# Almost-Asynchronous MPC with Faulty Minority

Zuzana Beerliová-Trubíniová[1], Martin Hirt[1], and Jesper Buus Nielsen[2⋆]

[1] ETH Zürich
[2] University of Aarhus

**Abstract.** Secure multiparty computation (MPC) allows a set of parties to securely evaluate any agreed function of their inputs, even when up to $t$ of the $n$ parties are faulty. Protocols for synchronous networks (where every sent message is assumed to arrive within a constant time) tolerate up to $t < n/2$ faulty parties, whereas in the more realistic asynchronous setting (with no *a priory* information on maximal message delay) only security against $t < n/3$ is possible. We present the first protocol that achieves security against $t < n/2$ without assuming a fully synchronous network. Actually our protocol guarantees security against any faulty minority in an *almost asynchronous* network, i.e. in a network with one single round of synchronous broadcast (followed by a fully asynchronous communication). Furthermore our protocol takes inputs of all parties (in a fully asynchronous network only inputs of $n - t$ parties can be guaranteed), and so achieves everything that is possible in synchronous networks (but impossible in fully asynchronous networks) at the price of just one synchronous broadcast round.

As tools for our protocol we introduce the notions of *almost non-interactive verifiable secret-sharing* and *almost non-interactive zero-knowledge proof of knowledge*, which are of independent interest as they can serve as efficient replacements for fully non-interactive verifiable secret-sharing and fully non-interactive zero-knowledge proof of knowledge.

⋆ Supported by Danish Agency for Science Technology and Innovation

# Table of Contents

# 1 Introduction

## 1.1 Multiparty Computation

Secure multiparty computation (MPC) allows a set of $n$ parties to securely evaluate any agreed function of their inputs, even if $t$ of the parties are corrupted by a central adversary. In this paper we focus on an active adversary, that can take full control over the corrupted parties (i.e., read their internal state and make them send wrong messages). A protocol is called secure if the uncorrupted parties output the correct function values (*correctness*), and if the adversary does not learn anything that cannot efficiently be derived from the inputs and outputs of the corrupted parties (*privacy*).

The MPC problem dates back to Yao [Yao82], and the first generic solutions were presented in [GMW87,CDG87]. These protocols are secure for $t < n/2$, and this is known to be optimal when the protocols are guaranteed to be deadlock-free – i.e., they deliver outputs even when the adversary actively tries to make this not happen.

## 1.2 Synchronous vs. Asynchronous Communication

The above mentioned protocols, and most protocols following them [BGW88,CCD88,RB89,Bea89, ...], were for the *synchronous model*, where it is assumed that the delay of messages in the network connecting the parties is bounded by a *known* constant.

This synchronicity assumption is extremely powerful, because it allows MPC protocols to proceed in rounds, with the guarantee that every message sent in some round will be delivered at the beginning of the next round. This again allows a party $P_j$ that did not receive an expected message from another party $P_i$ to conclude that $P_i$ is corrupted. A typical use of this ability is the following: If $P_j$ does not receive some message $m_{i,j}$ that should have been sent by $P_i$ according to the protocol, then $P_j$ broadcasts a complaint on $P_i$. In response $P_i$ must broadcast $m_{i,j}$.[1] Now, all parties can see whether or not $P_i$ sends the message in question. So, either the protocol can continue, or all parties agree that $P_i$ is corrupted. In the last case, the secrets of $P_i$ could be publicly reconstructed to allow the other parties to proceed without the help of $P_i$.

On the negative side, the assumptions of the synchronous model, forces the implementation of synchronization to be secure in a cryptographic sense. If a party $P_i$ is prevented from delivering a message, due to a cable being dug over or a flooding attack by some hacker, this affects the *privacy* of $P_i$ and maybe even the privacy other honest parties.[2] The security level of the synchronization protocol must therefore be as high as that of e.g. the encryption schemes used by the protocol. To get this high security against even such simple attacks as a broken cable and a flooding attack, the timeouts probably should be hours, as a party taking part in the computation should have time to recover from such events before their privacy is broken by the other parties. Note that this has nothing to do with the round trip time of the network in normal operation! To be cryptographically secure, the timeouts must be able to handle any unintended delaying event which can occur with *non-negligible probability*.[3] In fact, we do not know of a single published work which claims to implement a general synchronization mechanism suitable for emulating the synchronous cryptographic model in e.g. Internet-like environments.

Partially as a response to this, asynchronous protocols have been developed [CR93,BCG93,BKR94]. In the *asynchronous model*, arbitrary delays in the network are allowed, with the only restriction that eventually every sent message must be delivered. In order to model the worst case, the adversary is allowed to control the scheduling of messages in the

[1] Broadcasting $m_{i,j}$ is secure because either $P_i$ did not send a message (and thus is corrupted) or $P_j$ is lying about not having received it (and thus is corrupted), meaning that $m_{i,j}$ is already known to the adversary.

[2] Some protocols publicly reconstruct the secrets of $P_i$, which might include its inputs and the shares of the secrets of other parties.

[3] If the computers participating in a multiparty computation is supervised by a human, timeouts in the minutes might be sufficient – a supervisor could respond to a broken cable or a flooding attack by calling the other supervisors, report the claimed problem and temporarily halt the computation.

network. The asynchronous model is much easier to implement. On the Internet it is e.g. more or less implemented by the secure socket layer (SSL).[4] Furthermore, asynchronous protocols are best-effort protocols: When the message delays are short, the protocols are fast. This contrasts synchronous protocols, which have to allow each round to be long enough, such that all messages can get through, even in the very worst case.

On the downside, protocols for the asynchronous model have two inherent drawbacks when compared with their synchronous counterparts:

INPUT DEPRIVATION: A slow honest party cannot be distinguished from a corrupted party not sending messages at all, and can therefore not be waited for. Therefore a fully asynchronous protocol will sometimes exclude up to $t$ *honest* parties from providing their input to the computation. For some computations like auctions and elections, input deprivation is completely intolerable.

LOWER THRESHOLD: Since $t$ honest parties can be excluded entirely from the computation due to delay, the actual computation might be performed by just $n - t$ parties, of which $t$ might be corrupted. Since honest majority is still needed between the participating parties, this means that there must be $t + 1$ honest parties left among the $n - t$ parties, which requires that $t < n/3$. No fully asynchronous protocol can tolerate $t < n/2$, opposed to the best synchronous protocols.

### 1.3 Hybrid Network

The problem of input deprivation was addressed recently in [HNP05] and [BH07]. In order to achieve input guarantee without assuming a fully synchronous network, [HNP05] proposed a so-called hybrid network with few synchronous rounds followed by a body of asynchronous computation. Later, [BH07] showed that input guarantee can be achieved with only one synchronous round (followed by many asynchronous rounds). This is clearly optimal in the number of synchronous rounds, and can be practical even in settings where implementing synchronous communication is very inefficient. As an example, to run an auction one can have an initial deadline of a week for sealed bids to arrive – reasonable time to get a message through even when under attack. After this deadline the result is then securely computed using a completely asynchronous protocol.

However, the above protocols still suffer from the second problem of asynchronous protocols: they tolerate lower thresholds than their synchronous counterparts. In the cryptographic model, this means tolerating only $t < n/3$ instead of $t < n/2$.

### 1.4 Contributions

In this paper we present an MPC protocol for an almost-asynchronous network that achieves everything that is possible in the synchronous world (input guarantee and security threshold $t < n/2$). More precisely, we show that input guarantee and security $t < n/2$ is achievable with one single round of synchronous broadcast (followed by asynchronous computation). This way we get the best of both worlds: input guarantee and security threshold $t < n/2$ on one hand and minimal synchronicity assumptions on the other hand.

Our protocol starts by every party broadcasting an encryption of its inputs[5] (along with a non-interactive proof of plaintext knowledge). Then the circuit is asynchronously evaluated gate-by-gate, such that every party eventually learns an encryption of every intermediate result, leading into every party eventually learning an encryption of the output, which is then decrypted.

However, as in a fully asynchronous network no Byzantine agreement is possible for $t \geq n/3$, after the synchronous broadcast round in the input phase our protocol runs without any further agreement

[4] To implement that all messages *eventually* get delivered, all that is needed on top of the SSL is a way to reestablish broken connections, resulting from e.g. broken cables or broken routers. This need not be speedy and the time to reestablish connections only affect the running time of the protocol when needed. So, connections just have to be reestablished eventually when they get broken – a manageable requirement.

