Always Have a Backup Plan: Fully Secure Synchronous MPC with Asynchronous Fallback

Erica Blum[1], Chen-Da Liu-Zhang[2], and Julian Loss[1]

[1] {erblum,jloss}@cs.umd.edu, University of Maryland
[2] lichen@inf.ethz.ch, ETH Zurich

Abstract. Protocols for secure Multi-Party Computation (MPC) can be classified according to the underlying communication model. Two prominent communication models considered in the literature are the synchronous and asynchronous models, which considerably differ in terms of the achievable security guarantees. Synchronous MPC protocols can achieve the optimal corruption threshold $n/2$ and allow every party to give input, but become completely insecure when synchrony assumptions are violated. On the other hand, asynchronous MPC protocols remain secure under arbitrary network conditions, but can tolerate only $n/3$ corruptions and parties with slow connections unavoidably cannot give input.

A natural question is whether there exists a protocol for MPC that can tolerate up to $t_s < n/2$ corruptions under a synchronous network and $t_a < n/3$ corruptions even when the network is asynchronous. We answer this question by showing tight feasibility and impossibility results. More specifically, we show that such a protocol exists if and only if $t_a + 2t_s < n$ and the number of inputs taken into account under an asynchronous network is at most $n - t_s$.

1 Introduction

Secure multi-party computation (MPC) allows a set of parties $\mathcal{P} = \{P_1, \ldots, P_n\}$ to compute an arbitrary function of their private inputs, even if an adversary corrupts some of the parties. Intuitively, security in MPC means that the parties' inputs remain secret (apart from what is revealed by the computed output), and that the computed output is correct.

One can classify the results in MPC according to the underlying communication model. The *synchronous* model assumes that there is some parameter Δ known to all parties such that whenever a party sends a message, the recipient is guaranteed to receive it within time at most Δ. It is possible to achieve very strong security guarantees in this model; for example, prior work has shown how to achieve MPC with *full security*, where parties are guaranteed to obtain the correct output, for up to $t_s < \frac{n}{2}$ corruptions [31, 6, 15, 46, 2, 3, 30, 20, 21, 33, 22, 27, 24]. However, one can argue that the synchrony assumption is too strong: if an honest party P doesn't manage to send a message within Δ delay, it is considered dishonest in the synchronous model. As a consequence, synchronous protocols generally lose all security guarantees (e.g., parties can jointly reconstruct P's secret-shared input) if the network delays are greater than expected. This is of particular concern in real-world deployments, where it may not be possible to guarantee ideal network conditions at all times.

In the asynchronous model, the assumption of a known upper bound on network delay is dropped, so that the network delay can be arbitrarily large. The asynchronous model is therefore a safe choice for modeling even the most unpredictable real-world networks; however, prior work has shown that optimal security guarantees in this model are necessarily weaker than in the synchronous model: MPC can be achieved in the asynchronous model only for $t_a < \frac{n}{3}$ corruptions, and the output is not guaranteed to take into account all inputs into the computation [7, 34, 4, 19, 17].

In this paper, we investigate MPC protocols that keep strong security guarantees under both communication models. More specifically, let $t_a < \frac{n}{3}$ and $t_s < \frac{n}{2}$. We ask the following question:

Is there a protocol for MPC that is secure under t_s corruptions under a synchronous network, and t_a corruptions under an asynchronous network?

We completely answer this question by showing tight feasibility and impossibility results:

Feasibility result. We give an MPC protocol that is fully secure up to t_s corruptions under a synchronous network and up to t_a corruptions under an asynchronous network, as long as $t_a + 2t_s < n$. The number of inputs taken into account in the latter case is $n - t_s$.

