\mathcal{B}}_i := \bigcup_{j \in J} \boldsymbol{\mathcal{A}}_j$
3. enter n Byzantine Agreements (BAs) with inputs $v_1 \ldots v_n$, where $v_j = 1$ iff $j \in B_i$.
4. let $w_1, \ldots, w_n$ be the outputs of the BAs; let $\mathcal{W} := \{j \in [n] |\ w_j = 1\}$.
5. $\forall\, j \in B_i \cap \mathcal{W}$: send $(X_j, \alpha_j) \in \boldsymbol{\mathcal{B}}_i$ to all parties.
6. collect and output $(\mathcal{W}, \{(X_j, \alpha_j)\}_{j \in \mathcal{W}})$.

Fig. 1: Protocol SELECT(φ).

proves that P_i has distributed (X_i, α_i) to at least $n - t$ parties. When a party collects $n-t$ certificates of distribution, she 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 their certified inputs distributed to at least $(n-t)-t \geq t+1$ *honest* parties. Hence, 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.

Precomputation stage. The goal of this stage is the generation of certified random triples. The corresponding protocol GEN-TRIPLES uses two subprotocols: protocol SELECT presented above, and protocol ONE-TRIPLE for generating a single random triple in a computation lead by one of the parties.[2] Given these two sub-protocols, we proceed as follows (Fig.2): first every party generates its own random triple using ONE-TRIPLE, and then uses this triple as input to SELECT, in which parties agree on at least $(n-t)$ triples.

In protocol ONE-TRIPLE we need to generate certified, encrypted random values unknown to any party, so first we present a sub-protocol RANDOM (Fig. 3), which achieves exactly that. Given (U, α) output by RANDOM, king P_k can extend it to a random triple using ONE-TRIPLE, see Fig. 4. Note that when computing a certificate β_i for the claim «P_i *knows* v_i *in* V_i*, and* W_i *is a randomization of* $v_i \star U$» the variables P_i, V_i, W_i, and U

[2] In the ONE-TRIPLE protocol one party, say P_k, plays the role of a leader (called *king*) who with help of other players (called *slaves*), generates P_k's own random triple $(U^{(k)}, V^{(k)}, W^{(k)})$ together with a certificate $\sigma^{(k)}$ certifying the triple's correctness.

To generate c_M random triples and reach agreement on them, parties proceed as follows:

1. Every party P_k, $k \in [n]$ starts as a king for $\ell = \lceil c_M/(n-t) \rceil$ instances of ONE-TRIPLE(id, k) protocol to generate ℓ certified random triples $(U^{(k,s)}, V^{(k,s)}, W^{(k,s)}; \sigma^{(k,s)})$, $s = 1, \ldots, \ell$ (all parties play roles of slaves to help the king in P_k's instances)
2. All parties start SELECT, where party P_i uses as its input the triples $(U^{(i,s)}, V^{(i,s)}, W^{(i,s)}; \sigma^{(i,s)})$ generated in the previous step. When SELECT terminates, parties have agreed on a set of at least $\ell(n-t) \geq c_M$ valid triples $\{(U^{(j)}, V^{(j)}, W^{(j)})\}_{j \in J}$.
3. Every party P_i initializes its mapping Δ_i, using the triples from the previous step in some pre-agreed order.

Fig. 2: Protocol GEN-TRIPLES for generating random triples.

To generate for king P_k a certified random ciphertext (U, α), with α valid for «*U: 1st part of triple* $\mathsf{id}(k)$» parties proceed as follows:

GENERATION: code for every P_i:

1. pick random $u_i \in \mathbb{M}$ and compute $U_i := \mathscr{E}(u_i)$
2. construct $\beta_i = \mathsf{certify}_{\mathsf{zkp}}(\text{«}P_i \textit{ knows } u_i \textit{ in } U_i\text{»})$
3. compute $\sigma_i := \mathsf{Sign}_i(\text{«}U_i : \textit{component of 1st part of triple } \mathsf{id}(k)\text{»})$
4. send (U_i, β_i, σ_i) to P_k:

CONSTRUCTION: code for P_k:

1. collect a set $S_{\mathsf{id}(k)} := \{(U_i, \beta_i, \sigma_i)\}_{i \in I_{\mathsf{id}(k)}}$, $|I_{\mathsf{id}(k)}| \geq t+1$, with each β_i valid for «P_i *knows* u_i *in* U_i»,
   and each σ_i valid for «U_i : *component of 1st part of triple* $\mathsf{id}(k)$».
2. send $S_{\mathsf{id}(k)}$ to all parties; each P_i computes $U := \bigoplus_{i \in I_{\mathsf{id}(k)}} U_i$, and helps to construct α in the next step.
3. construct $\alpha := \mathsf{certify}(\text{«}U\textit{: 1st part of triple } \mathsf{id}(k)\text{»})$.
4. output (U, α).