[5] A party inputs a number to each input gate associated to that party.

on whatsoever. In particular, this means that the parties might get different *encryptions* of the (same correct) output.

Nevertheless, the following holds: once the encryptions of the inputs are consistently distributed among the parties, it is possible to securely compute any deterministic function in a *fully asynchronous* network with honest majority (and thus without any further agreement).

### 1.5 Related Work

The number of synchronous rounds of broadcast needed for a fully synchronous protocol has been studied in [BB89,BMR90,IK00,GIKR02], and is known to be 3. Lately Katz, Koo and Kumaresan [KK06,KK07,KKK08] have done important progress on the subject, by focusing on the number of synchronous rounds of point-to-point communication instead of insisting on all rounds being rounds of broadcast. This allows to do much better than emulating each of the three needed rounds of broadcast individually. Still, the models of Katz, Koo and Kumaresan insist on all rounds being synchronous, and therefore fall pray to known lower bounds on the number of synchronous rounds needed. Our result can, on the other hand, do with as little as one synchronous round of *point-to-point* communication, as discussed in the conclusions of this paper.

## 2 Preliminaries

### 2.1 Model

We consider a set of $n$ parties $\mathcal{P} = \{P_1, \ldots, P_n\}$, each $P_i$ holding an input $x_i$.

The faultiness of parties is modeled by a central poly-time adversary who can corrupt up to $t < n/2$ of the parties (for a given threshold $t$) and make them deviate from the protocol in any desired matter. The number of actually corrupted parties is denoted by $f$.

The parties are connected by a network of authenticated point-to-point channels. Our protocol consists of two phases – a synchronous and a subsequent asynchronous phase.

In the synchronous phase, all communication is synchronous – there is a common clock and the message delay in the network is upper-bounded by a known constant. The synchronous phase proceeds in rounds, and every message sent in some round is delivered at the beginning of the next round.

In the asynchronous phase, the messages can be delayed arbitrarily and the order of the messages does not have to be preserved (however every sent message is eventually delivered). In this phase, the computation proceeds in steps. In every step one party is active – it is activated by receiving a message, then it performs some local computation and eventually sends out some messages. To model the worst case scenario of asynchronous communication, we give the power to schedule the message delivery to the adversary – he can choose in every step which of the messages in the network is to be delivered.

Our protocol can be proved static secure in the UC framework [Can01]. We conjecture that using the techniques that were used by [DN03] in order to present an adaptively UC-secure version of [CDN01], our protocol can be modified to be adaptively secure in the UC framework [Can01] as well. Since the focus of the present paper is not on adaptive security the considerable complications needed to obtain adaptive security would, however, only blur the focus of the paper.

### 2.2 Threshold Signatures

We use a threshold signature scheme with threshold $t$, where $t < n/2$ is the maximal number of corrupted parties the protocol is meant to tolerate. There is a publicly known *verification key* $vk$ and a secret *signature key* $sk$. Each $P_i$ holds a *signature key share* $sk_i$. Given a message $m$ the party $P_i$ can compute a *signature share* $\sigma_i$ on $m$ and can prove to any other party, using a two-party zero-knowledge protocol, that $\sigma_i$ is a correct signature share on $m$. Given the verification key $vk$ and $t+1$ distinct, correct signature shares, anyone can compute a signature $\sigma = \sigma_{sk}(m)$. The system is unforgeable by a poly-time adversary knowing up to $t$ of the shares $sk_i$ – it takes a signature share from at least

one honest party to create a valid signature on $m$ under $vk$. The system from [Sho00] meets these requirements.[6]

### 2.3 Threshold Homomorphic Encryption

Our protocols will use ideas from [HNP05], which uses threshold homomorphic encryption to implement asynchronous MPC. One possible instantiation of threshold homomorphic encryption is using Paillier's encryption system [Pai99] as used in [CDN01]. The details are described in [HNP05], but all we need for the level of discussion in this extended abstract is the following.

*Threshold Decryption.* In a threshold system, with threshold $t$, there is an *encryption key* $ek$ and a *decryption key* $dk$. The encryption key is known by all parties and $dk$ is shared among the parties, with each $P_i$ holding a share $dk_i$. Given $ek$, a *plaintext* $x$ and a *randomizer* $r$ anyone can compute a *ciphertext* $X = E_{ek}(x; r)$. For a ciphertext $X$ each $P_i$ can compute a *decryption share* $X_i$ and can prove, using a two-party zero-knowledge protocol, that $X_i$ is a correct decryption share. Given the encryption key $ek$ and $t + 1$ correct decryption shares, from different parties, anyone can compute the plaintext $x = D_{dk}(X)$. The system is IND-CPA secure against an adversary knowing $t$ of the shares $dk_i$.

If such a system has been setup and $t < n/2$, then any party $P_d$, which is allowed to, can decrypt a ciphertext $X$ asynchronously: The party sends $X$ to all parties. Each party which agrees that $P_d$ is allowed to decrypt $X$ sends a decryption share to $P_d$ and proves to $P_d$ that the decryption share is correct. The party $P_d$ waits for $n - t$ shares to arrive for which valid proofs were provided. Since there are $n - t$ honest parties, if $P_d$ is allowed to decrypt $X$, then this is deadlock free. And, since $n - t \geq t + 1$, $P_d$ gets enough shares to compute $x = D(X)$.

*Homomorphic Encryption.* We assume that the encryption is homomorphic modulo some publicly known integer $N$. There exists some operation $\boxplus$ on ciphertexts such that $E_{ek}(x; r) \boxplus E_{ek}(y; s) = E_{ek}(x + y \bmod N; t)$ for some randomizer $t$, which can be computed efficiently from $x, r, y, s$ if these are known. We also assume that given $X = E_{ek}(x; r)$ one can compute $X' = E_{ek}(N - x; s)$ for some randomizer $s$, which can be computed efficiently from $x$ and $r$. We assume that these operations can be performed efficiently given just the encryption key $ek$. We use $C \in E(x)$ to mean that there exists a randomizer $r$ such that $C = E_{ek}(x; r)$. Note that by combining the homomorphic properties one can take any ciphertext $B \in E(b)$ and any integer $a \in \mathbb{Z}_N$ and efficiently compute an encryption $C \in E(ab \bmod N)$ using double-and-add. We call this *multiplication by a constant* and write $C = a \boxdot B$. We also assume that it is possible to take any ciphertext $C \in E(c)$ and efficiently compute a *uniformly random* ciphertext $C' \in_R E(c)$, using just the encryption key. We write $C' \leftarrow [C]$ and call this *re-randomization*. We write $C' = [C](r)$ when we want to make explicit the randomness $r$ used for re-randomization. To guarantee robustness of some of our protocols we assume that there exists a concurrent, non-malleable zero-knowledge (ZK) proof which allows a party having computed $C' = [a \boxdot B](r)$ to prove to another party that it knows $a$ such that there exists $r$ such that $C' = [a \boxdot B](r)$ – the verifier is expected to know just $ek$, $B$ and $C'$.

### 2.4 Concurrent, Non-malleable ZK

All ZK proof mentioned above, and in the following sections, can be implemented as in [CDN01,HNP05] by transforming three-move, public-randomness, honest-verifier ZK proofs as described in [Dam00]. This yields concurrent, non-malleable ZK proofs for the common reference string model. The same transformation can be performed in the model where each party has a registered public key. In fact, one can prove that the resulting proof system is a static, universally composable

[6] The system from [Sho00] uses the random oracle model to be non-interactive. To avoid the random oracle we simply use interactive proofs, as described in e.g. [Nie02].

ZK proof of *membership* in this setting, and in addition a proof of *knowledge* when one is allowed to rewind the adversary[Nie03, Corollary 5.2].[7]

## 3 Protocol Overview

Our protocol follows the standard approach with homomorphic threshold encryption, along the lines of [FH96,CDN01,HNP05]. At the beginning, all parties distribute encryptions of their inputs. Then, the agreed function is evaluated gate-by-gate, where for each gate, an encryption of its value is computed. Finally, the value of the output gate(s) is decrypted using threshold decryption.

As we require only $t < n/2$, no agreement on whatsoever can be achieved (provably, with $t \geq n/3$ BA is impossible). Hence, without any synchronicity assumptions, the parties could not reach agreement on the encryptions of the inputs. We therefore employ one round of synchronous broadcast which allows to consistently and verifiably distribute the inputs. [8] Once the inputs are distributed, the body of the computation is performed fully asynchronously, without any further agreement, with $t < n/2$. This implies that the function to be evaluated must be *deterministic*. However, probabilistic functions can easily be computed by evaluating a deterministic function on the actual inputs and some additional random inputs provided by the parties. Furthermore, for the sake of simplicity we assume that the function has public outputs only. Also this restriction can easily be overcome by letting the parties input random pads that are XORed on their local outputs. Thus the function to be computed is deterministic with public outputs. It is expressed as an arithmetic circuit over $\mathbb{Z}_N$ – the plaintext space of the encryption scheme. Each $P_i$ has some input wires associated. Internal gates either add or multiply modulo $N$, and then there are some output wires which will hold a representation of the outputs.

In the following section, we describe the fully asynchronous MPC protocol with predistributed inputs. In the subsequent section, we describe the input stage when given a single round of synchronous broadcast. Finally, we discuss under which assumptions this broadcast round can be simulated.

## 4 Fully-Asynchronous MPC with $t < n/2$ and Predistributed Inputs

In this section we present a *fully asynchronous* MPC protocol which allows to distributively evaluate an agreed function on predistributed inputs. This protocol tolerates $t < n/2$ corrupted parties, which means (among others things), that Byzantine agreement cannot be achieved.

The function is evaluated in the usual gate-by-gate manner. Starting with the given input encryptions, the parties jointly compute encryptions of each intermediary value (one after the other), until eventually an encryption of the output is available and jointly decrypted (using threshold decryption). As asynchronous BA is not possible for $t < n/2$, agreement on the *encryptions* of intermediary values cannot be guaranteed (however, agreement on the intermediary values is possible, as they can be deterministically derived from the predistributed inputs).

We solve the issue of inconsistent views on encryptions by evaluating the whole circuit many times in parallel, once for every party, denoted as *king*. The other parties act as *slaves* and help the king evaluating his copy of the circuit. When the king is honest, then all slaves will have consistent views on all encryptions. When the king is faulty, inconsistencies will occur, but we will show that they do not violate privacy (by cheating the king learns either the correct output or some uniformly random value).

[7] We note that the fact that the proofs are only proofs of *knowledge* when *rewinding* is allowed is not a problem for the UC security, where the simulator is not allowed to rewind. The extractor will not be used by the UC simulator, but by the *analysis* of the UC simulator. This is by now a standard trick (cf. [DN03]). In Appendix B we sketch how our protocol is UC simulated using standard techniques. In the main text we focus on describing the new techniques specific to our protocol.

[8] It is clear that in a real-life network, like the Internet, this round of broadcast itself has to be simulated using synchronous point-to-point communication. We return to how this one round of broadcast can be implemented in a point-to-point network in the conclusions of this paper.

The protocol proceeds in two phases: In the *computation phase*, the circuit is evaluated $n$ times in parallel, once for every king. In the subsequent *termination phase*, the parties ensure that all parties have learned the output, and hence all programs can safely be stopped. Note that not necessarily all kings can (or must) finish their copy of the circuit; once $t+1$ kings have finished *with the same output*, then obviously this must be the correct output, and all parties adopt this value and stop.

### 4.1 Computation Phase

We assume that for every input wire, the parties have agreement on the ciphertext $X = E(x)$ of the input value $x$. Then every king $P_k$ runs (with the help of the other parties acting as slaves) his own circuit evaluation, learning an encryption of the output of every gate. Throughout the whole computation it holds that: whenever an honest party holds a ciphertext $X$ for the output wire of some gate, then indeed $x = D(X)$ is the correct value of that wire; however, we do not require that different slaves hold the same encryption $X$ of a wire when the king is faulty. To every gate a unique *gate id* $gid$ is assigned. In the following, we present the protocols for addition, output, and multiplication gates.

**Addition Gate:**
Whenever a slave $P_i$ of $P_k$ holds ciphertexts $X$ and $Y$ of the input wires of an addition gate $gid$, he computes $Z = X \boxplus Y$ as encryption of the output wire.

**Output Gates:**
Whenever a slave $P_i$ of $P_k$ holds a ciphertext $Z$ of an output gate $gid$, he sends to $P_k$ a decryption share of $Z$, and gives (interactive) proofs that the decryption share is correct for $Z$. Once $P_k$ holds a ciphertext $Z$ of the output gate $gid$, and receives $t+1$ valid decryption shares for this $Z$, he computes the output $z$ for gate $gid$.

**Multiplication Gates:**
For multiplication, first the slaves help the king to generate a random multiplication triple [Bea91]. This triple consists of two encrypted random factors and the corresponding encrypted product. The actual multiplication is then evaluated with help of this prepared triple.

Intuitively, the generation of the multiplication triple proceeds as follows: $P_k$ starts with the *initial triple* $(A_0, B_0, C_0) = \big(E(1;\epsilon), E(1;\epsilon), E(1;\epsilon)\big)$, where $\epsilon$ denotes some fixed agreed-upon randomness for encryption. Trivially, $(A_0, B_0, C_0)$ is a correct multiplication triple (though far from being random). Then, in turn for $j = 0, \ldots, t$, $P_k$ sends $(A_j, B_j, C_j)$ to some party, who randomizes it to $(A_{j+1}, B_{j+1}, C_{j+1}) = \big(A_j \boxplus E(u), B_j \boxplus E(v), C_j \boxplus E(uv) \boxplus (u \boxdot B_j) \boxplus (v \boxdot A_j)\big)$ for randomly chosen $u, v \in \mathbb{Z}_N$, and sends back to $P_k$ the new triple $(A_{j+1}, B_{j+1}, C_{j+1})$ along with a ZK proof that it was correctly generated. Clearly, $(A_{t+1}, B_{t+1}, C_{t+1})$ is still a correct multiplication triple. Furthermore, as $t+1$ parties have randomized the triple, at least one of them being honest, the resulting triple is a random multiplication triple.

We first present the protocol that allows a party $P_i$ to randomize a triple $(A_j, B_j, C_j)$ to $(A_{j+1}, B_{j+1}, C_{j+1})$, and get the new triple certified to be a correct $j$-th randomization for gate $gid$ by party $P_i$ for king $P_k$.

**Protocol RandomizeTriple:**

0. $P_i$ has input $P_k$, $gid$, $j$, and $(A_j, B_j, C_j)$.
1. $P_i$ picks uniformly random plaintexts $u, v \in_R \mathbb{Z}_N$ and computes $U \leftarrow E(u)$, $V \leftarrow E(v)$, $X \leftarrow [u \boxdot B_j]$, $Y \leftarrow [v \boxdot A_j]$ and $Z \leftarrow [u \boxdot V]$. It sends $(A_j, B_j, C_j)$ and $(U, V, X, Y, Z)$ to all parties and gives a concurrent, non-malleable ZK proof of knowledge to each party of:
   - $u$ such that $U \in E(u)$ and $X \in [u \boxdot B_j]$,
   - $v$ such that $V \in E(v)$ and $Y \in [v \boxdot A_j]$, and
   - $u$ such that $U \in E(u)$ and $Z \in [u \boxdot V]$.

2. Any $P \in \mathcal{P}$ receiving $(A, B, C)$ and $(U, V, X, Y, Z)$, along with accepting proofs, computes $A_{j+1} = A_j \boxplus U$, $B_{j+1} = B_j \boxplus V$, $C_{j+1} = C_j \boxplus X \boxplus Y \boxplus Z$, and sends a signature share on $\big((A_j, B_j, C_j), (P_k, gid, j, P_i), (A_{j+1}, B_{j+1}, C_{j+1})\big)$ to $P_i$.
3. $P_i$ waits for $t+1$ valid signature shares on $\big((A_j, B_j, C_j), (P_k, gid, j, P_i), (A_{j+1}, B_{j+1}, C_{j+1})\big)$, computes a signature $\sigma$, and outputs $[(A_j, B_j, C_j), (P_k, gid, j, P_i, \sigma), (A_{j+1}, B_{j+1}, C_{j+1})]$.

The following protocol allows the king (with help of the other parties) to generate a random multiplication triple (with gid $gid$). The idea is to start with an initial triple (i.e., encryption of $(1, 1, 1)$) and randomize it $t+1$ times – each time by a different party. For this the king first sends a randomization request for the initial triple to *every* party. Then he waits for the first correct answer and sends it as the second randomization request to all other parties (except the provider of the first randomization). Then again the first correct answer is used for the next randomization, etc. In every round, all but the first correct answers are ignored.