Fig. 3: Protocol RANDOM(id, k) for generating a certified random value for king P_k.

are replaced by the actual values they stand for, but v_i stays as a literal, since it is just a name for the plaintext from V_i.

On the use of Byzantine Agreement. The protocol GEN-TRIPLES uses n BAs (as it invokes SELECT) and generates $n-t$ random triples. To implement multiplication of encrypted values via circuit randomization (cf. Fig. 6), we need one random triple per multiplication gate. A straightforward solution would be to use $\ell = \lceil c_M/(n-t) \rceil$ runs of GEN-TRIPLES, but this would lead to $\mathcal{O}(c_M)$ invocations of BA. To avoid this, we run ℓ invocations of ONE-TRIPLE in parallel, using only one invocation of SELECT. In particular, in the second step of GEN-TRIPLE each P_i uses all ℓ triples as its input to SELECT. Since SELECT returns a set of at least $(n-t)$ inputs, we obtain an agreement on c_M random triples with only n BAs, which is independent of the circuit size.

Input stage. The protocol is presented in Fig. 5. When providing (encrypted) inputs, the parties are required to prove plaintext knowledge for their encryptions, to ensure

To generate for P_k a certified random triple $(U, V, W; \beta)$, with β valid for «(U, V, W): *correct triple* $\mathsf{id}(k)$» parties proceed as follows:

REQUEST: code for P_k:

1. run RANDOM(id, k) to generate (U, α), with α valid for «U: *1st part of triple* $\mathsf{id}(k)$», and send (U, α) to all parties.

REPLY: code for every P_i:

1. wait for (U, α) from P_k
2. compute $V_i := \mathscr{E}(v_i)$ and $W_i = \mathscr{R}(v_i \star U)$ for a random $v_i \in \mathbb{M}$
3. construct $\beta_i := \mathsf{certify}_{\mathsf{zkp}}$(«$P_i$ *knows* v_i *in* V_i, *and* W_i *is a randomization of* $v_i \star U$»)
4. compute $\sigma_i := \mathsf{Sign}_i$(«$(V_i, W_i)$: *part of triple* $\mathsf{id}(k)$»)
5. send $(V_i, W_i; \beta_i, \sigma_i)$ to P_k

CONSTRUCTION: code for P_k:

1. collect $T_{\mathsf{id}(k)} := \{(V_i, W_i; \beta_i, \sigma_i)\}_{i \in I_{\mathsf{id}(k)}}$, with each σ_i valid for «(V_i, W_i) : *part of triple* $\mathsf{id}(k)$», and each β_i valid for «P_i *knows* v_i *in* V_i, *and* W_i *is a randomization of* $v_i \star U$» $|I_{\mathsf{id}(k)}| \geq t + 1$.
2. send $T_{\mathsf{id}(k)}$ to all parties; each P_i computes $V := \bigoplus_{i \in I_{\mathsf{id}(k)}} V_i$, $W := \bigoplus_{i \in I_{\mathsf{id}(k)}} W_i$, and helps to construct β in the next step.
3. construct $\beta :=$ $\mathsf{certify}$(«(U, V, W): *correct triple* $\mathsf{id}(k)$»).
4. output $(U, V, W; \beta)$.

Fig. 4: Protocol ONE-TRIPLE(id, k) for generating a random triple for king P_k.

Input stage code for P_i: given an input $x_i \in \mathbb{M}$ do the following:

1. compute $X_i := \mathscr{E}(x_i)$ and construct $\alpha_i := \mathsf{certify}_{\mathsf{zkp}}$(«$X_i$ *is* P_i*'s valid input*») (every party P_j helps to construct at most one α_i, for each $P_i \in \mathcal{P}$).
2. enter execution of SELECT protocol with input (X_i, α_i).
3. output value(s) returned by SELECT.