**Protocol GenerateTriple:**

0. $P_k$: Initialize $j = 0$ and $(A_0, B_0, C_0) = \big(E(1;\epsilon), E(1;\epsilon), E(1;\epsilon)\big)$.
1. For $j = 0$ to $t$ do
   1.1 Send a randomization request $[P_k, gid, j, (A_j, B_j, C_j)]$ to every party $P_i$ of whom no randomization for $gid$ has been stored so far.
   1.2 $P_i$: Upon receiving a randomization request $[P_k, gid, j, (A_j, B_j, C_j)]$, employ the protocol RandomizeTriple to obtain $[(A_j, B_j, C_j), (P_k, gid, j, P_i, \sigma), (A_{j+1}, B_{j+1}, C_{j+1})]$, and send it to $P_k$. This is performed only once per $gid$ and $j$.
   1.3 $P_k$: Upon receiving (from some party $P_i$ for which no randomization for $gid$ is stored so far) the first (correct) randomization answer $[(A_j, B_j, C_j), (P_k, gid, j, P_i, \sigma), (A_{j+1}, B_{j+1}, C_{j+1})]$, store this answer. Further answers from other parties (for the same $j$) are ignored.
2. $P_k$: Send $[(A_j, B_j, C_j), (P_k, gid, j, P_{i_j}, \sigma_j), (A_{j+1}, B_{j+1}, C_{j+1})]$ for $j = 0, \ldots, t$ to every $P_i \in \mathcal{P}$, who accepts $(A, B, C) = (A_{t+1}, B_{t+1}, C_{t+1})$ as the final multiplication triple for $gid$ if the following holds: For $j = 0, \ldots, t$, the $j$-th output triple is equal to $(j+1)$-th input triple, there are $t+1$ *different* parties that have randomized, and all transitions are correctly signed.

Given the multiplication triples $(A, B, C)$ from GenerateTriple, and given encryptions $X$ and $Y$ to be multiplied, the following protocol computes an encryption of the product $Z$.

**Protocol Multiply:**

0. Every $P_i$ has input $(A, B, C)$, $X$ and $Y$.
1. $P_i$: send to $P_k$ *and all slaves* decryption shares of $F = X \boxplus A$ and $G = Y \boxplus B$, and give proofs that the decryption shares are correct.
2. $P_i$ and $P_k$: If $t+1$ valid decryption shares for $F$ and $G$ arrive, compute $f = x + a \bmod N$ and $g = y + b \bmod N$ and let $Z = E(fg) \boxplus (-f \boxdot B) \boxplus (-g \boxdot A) \boxplus C$.

We first analyze the generation of the multiplication triple, then the multiplication protocol.

### 4.2 Analysis of Computation Phase

**Analysis of GenerateTriple:**

Although there is no agreement among the parties on the multiplication triple $(A, B, C)$ (as such an agreement cannot be achieved with $t \geq n/3$) we are given certain guarantees about the triple (except with negligible probability):

– When an honest slave $P_i$ accepts a triple $(A, B, C)$, then $A$ and $B$ are encryptions of values $a$ and $b$, and $C$ is an encryption of $ab$. Furthermore, the set of corrupted parties (the adversary) cannot distinguish $a$ and $b$ from uniformly random values. This is formalized by a game, where the

adversary is given $(a, b, c) = (D(a), D(b), D(c))$ or $(a, b, ab)$ for uniformly random $a, b \in \mathbb{Z}_N$, by requiring that it distinguishes with probability negligibly close to $\frac{1}{2}$. Correctness follows from the correctness of the initial triple and the proofs of correct randomization. The indistinguishability follows from the fact that $A$ and $B$ result from $t+1$ randomizations, so $A$ and $B$ were randomized by at least one honest party $P_i$. Therefore $a$ and $b$ sum over the $u_i$ respectively the $v_i$ contributed by $P_i$. Furthermore, every randomizing party $P_j$ proves knowledge of its randomizers $u_j$ and $v_j$ (using a concurrent, non-malleable proof of knowledge, see Section 2.4). Hence we can, by rewinding, extract the $u_j$ and $v_j$ from the view of the adversary. So, for the adversary, distinguishing $a$ and $b$ from uniformly random is equivalent to distinguishing $u_i$ and $v_i$ from uniformly random for at least one honest $P_i$, which is impossible by the semantic security of the cryptosystem and the proofs given by $P_i$ being concurrent zero-knowledge.

- When an honest slave $P_i$ accepts a triple $(A, B, C)$ for a gate $gid$, then the plaintexts of $A$ and $B$ are indistinguishable from uniformly random values which are *statistically independent* from the plaintexts of any triple accepted for any other gate $gid' \neq gid$. This does not follow from the above property which addresses the distribution of individual triples, but follows trivially from the fact that honest parties use different randomizers when contributing to different multiplication gates $gid$.
- When for the same multiplication gate $gid$, two honest parties accept the triples $(A, B, C)$ and $(A', B', C')$, respectively, then either the plaintexts of $(A, B, C)$ and $(A', B', C')$ are indistinguishable from uniformly random, statistically independent values to the adversary, or the adversary knows the plaintexts of $A - A'$ and $B - B'$. This follows from the fact that either there is at least one honest party $P_i$ that has randomized one triple in some position, but not the other one in the same position with the same $(u_i, v_i)$ (then the plaintexts of the two triples are indistinguishable from uniformly random statistically independent values), or both triples have been randomized by exactly the same set of honest parties $P_i$ in exactly the same positions with exactly the same $(u_i, v_i)$. In this case only the adversarially chosen randomizers are different, and they are known to the adversary in the sense that they can be extracted from the adversary in expected polynomial time.

We now argue termination. Note that as long as at most $t$ parties have randomized the triple, there are still $(n - t) - t \geq 1$ honest parties $P_i$ which did not yet do so and thus, when requested, will eventually produce a randomization for $gid$ and send it to $P_k$. Therefore, eventually a chain of $t + 1$ randomizations will be achieved.

**Analysis of Multiply:**
If $P_k$ is honest, then all slaves will constantly agree on all ciphertexts $X$ for each wire, and therefore the computation will terminate and will yield correct encryptions for all wires. When $P_k$ is corrupted we do not *per se* care about the correctness of $P_k$, so what remains is to argue privacy.

The first important observation is that if an honest slave $P_i$ associates $X$ to some wire, then $X$ is an encryption of the correct value for that wire. This holds for input wires by assumption and is maintained by addition. As for multiplication gate $z = xy$, we can assume that $X$ and $Y$ decrypt to correct values. If $(A, B, C)$ was accepted by $P_i$ as a correct triple, it is indeed a correct multiplication triple, except with negligible probability. From this it follows that if $P_i$ computes some $Z$, then $P_i$ computes a correct $Z$, except with negligible probability. Different parties might, however, hold different $Z$ if $P_k$ is corrupted – only the plaintexts are guaranteed to be the same.

We address the privacy. Assume that party $P_i$ holds encryptions $X^{(i)}$ and $Y^{(i)}$ of the factors, and gets the multiplication triple $(A^{(i)}, B^{(i)}, C^{(i)})$ from $P_k$. At the same time, $P_j$ holds encryptions $X^{(j)}$ and $Y^{(j)}$ of the factors, and gets the multiplication triple $(A^{(j)}, B^{(j)}, C^{(j)})$ from $P_k$. Then, $P_k$ might learn the decryptions $f^{(i)} = x^{(i)} + a^{(i)} \bmod N$ and $g^{(i)} = y^{(i)} + b^{(i)} \bmod N$ as well as the decryptions $f^{(j)} = x^{(j)} + a^{(j)} \bmod N$ and $g^{(j)} = y^{(j)} + b^{(j)} \bmod N$. However, by the invariant that the values

$X^{(i)}$ and $Y^{(i)}$ held by $P_i$ (and the values $X^{(j)}$ and $Y^{(j)}$ held by $P_j$) encrypt correct wire values $x$ and $y$, we have $x^{(i)} = x^{(j)} = x$ and $y^{(i)} = y^{(j)} = y$. Furthermore, from $(A^{(i)}, B^{(i)}, C^{(i)})$ and $(A^{(j)}, B^{(j)}, C^{(j)})$ being correct multiplication triples for the same gate $gid$, it follows that either 1) they encrypt values $(a^{(i)}, b^{(i)})$ and $(a^{(j)}, b^{(j)})$ which are uniformly random and independent, or 2) they encrypt values $(a^{(i)}, b^{(i)})$ and $(a^{(j)}, b^{(j)})$ which are individually uniformly random and $(a^{(j)}, b^{(j)}) = (a^{(i)}, b^{(j)}) + (\delta_a, \delta_b)$ for $(\delta_a, \delta_b)$ *known*[9] to the adversary. In the first case, $(f^{(i)}, g^{(i)})$ and $(f^{(j)}, g^{(j)})$ are uniformly random and independent and thus together leak no information to the adversary. In the second case, $(f^{(i)}, g^{(i)})$ is uniformly random, and therefore leaks no information to the adversary, and $(f^{(j)}, g^{(j)}) = (f^{(i)}, g^{(i)}) + (\delta_a, \delta_b)$ and therefore leaks no more information than $(f^{(i)}, g^{(i)})$ to the adversary, as the adversary can compute it from $(f^{(i)}, g^{(i)})$ in expected poly-time.

In Appendix B we sketch how the above properties are used in the analysis of a UC simulation.

### 4.3 Termination Phase

As for now no party can terminate until it knows that all honest parties for which it acts as slave terminated. However, this condition cannot be checked. Instead, we add the following simple procedure inspired by [CKS00] to terminate the protocol: When a king $P_k$ learns the result $z$, it sends a signature share on $(\text{“}\texttt{result}\text{”}, z)$ to all parties and continues to act as slave. When it received signature shares from $t+1$ parties on $(\text{“}\texttt{result}\text{”}, z)$, it constructs a signature $\sigma$ on $(\text{“}\texttt{result}\text{”}, z)$, sends $((\text{“}\texttt{result}\text{”}, z), \sigma)$ to all parties and terminates with output $z$. Any party ever receiving a value of the form $(\text{“}\texttt{result}\text{”}, z, \sigma)$ where $\sigma$ is a valid signature on $(\text{“}\texttt{result}\text{”}, z)$ sends it to all parties, and terminates with output $z$. Eventually all $n - t \geq t + 1$ honest $P_k$ learn $z$ and thus some honest party eventually receives $t+1$ correct signature shares. After this all honest parties will eventually terminate.

## 5 Almost-Asynchronous Input-Distribution with $t < n/2$

In this section, we describe how the inputs can be distributed. Note that in a fully asynchronous network, input-guarantee is not possible (up to $t$ possibly honest inputs are ignored). Furthermore, when $t \geq n/3$, input-distribution (even of a subset of parties) is impossible, as consistent distribution of a single input implies BA. In the following, we show how *all inputs* can be distributed with $t < n/2$ with a single synchronous broadcast round. See Section 6 for a discussion of implementing this one round of broadcast.

In the input phase, every party $P_i$ computes $X = E(x; r)$ for each of its inputs $x$ and broadcasts $X$ along with a ZK proof of plaintext knowledge (*PoPK*).[10] This ensures *input correctness*, in the sense that if $P_i$ is honest, then $D_{dk}(X) = x$, and *input privacy*, in the sense that as long as at most $t$ parties are corrupted, the input $x$ of an honest $P_i$ remains unknown to the adversary. This follows from the threshold IND-CPA security of the encryption scheme and the PoPK being ZK. Finally, the ZK proof of *knowledge* of $x$ ensures *input knowledge*, meaning that $P_i$ knows $D_{dk}(X)$ for his $X$.

The proof of plaintext knowledge could be based on standard assumptions by resorting to generic NIZK. In the following, we give a much more efficient proof, which exploits the fact that the proofs do not need to be fully non-interactive, but asynchronous interaction (with $t < n/2$) is allowed for verifying the proof. We call such proofs *almost non-interactive proofs*.

The intuition of our almost non-interactive proof is the following: The prover sends along with the encrypted input a *transcript* of many instances of an interactive zero-knowledge proof of plaintext knowledge with binary challenges. For each instance, the prover provides the answers for *both challenges*, but encrypts them with the threshold encryption scheme. To verify the proof, for each instance exactly one response (depending on an agreed-upon challenge) is decrypted. The challenge is generated simultaneously, by letting every party $P_i$ broadcast an encryption $R_i$ of a random value $r_i$, where

[9] In the sense that we can extract them from the adversary in expected poly-time.

[10] The details of the PoPK are given below.

the encryption scheme has the property that both the parties jointly as well as $P_i$ alone can decrypt.[11] Then the parties decrypt all contributions and compute the challenge $r$ as the sum.

In the next section, we describe how to generate the random challenge (using almost non-interactive VSS). Subsequently, we describe in more detail how to construct the almost non-interactive zero-knowledge proof of plaintext knowledge.

### 5.1 Almost Non-Interactive VSS

The following protocol allows a sender $P_S$ to verifiably secret share a secret $x$ with threshold $t < n/2$, using a single round of synchronous broadcast. The reconstruction of the shared value is fully asynchronous. We call this *almost non-interactive VSS* (*ANI-VSS*).

The ANI-VSS requires a setup – for every $P_S$ there is an independent random key pair $(pk_S, sk_S)$ for a threshold cryptosystem such that the public key $pk_S$ is known to all parties and the secret key $sk_S$ is shared among the other parties with threshold $t$ (such that correct decryption shares from $t+1$ parties are enough to decrypt under $sk_S$). We also require that $P_S$ knows $sk_S$. If not already the case, this can be ensured by all parties once-and-for-all sending their shares of $sk_S$ to $P_S$.[12] The protocol proceeds as follows:

**Synchronous sharing:** $P_S$ computes $X \leftarrow E_{pk_S}(x)$ and broadcasts $X$ (using synchronous broadcast).
**Asynchronous reconstruction:**

1. Each $P_i$ computes a decryption share of $X$ using his share of $sk_S$ and sends the share to all parties along with a proof of correctness.
2. Each $P_j$ waits for $t+1$ correct decryption shares and reconstructs $x = D_{sk_S}(X)$.

**Analysis:**

We assume that the encryption schemes have perfect decryption. This means that the broadcasted message $X$ uniquely defines a secret $x = D_{sk_S}(X)$. Reconstruction will always terminate as at least the $n - t \geq t + 1$ honest parties send correct decryption shares.

In terms of simulation security, an ANI-VSS is extracted by decrypting $X$. This is possible as the honest parties hold enough decryption key shares to compute $sk_S$. When $P_S$ is honest, an ANI-VSS is opened to any $x'$ simply by simulating the decryption of $X$ to hit $x'$.

### 5.2 Almost Non-Interactive ZKPoK

We now describe a system which allows a prover $P$ to give a ZK proof of knowledge (*ZKPoK*) towards all parties such that all parties agree on the outcome of the proof. The protocol uses only one round of synchronous broadcast, followed by an asynchronous computation. We call it an *ANI-ZKPoK*.

We consider some NP relation $R$ and assume that $P$ holds an (instance, witness)-pair $(x, w)$. We assume that there is a standard three-move $\Sigma$-protocol for $R$, where $P$ computes the first message $a$, gets a challenge $e \in \{0, 1\}$ and replies with some response $z$. The verifier accepts or rejects based on $(x, a, e, z)$. We use that from two accepting conversations $(x, a, 0, z_0)$ and $(x, a, 1, z_1)$ one can compute (in PPT) a witness $w$ such that $(x, w) \in R$. The protocol proceeds as follows:

**Synchronous proof:** The synchronous round proceeds as follows:

- Prover $P$: For $k = 1, \ldots, \kappa$, compute a first message $a^{(k)}$ and a reply $z_0^{(k)}$ to the challenge $e = 0$ and a reply $z_1^{(k)}$ to the challenge $e = 1$. Then broadcast $x$ and each $a^{(k)}$ and ANI-VSS each $z_0^{(k)}$ and $z_1^{(k)}$.
- Each other party $P_i$: ANI-VSS a uniformly random value $r_i \in \{0, 1\}^\kappa$.

**Asynchronous verification:** The verification of the proof is asynchronous, and proceeds as follows:

1. Reconstruct each $r_i$ and compute $(e_1, \ldots, e_\kappa) = \oplus_{i=1}^n r_i$.
2. For $k = 1, \ldots, \kappa$ in parallel: Reconstruct $z_{e_k}^{(k)}$ and accept the proof iff $(a^{(k)}, e_k, z_{e_l}^{(k)})$ is an accepting conversation for $k = 1, \ldots, \kappa$.

[11] This way no PoPK for $R_i$ is necessary.

[12] In fact, the fact that they *could* do this is sufficient for the analysis.