Fig. 5: The input stage code for P_i holding input $x_i \in \mathbb{M}$.

independence of the inputs. To cope with the inherent problems of the asynchronous setting we use a protocol SELECT to agree on inputs from at least $(n-t)$ *input providers*, whose private inputs will be used in the actual computation. For the remaining inputs the default values will be used.

Computing linear gates. Due to the homomorphic property of encryption, linear gates are computed locally, without interaction: after P_i accepts encryptions of inputs to a gate $(G, \mathsf{linear}, a_0, G_1, a_1, \ldots, G_l, a_l)$, i.e. when $\Gamma_i(G_u) \neq \perp$, for $u = 1 \ldots l$, then P_i computes $\Gamma_i(G) := A_0 \oplus \left(\bigoplus_{u=1}^{l} (a_j \star \Gamma_i(G_u))\right)$, where A_0 is a "dummy" encryption of a_0, computed using fixed, public random bits.

Party P_i evaluating a multiplication gate $(G, \mathsf{mul}, G_1, G_2)$:

1. wait until $A := \Gamma_i(G_1) \neq \perp$, $B := \Gamma_i(G_2) \neq \perp$, and $(U, V, W) := \Delta_i(G) \neq \perp$
2. compute $X := A \oplus U$ and $Y := B \oplus V$
3. compute decryption shares and corresponding validity proofs: $x_i := \mathscr{D}_i(X)$, $\beta_i := \mathsf{certify}_{\mathsf{zkp}}(\text{«}x_i \textit{ is valid}\text{»})$ $y_i := \mathscr{D}_i(Y)$, $\gamma_i := \mathsf{certify}_{\mathsf{zkp}}(\text{«}y_i \textit{ is valid}\text{»})$; send (x_i, β_i) and (y_i, γ_i) to all parties
4. collect sets $\mathcal{X} := \{(x_j, \beta_j)\}$ and $\mathcal{Y} := \{(y_j, \gamma_j)\}$, each containing t_{D} correct decryption shares, with corresponding validity proofs.
5. compute plaintexts $x := \mathscr{D}(X, \mathcal{X})$ and $y := \mathscr{D}(Y, \mathcal{Y})$.
6. compute $Z := \mathscr{E}(x \cdot y, r_0)$ for a public constant r_0, and set $\Gamma_i(G) := Z \ominus (x \star V) \ominus (y \star U) \oplus W$

Fig. 6: Code for P_i evaluating a multiplication gate.

Party P_i evaluating an output gate $(G, \mathsf{output}, G_1)$:

1. wait until $\Gamma_i(G_1) = C \neq \perp$
2. compute a decryption share $c_i := \mathscr{D}_i(C)$ & a certificate $\delta_i := \mathsf{certify}_{\mathsf{zkp}}(\text{«}c_i \textit{ is valid}\text{»})$; send (c_i, δ_i) to every P_j
3. collect a set $T = \{(c_j, \delta_j)\}$ of t_{D} decryption shares for C, with corresponding validity certificates δ_j
4. compute $c := \mathscr{D}(C, T)$
5. compute and send to all parties a signature share $\sigma_{i,G} = \mathscr{S}_i(\text{«}\textit{The value of } G \textit{ is } c\text{»})$, together with a certificate of its correctness, $\xi_{i,G} := \mathsf{certify}_{\mathsf{zkp}}(\text{«}\sigma_{i,G} \textit{ is correct}\text{»})$
6. collect a set $\{\sigma_{i,G}, \xi_{i,G}\}_{i \in I}$ of t_{S} certified signature shares and compute $\zeta_G = \mathscr{S}(x, \{\sigma_i\}_{i \in I})$ valid for «*The value of G is c*»
7. mark G as `decrypted`

Fig. 7: Code for player P_i evaluating an output gate.

Computing multiplication gates. The multiplication protocol (Fig. 6) is based on a trick by Beaver [1]. Essentially, this trick reduces the problem of multiplication to two decryptions and a few linear operations (cf. eq. (1)).

Output stage. When P_i completes the computation of a gate $(G, \mathsf{output}, G_1)$ (i.e. when $\Gamma_i(G) = C \neq \perp$), but the gate has not been decrypted yet, then P_i sends a decryption share c_i of C to all parties, along with a certificate for the correctness of the share. Every P_j collects sufficiently many certified decryption shares, and uses them to decrypt the output. Subsequently the parties construct a certificate ζ_G, which certifying that the decrypted output value is correct. With such a certificate any party P_i can convince any other party about the correctness of the output.