**Analysis:**
After the first (synchronous) part, all parties will hold consistent proof transcripts $(a^{(1)}, Z_0^{(1)}, Z_1^{(1)}), \ldots, (a^{(\kappa)}, Z_0^{(\kappa)}, Z_1^{(\kappa)})$ (as broadcasted by the prover) as well as consistent encryptions of challenge-contributions $R_i \in \{0,1\}^\kappa$ of every party $P_i$ (as broadcasted by $P_i$). It follows that in the asynchronous part all parties will reconstruct the same $r_1, \ldots r_n$ leading to the same $(e_1, \ldots, e_\kappa)$ and thus leading to the same outcome of the verification test. It is clear that if the prover is honest, this outcome will be accepting. Since each reconstruction eventually terminates, the proof eventually terminates.

Assume that $P$ broadcasted $(a^{(1)}, Z_0^{(1)}, Z_1^{(1)}), \ldots, (a^{(\kappa)}, Z_0^{(\kappa)}, Z_1^{(\kappa)})$ without knowing a witness for $x$. Then for each $k$ there exists $e'_k$ such that $(a^{(k)}, e'_k, z_{e'_k}^{(k)})$ is not accepting. Thus there is at most one challenge $e = (e_1, \ldots, e_\kappa)$ for which the verification of the proof is not rejecting, namely $(1 - e'_1, \ldots, 1 - e'_\kappa)$. As the prover had to choose and broadcast $(a^{(1)}, Z_0^{(1)}, Z_1^{(1)}), \ldots, (a^{(\kappa)}, Z_0^{(\kappa)}, Z_1^{(\kappa)})$ without knowing the $r_i$'s of the honest parties (and thus without knowing the resulting challenge $e$), his success probability is negligible.

To simulate a proof, the $r_i$ contributed by corrupted parties are extracted from the ANI-VSS's. Then any challenge $e$ can be hit simply by opening the ANI-VSS for an honest $P_j$ to $r_j = e \oplus \bigoplus_{i \neq j} r_i$.[13]

## 6 Almost-Asynchronous Broadcast

In Section 5, we have employed synchronous broadcast for distributing the inputs. This single round of synchronous broadcast can be simulated using synchronous rounds of point-to-point communication. We now explore what is the number of synchronous rounds of point-to-point communication needed to simulate one round of broadcast using known techniques.

### 6.1 The $t < n/3$ Case:

When $t < n/3$, then just one synchronous point-to-point round is needed. In this synchronous round, the sender $P_S$ sends $x$ (the value he wants to broadcast) to all parties. Then the parties run an asynchronous consensus protocol (cf. [Bra84,Ben83,CR93,ADH08]) on $x$, where each $P_i$ inputs the value it received from $P_S$. The output of this consensus is taken as the broadcasted value. The parties wait until all the broadcasts of inputs, and other values, completed before they start running the rest of the asynchronous phase, to ensure that all corrupted parties has to decide on their broadcasted value before any of the other computations are performed.

### 6.2 The $t < n/2$ Case:

When $t < n/2$, Byzantine agreement is not possible using a fully asynchronous protocol, and therefore another approach must be taken. An easy solution is to use a fully synchronous broadcast protocol for $t < n/2$. Classical broadcast protocols [PSL80,LSP82,DS82,FL82,MT86,GM93] require $t + 1$ rounds of synchronous point-to-point communication, and this is known to be optimal if the parties are required to terminate in the same round. When stopping in different rounds is allowed, the number of communication rounds can be reduced to $\min(t + 1, f + 2)$, where $f$ is the actual number of corrupted parties in the execution and $t$ is the maximal number of corruptions tolerated, and this is optimal [DRS90,BGP92]. Note that the fact that parties can (and will) terminate in different rounds usually requires relatively expensive resynchronization efforts[LLR02][Nie03, Chapter 7], but that this is not an issue in our setting: we go asynchronous after the broadcast anyway, making the resyncronization unnecessary.

Even more interestingly, it was recently shown in [FN08] that one round of synchronous broadcast can be simulated by some initial number $\tau$ of synchronous rounds of point-to-point communication

[13] For the ANI-VSS described above, the extraction would simply amount to decryption under the secret key used by $P_i$. The honest parties have enough shares to facilitate this.

followed by asynchronous point-to-point communication. The number $\tau$ beats the $t+1$ bound on the worst-case number of rounds needed when insisting on using a broadcast protocol where all rounds are synchronous. It is essentially $\tau = t/2$ when $t = n/2$ and goes linear to $\tau = 1$ when $t = n/3$.

Another approach is to use a probabilistic broadcast protocol, which allows *expected* constant round complexity for $t < n/2$ [Rab83,Tou84,FM88]. The practical implications are however questionable, as the expected round complexities of all known randomized protocols are fairly large constants.

## 7 Conclusions

We presented the first multi-party protocol for a non-synchronous network which tolerates $t < n/2$ faulty parties. The protocol achieves all properties of synchronous MPC in an almost-asynchronous network. With one single round of synchronous broadcast, both input-guarantee and security for $t < n/2$ can be achieved. This contrasts fully-asynchronous MPC protocols which ignore the inputs of up to $t$ honest parties and cannot guarantee security better than $t < n/3$.

**Theorem 1.** *When $t < n/2$, then any PPT function can be computed without input deprivation using* $1$ *synchronous broadcast round.*

By simulating the synchronous broadcast round by point-to-point communication, we obtain the following result:

**Corollary 1.** *When $t < n/2$, then any PPT function can be computed without input deprivation using* $\min(\tau, r_n)$ *synchronous* point-to-point *rounds, where $\tau \leq t/4 + 2$ was defined in Section 6 and $r_n$ denotes the expected round complexity of a synchronous constant-round broadcast protocol among $n$ parties.*

By simulating the synchronous broadcast round by asynchronous consensus (which requires $t < n/3$), we obtain the following result:

**Corollary 2.** *When $t < n/3$, then any PPT function can be computed without input deprivation using* one *synchronous* point-to-point *round.*

### 7.1 Some Open Problems

On the theoretical side it would be interesting to find out what are the minimal setup assumptions needed for our result. The UC model is known to require some setup assumption, but can one reproduce our result in the model from [Can00] without using setup? Can our result be reproduced under more general computational assumptions?

As described in Appendix A, the communication complexity of our protocol can be made as low as $\mathcal{O}(n^3\kappa)$ bits per gate using a few simple optimizations. This is, however, far from the complexity of the currently best synchronous protocols. It would be interesting to get an efficiency of "almost-asynchronous MPC", which can compete with the currently best fully synchronous protocols.

Our protocols in this paper recently inspired an efficient implementation of secure MPC in an "almost-asynchronous" model, see [DGKN08]. The security guarantees are lower than what we get in this paper, but we hope that [DGKN08] is an indication that the approach we introduce here can lead to highly secure and highly efficient MPC for realistic networks.

It is also an interesting open problem whether our result can be obtained using a protocol which defaults to the normal asynchronous security guarantees if the initial synchrony assumption fails — i.e., the protocol still guarantees security against $t' < n/3$ corrupted parties but might deprive $t'$ parties of giving inputs.

## References


[ADH08] Ittai Abraham, Danny Dolev, and Joseph Y. Halpern. An almost-surely terminating polynomial protocol forasynchronous byzantine agreement with optimal resilience. In Rida A. Bazzi and Boaz Patt-Shamir, editors, *PODC*, pages 405–414. ACM, 2008.

[BB89] Judit Bar-Ilan and Donald Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In *Proc. ACM PODC'89*, pages 201–209, 1989.

[BCG93] Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation (extended abstract). In *Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing*, pages 52–61, San Diego, California, 16–18 May 1993.

[Bea89] Donald Beaver. Multiparty protocols tolerating half faulty processors. In Gilles Brassard, editor, *Advances in Cryptology - Crypto '89*, pages 560–572, Berlin, 1989. Springer-Verlag. Lecture Notes in Computer Science Volume 435.

[Bea91] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Advances in Cryptology - Crypto '91*, pages 420–432, Berlin, 1991. Springer-Verlag. Lecture Notes in Computer Science Volume 576.

[Ben83] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30, 1983.

[BGP92] Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Optimal early stopping in distributed consensus. In *Proceedings of the sixth International Workshop on Distributed Algorithms*, pages 221–237, 1992.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, pages 1–10, Chicago, Illinois, 2–4 May 1988.

[BH07] Zuzana Beerliová-Trubíniová and Martin Hirt. Simple and efficient perfectly-secure asynchronous mpc. In *Advances in Cryptology - ASIACRYPT 2007*, Berlin, 2007. Springer. Lecture Notes in Computer Science.

[BKR94] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computation with optimal resilience. In *Proc. ACM PODC'94*, pages 183–192, 1994.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *Proceedings of the Twenty Second Annual ACM Symposium on Theory of Computing*, pages 503–513, Baltimore, Maryland, 14–16 May 1990.

[Bra84] Gabriel Bracha. An asynchronous $\lceil (n-1)/3 \rceil$-resilient consensus protocol. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 154–162, 1984.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, winter 2000.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, Nevada, 14–17 October 2001. IEEE.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, pages 11–19, Chicago, Illinois, 2–4 May 1988.

[CDG87] David Chaum, Ivan Damgård, and Jeroen van de Graaf. Multiparty computations ensuring privacy of each party's input and correctness of the result. In Carl Pomerance, editor, *Advances in Cryptology - Crypto '87*, pages 87–119, Berlin, 1987. Springer-Verlag. Lecture Notes in Computer Science Volume 293.

[CDN01] Ronald Cramer, Ivan Damgaard, and Jesper Buus Nielsen. Multiparty computation from threshold homomorphic encryption. In *Advances in Cryptology - EuroCrypt 2001*, pages 280–300, Berlin, 2001. Springer-Verlag. Lecture Notes in Computer Science Volume 2045.

[CKS00] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous Byzantine agreement using cryptography. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC 2000)*, pages 123–132. ACM, July 2000.

[CR93] Ran Canetti and Tal Rabin. Fast asynchronous Byzantine agreement with optimal resilience (extended abstract). In *Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing*, pages 42–51, San Diego, California, 16–18 May 1993.

[Dam00] Ivan Damgård. Efficient concurrent zero-knowledge in the auxiliary string model. In *EUROCRYPT*, pages 418–430, 2000.

[DGKN08] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. Technical Report 496, Cryptology ePrint Archive, 2008.

[DN03] Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In D. Boneh, editor, *Advances in Cryptology - Crypto 2003*, pages 247–264, Berlin, 2003. Springer-Verlag. Lecture Notes in Computer Science Volume 2729.

[DRS90] Danny Dolev, Ruediger Reischuk, and H. Raymond Strong. Early stopping in Byzantine agreement. *ACM Transactions on Programming Languages and Systems*, 37(4):720–741, October 1990.

[DS82] Danny Dolev and Raymond H. Strong. Polynomial algorithms for multiple processor agreement. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pages 401–407, San Francisco, CA, 5–7 May 1982.

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

[FL82] Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, June 1982.

[FM88] Paul Feldman and Silvio Micali. Optimal algorithms for Byzantine agreement. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, pages 148–161, Chicago, Illinois, 2–4 May 1988.

[FN08] Matthias Fitzi and Jesper Buus Nielsen. On the number of synchronous rounds required for byzantine agreement. Technical Report 495, Cryptology ePrint Archive, 2008.

[GIKR02] Rosario Gennaro, Yuval Ishai, Eyal Kushilevitz, and Tal Rabin. On 2-round secure multiparty computation. In M. Yung, editor, *Advances in Cryptology - Crypto 2002*, pages 178–193, Berlin, 2002. Springer-Verlag. Lecture Notes in Computer Science Volume 2442.

[GM93] Juan A. Garay and Yoram Moses. Fully polynomial Byzantine agreement in $t + 1$ rounds. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing*, pages 31–41, San Diego, California, 16–18 May 1993.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 218–229, New York City, 25–27 May 1987.

[HNP05] Martin Hirt, Jesper Buus Nielsen, and Bartosz Przydatek. Cryptographic asynchronous multi-party computation with optimal resilience (extended abstract). In R.Cramer, editor, *Advances in Cryptology - EuroCrypt 2005*, pages 322–340, Berlin, 2005. Springer-Verlag. Lecture Notes in Computer Science Volume 3494.

[IK00] Yuval Ishai and Eyal Kushilevitz. Randomizing polynomials: A new representation with applications to round-efficient secure computation. In *41st Annual Symposium on Foundations of Computer Science*, pages 294–304, Redondo Beach, California, 12–14 November 2000. IEEE.

[KK06] Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. In Cynthia Dwork, editor, *CRYPTO*, volume 4117 of *Lecture Notes in Computer Science*, pages 445–462. Springer, 2006.

[KK07] Jonathan Katz and Chiu-Yuen Koo. Round-efficient secure computation in point-to-point networks. In Moni Naor, editor, *EUROCRYPT*, volume 4515 of *Lecture Notes in Computer Science*, pages 311–328. Springer, 2007.

[KKK08] Jonathan Katz, Chiu-Yuen Koo, and Ranjit Kumaresan. Improving the round complexity of vss in point-to-point networks. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP (2)*, volume 5126 of *Lecture Notes in Computer Science*, pages 499–510. Springer, 2008.

[LLR02] Yehuda Lindell, Anna Lysyanskaya, and Tal Rabin. Sequential composition of protocols without simultaneous termination. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 203–212. ACM Press, 2002.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):381–401, July 1982.

[MT86] Yoram Moses and Mark R. Tuttle. Programming simulataneous actions using common knowledge. In *27th Annual Symposium on Foundations of Computer Science*, pages 208–221, Toronto, Ontario, Canada, 27–29 October 1986. IEEE.

[Nie02] Jesper Buus Nielsen. A threshold pseudorandom function construction and its applications. In M. Yung, editor, *Advances in Cryptology - Crypto 2002*, pages 401–416, Berlin, 2002. Springer-Verlag. Lecture Notes in Computer Science Volume 2442.

[Nie03] Jesper Buus Nielsen. On protocol security in the cryptographic model. Technical Report DS-03-8, August 2003.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residue classes. In Jacques Stern, editor, *Advances in Cryptology - EuroCrypt '99*, pages 223–238, Berlin, 1999. Springer-Verlag. Lecture Notes in Computer Science Volume 1592.

[PSL80] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[Rab83] Michael O. Rabin. Randomized Byzantine generals. In *23th Annual Symposium on Foundations of Computer Science*, pages 403–409, Los Alamitos, CA, 1983. IEEE.

[RB89] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, pages 73–85, Seattle, Washington, 15–17 May 1989.

[Sho00] Victor Shoup. Practical threshold signatures. In Bart Preneel, editor, *Advances in Cryptology - EuroCrypt 2000*, pages 207–220, Berlin, 2000. Springer-Verlag. Lecture Notes in Computer Science Volume 1807.

[Tou84] Sam Toueg. Randomized Byzantine agreements. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 163–178, 1984.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science*, pages 160–164, Chicago, Illinois, 3–5 November 1982. IEEE.

## A Efficiency Considerations

In this section we analyze the communication complexity of the protocol. In the analysis we measure the total number of bits sent by the honest parties. We let $c_I$ ($c_M$, $c_O$) be the number of input (multiplication, output) gates in the circuit.

During the analysis we give a few optimizations. We excluded these from the main text to keep the focus on that which is essentially new, namely that one can obtain corruption threshold $t < n/2$ and guarantee inputs from all parties using a single synchronous round.

We assume that the size of signature shares, proofs and signatures are $\mathcal{O}(\kappa)$ bits, where $\kappa$ is the security parameter. The tools we use from [Pai99,Sho00,HNP05] meet these requirements.

### A.1 Circuit Evaluation

The linear gates require no communication. When ignoring the generation of the triples, each multiplication gate requires two decryptions and each output gate requires one decryption. Each decryption communicates $\mathcal{O}(n^2\kappa)$ bits, as all parties send decryption shares to all parties and give proofs to all parties. So, each gate handled by $P_k$ communications $\mathcal{O}(n^2\kappa)$ bits. Each gate is handled by each of the $n$ kings, for a total of $\mathcal{O}(n^3\kappa)$ bits per gate. Ignoring the generation of triples, the evaluation of the circuit thus communications $\mathcal{O}((c_M + c_O)n^3\kappa)$ bits.

### A.2 Triple Generation

The communication of triple generation (per king $P_k$ and per triple $gid$) can be seen to be $\mathcal{O}(n^3\kappa)$ bits – each randomization communicates $\mathcal{O}(n\kappa)$ bits, and a total of $\mathcal{O}(n^2)$ randomizations are needed to create a triple. Exploiting that a king can generate many triples in parallel, the communication per generated triple can, however, be brought down to an amortized $\mathcal{O}(n^2\kappa)$: The king contacts all $n$ parties with different, incomplete $gid$, and waits for $n - t$ parties to return valid randomizations. Thereby $n$ randomizations give rise to $n - t \geq t + 1$ triples $gid$ coming one randomization closer to completion. Since each $gid$ needs $t + 1$ randomizations to be complete, on average one $gid$ is completed in each round of $n$ randomizations. Since each of the $n$ kings need one triple per multiplications gate, the total communication complexity of generating triples is $\mathcal{O}(c_M n^3\kappa)$ bits.