Termination stage. Essentially, every P_i waits until he receives or computes the decrypted output value with a correctness certificate, and echoes this certified output to all parties before terminating (see [20] for details).

Summary. The main result of this paper is summarized in the theorem below. The analysis leading to this theorem is presented in the full version [20].

Theorem 1. *Assuming the cryptographic primitives from Sect. 2.1, there exists a protocol allowing n parties connected by an asynchronous network to securely evaluate any circuit in the presence of a poly-time adversary actively corrupting up to $t < n/3$ parties. The bit complexity of the protocol is $\mathcal{O}((c_I + c_M + c_O)n^2\kappa)$, where c_I, c_M, c_O denote the number of input, multiplication, and output gates, respectively, and κ is a security parameter.*

References

1. D. Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, pp. 420–432, 1991.
2. M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proc. ACM CCS*, pp. 62–73, 1993.
3. M. Bellare and P. Rogaway. The exact security of digital signatures — How to sign with RSA and Rabin. In *Proc. EUROCRYPT '96*, pp. 399–416, 1996.
4. M. Ben-Or, R. Canetti, and O. Goldreich. Asynchronous secure computation. In *STOC*, pp. 52–61, 1993.
5. M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proc. 20th STOC*, pp. 1–10, 1988.
6. M. Ben-Or, B. Kelmer, and T. Rabin. Asynchronous secure computations with optimal resilience. In *Proc. 13th PODC*, pp. 183–192, 1994.
7. C. Cachin, K. Kursawe, and V. Shoup. Random oracles in Constantinopole: Practical asynchronous Byzantine agreement using cryptography. In *Proc. 19th PODC*, pp. 123–132, 2000.
8. R. Canetti. *Studies in Secure Multiparty Computation and Applications*. PhD thesis, Weizmann Institute of Science, Rehovot 76100, Israel, June 1995.
9. R. Canetti. Security and composition of multiparty cryptographic protocols. *JoC*, 13(1):143–202, 2000.
10. R. Canetti, O. Goldreich, and S. Halevi. The random oracle methodology, revisited. In *Proc. 30th STOC*, pp. 209–218, 1998.
11. R. Canetti and T. Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *Proc. 25th STOC*, pp. 42–51, 1993.
12. D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols (extended abstract). In *Proc. 20th STOC*, pp. 11–19, 1988.
13. R. Cramer, I. Damgård, and J. B. Nielsen. Multiparty computation from threshold homomorphic encryption. In *Proc. EUROCRYPT '01*, pp. 280–300, 2001.
14. I. Damgård and M. Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In *Proc. 4th PKC*, pp. 110–136, 2001.
15. A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Proc. CRYPTO '86*, pp. 186–194, 1986.
16. P.-A. Fouque, G. Poupard, and J. Stern. Sharing decryption in the context of voting or lotteries. In *Proc. Financial Cryptography '00*, 2000.
17. M. Franklin and S. Haber. Joint encryption and message-efficient secure computation. *JoC*, 9(4):217–232, 1996. Preliminary version in Proc. CRYPTO '93.
18. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game — a completeness theorem for protocols with honest majority. In *Proc. 19th STOC*, pp. 218–229, 1987.
19. M. Hirt, J. B. Nielsen, and B. Przydatek. Cryptographic asynchronous multi-party computation with optimal resilience (extended abstract). In *Proc. EUROCRYPT '05*, pp. 322–340.
20. M. Hirt, J. B. Nielsen, and B. Przydatek. Asynchronous multi-party computation with quadratic communication, 2008. Full version of this paper, eprint.iacr.org.

21. U. Maurer, R. Renner, and C. Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In *Proc. TCC '04*, pp. 21–39.
22. J. B. Nielsen. A threshold pseudorandom function construction and its applications. In *Proc. CRYPTO '02*, pp. 401–416, 2002.
23. P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proc. EUROCRYPT '99*, pp. 223–238, 1999.
24. B. Prabhu, K. Srinathan, and C. P. Rangan. Asynchronous unconditionally secure computation: An efficiency improvement. In *Proc. Indocrypt 2002*, pp. 93–107.
25. V. Shoup. Practical threshold signatures. In *Proc. EUROCRYPT '00*, pp. 207–220, 2000.
26. A. C. Yao. Protocols for secure computations. In *Proc. 23rd IEEE FOCS*, pp. 160-164, 1982.