### A.3 Input

In the input stage each input encryption is broadcast along with a ANI-ZKPoK of the plaintext. The broadcast of a $\kappa$-bit value can be implemented with communication complexity $\mathcal{O}(n^3\kappa)$ [DS82], so the communication complexity of the broadcasts can be $\mathcal{O}(c_I n^3\kappa)$ bits.

**ANI-ZKPoK:**

In our setting, we only prove knowledge of relations for which we have $\Sigma$-protocols where all messages have length $\mathcal{O}(\kappa)$. The communication complexity of the ANI-ZKPoK protocol is therefore that of ANI-VSS'ing $\mathcal{O}(\kappa + n)$ values of length $\mathcal{O}(\kappa)$, which is the same as broadcasting $\mathcal{O}(\kappa + n)$ values of length $\mathcal{O}(\kappa)$.

When $\ell$ proofs are performed in parallel, one value $r = \oplus_{i=1}^{n} r_i$ can be expanded into $\ell$ challenges $(e^{(1)}, \ldots, e^{(\ell)})$ using a pseudo-random generator. In that case $\ell$ proofs use $\mathcal{O}(\ell\kappa + n)$ ANI-VSS's. We will always have $\ell > n$, meaning that the amortized number of ANI-VSS's used per proof is $\mathcal{O}(\kappa)$. Each ANI-VSS broadcasts $\kappa$ bits and communicates an additional $\mathcal{O}(n^2\kappa)$ bits. Each proof therefore broadcasts $\mathcal{O}(\kappa^2)$ bits and communicates an additional $\mathcal{O}(n^2\kappa^2)$ bits.

We can reduce the factor $\kappa^2$ to $\kappa$ as follows. We only use the ANI-ZKPoK to prove plaintext knowledge for the homomorphic encryption function, which comes to our help. Assume that we prove $\kappa$ instances in parallel, and note that by running the basic one-bit challenged proof for each instance not $\kappa$ times but only some constant $c$ times, the amortized communication per proof instance can be brought down to broadcasting $\mathcal{O}(\kappa c) = \mathcal{O}(\kappa)$ bits and sending an additional $\mathcal{O}(n^2\kappa c) = \mathcal{O}(n^2\kappa)$

bits. Since $\mathcal{O}(\kappa)$ bits can be broadcast with communication $\mathcal{O}(n^3\kappa)$, the amortized communication per input can be $\mathcal{O}(n^3\kappa)$.

The price to pay is that each proof has knowledge error $2^{-c}$. By setting the constant $c$ appropriately and using a Chernoff bound, it can, however, be guaranteed that the probability that $\kappa/2$ false instances pass the proof is negligible. For some party $P$ to prove knowledge of $k = \kappa/2 - 1$ plaintexts $(x_1, \ldots, x_k)$ encrypted by $(X_1, \ldots, X_k)$, the linear homomorphic properties are used to expand $(X_1, \ldots, X_k)$ into $(Y_1, \ldots, Y_\kappa)$ such that all parties can verify that $(Y_1, \ldots, Y_\kappa)$ decrypts to $(y_1, \ldots, y_\kappa)$ which are an encoding of $(x_1, \ldots, x_k)$ allowing efficient reconstruction given at most $k/2$ loses – one can e.g. use a Reed-Solomon code. Then $P$ proves plaintext knowledge of the $y_i$ and will fail if it does not know at least $k/2$ of them. This, however, implies that it can compute $(x_1, \ldots, x_k)$ efficiently.

### A.4 Overall Communication Analysis

Summing the above complexities gives a communication complexity of $\mathcal{O}((c_I + c_M + c_O)n^3\kappa)$ bits.

## B Sketch of UC Simulation

It this section we sketch how the security properties argued in the main text imply a UC simulation using known techniques from [CDN01,DN03]. The UC simulator runs the protocol honestly, with the following changes:

- Since it does not know the input of the honest parties, it uses $x' = 0$ as plaintext in the encryption $X$ to all input wires for which an honest party is to provide the input.
- After the corrupted parties broadcasted their encryptions $X$ for their input wires along with correct PoPKs, the simulator decrypts $X$ to get the plaintext $x = D_{dk}(X)$ and gives $x$ to the ideal functionality as the input of the corrupted party. This is possible as it is the simulator which simulates the setup phase, which allows it to learn $dk$.
- After providing the inputs on behalf of the corrupted parties, the simulator gets the result $y$ from the ideal functionality, where $y$ is a function of the inputs of the honest parties and the inputs $x$ chosen by the simulator for the corrupted parties. Since the simulator used $x' = 0$ for the honest parties in the simulation, the output ciphertext $Y$ corresponding to $y$ in the simulation is most likely not going to be an encryption of $y$.[14] Since we only want static security, this can be dealt with as in [CDN01]: The simulator simply cheats with the decryption shares of the honest parties to make $Y$ wrongly decrypt to $y$. It then simulates the ZK proofs that these decryption shares are correct. Below we call this a simulated decryption of $Y$ to $y$.

What remains is simply to argue that this UC simulation is indistinguishable from the real protocol (to the environment). This can be done along the lines of [CDN01]: First we show that replacing the wrong inputs $x' = 0$ by the correct inputs[15] $x$ in the encryptions $X$ used by the honest parties produces a distribution which is indistinguishable. This follows from the IND-CPA security of the cryptosystem.

One subtlety with the above step is that when we appeal to the IND-CPA security of the cryptosystem, clearly the simulator cannot use $dk$ to decrypt the plaintexts $X$ provided by the corrupted parties. This is handled as in [DN03]: Use the rewinding proofs of plaintext knowledge to extract $x$ instead of getting it by decryption. Since we are no longer describing a UC simulator, but simply analyzing it, we can use rewinding on the proofs of knowledge: We just rewind the entire UC execution including environment and adversary — see [DN03] for details. Problems are not over, however, as $dk$ is also used in Multiply when the parties decrypt $F$ and $G$. This is handled by picking uniformly random $f, g \in \mathbb{Z}_N$

[14] For convenience of language, let us just assume that the computation has one public output $y \in \mathbb{Z}_N$ and therefore just one output wire in the circuit and just one output ciphertext $Y$ in the protocol.

[15] those contained in the ideal functionality

and then doing a simulated decryption of $F$ to $f$ and a simulated decryption of $G$ to $g$. I.e., except if we are looking at the case where $(f, g) = (f', g') = (\delta_a, \delta_b)$ for a pair $(F', G')$ for which we already simulated a decryption to $(f', g')$ and for which the adversary knows $\delta_a$ and $\delta_b$. In this case we extract $\delta_a$ and $\delta_b$ from the adversary, set $f = f' + \delta_a$ and $g = g' + \delta_b$ and simulate decryption of $F$ and $G$ to $f$ respectively $g$.

We then make the change that we do simulated decryptions of $F$ and $G$ to $D_{dk}(F)$ respectively $D_{dk}(G)$, instead of doing simulated decryptions of $F$ and $G$ to the random elements $f$ and $g$ defined above. To distinguish the two distributions produced by this change, the adversary has to be able to distinguish $D_{dk}(F)$ and $D_{dk}(G)$ from the random elements $f$ and $g$ defined above. In Section 4.2 we argued that it can do this with at most negligible advantage.

We then make the change that we do real decryptions of $F$ and $G$ instead of simulated decryptions to $D_{dk}(F)$ respectively $D_{dk}(G)$. This defines indistinguishable distribution by the security of the decryption simulator.

We then make the change that we do a real decryption of $Y$ instead of a simulated decryption of $Y$ to the $y$ obtained from the ideal functionality. That this produces an indistinguishable distribution is seen as follows: We are by now using the correct inputs $x$ on behalf of the honest parties, and we are inputting the plaintext $x$ of the $X$ provided by corrupted parties to the ideal functionality. Therefore the ideal functionality and the protocol are computing on the same input values $x$. Therefore $Y$ contains the $y$ returned by the ideal functionality, except with negligible probability, and thus the simulated decryption to $y$ and the honest decryption of $Y$ are indistinguishable.

After this last change, we have actually arrived at the real-life protocol and thus at the end of the sketch of the UC simulator and its security.