Optimality of our protocol. We show that our protocol is tight with respect to both the threshold tradeoffs t_a and t_s, and also the number of inputs taken into account. More concretely, we show:

- For any t_s, any MPC protocol which achieves full security up to t_s corruptions under a synchronous network cannot take into account more than $n - t_s$ inputs when run over an asynchronous network, even if all parties are guaranteed to be honest in this case.
- For any $t_a + 2t_s \geq n$, there is no MPC protocol which gives full security up to t_s corruptions under a synchronous network, and where all parties output the same value up to t_a corruptions under an asynchronous network.

1.1 Technical Overview

In this section, we briefly sketch our protocol for MPC that achieves full security up to t_s corruptions under a synchronous network and up to t_a corruptions under an asynchronous network, for any $0 \leq t_a < \frac{n}{3} \leq t_s < \frac{n}{2}$ satisfying $t_a + 2t_s < n$. Note that we impose $t_s \geq \frac{n}{3}$, because otherwise one can use existing asynchronous MPC protocols (e.g. [34]), which already achieve such security guarantees, i.e., are fully secure under an asynchronous network (and hence also a synchronous network), and moreover take into account all inputs when given some initial synchronous rounds.

At a very high level, we run two sub-protocols $\Pi_{\texttt{smpc}}$ and $\Pi_{\texttt{ampc}}$ one after the other, where $\Pi_{\texttt{smpc}}$ is a t_s-secure synchronous protocol and $\Pi_{\texttt{ampc}}$ is a t_a-secure asynchronous protocol (e.g. [7, 34, 17]). Conceptually, a key challenge is that parties are not able to obtain output in both protocols, as this would violate privacy. Thus, parties need to agree on whether to run the second sub-protocol. For that, the key is that the protocol $\Pi_{\texttt{smpc}}$ gives guarantees even when the network is asynchronous. More concretely, $\Pi_{\texttt{smpc}}$ achieves unanimous output up to t_a corruptions under an asynchronous network. Intuitively, this means that the protocol is secure, except the fact that either all parties learn the correct output, or all parties obtain $\perp$ as the output.

When the network is synchronous, security of the overall protocol is inherited from the first sub-protocol. In the case where the network is asynchronous, parties either learn the correct output from the first sub-protocol or all parties obtain $\perp$ and can safely execute the second sub-protocol.

Synchronous MPC with Asynchronous Unanimous Output. In order to construct the first sub-protocol, we modify a synchronous MPC protocol that uses threshold homomorphic encryption [22, 27]. The original protocol provides full security up to $t_s < \frac{n}{2}$ corruptions in a synchronous network.

Let us briefly recall the high-level structure of the original protocol [22, 27]. The protocol is based on a threshold version of the Paillier cryptosystem [43]. For a plaintext a, let us denote $\overline{a}$ an encryption of a. The cryptosystem is homomorphic: given encryptions $\overline{a}$, $\overline{b}$, one can compute an encryption of $a + b$, which we denote $\overline{a} \boxplus \overline{b}$. Similarly, from a constant plaintext α and an encryption $\overline{a}$ one can compute an encryption of αa, which we denote $\alpha \boxdot \overline{a}$.

The protocol starts by having each party publish encryptions of its input values, as well as zero-knowledge proofs that it knows these values. Then, parties compute addition and multiplication gates to obtain a common ciphertext, which they jointly decrypt using threshold decryption. Any linear operation (addition or multiplication by a constant) can be performed non-interactively, due to the homomorphism property of the threshold encryption scheme. Given encryptions $\overline{a}$, $\overline{b}$ of input values to a multiplication gate, parties can compute an encryption of $c = ab$ as follows:

1. Each P_i chooses a random $d_i \in \mathbf{Z}_n$ and uses a byzantine broadcast protocol to distribute encryptions $\overline{d_i}$ and $\overline{d_i b}$.
2. Parties prove (in zero-knowledge) knowledge of the plaintext d_i and that $\overline{d_i b}$ encrypts the correct value. Let S be the subset of parties succeeding in both proofs.
3. Parties compute $\overline{a} \boxplus (\boxplus_{i \in S} \overline{d_i})$ and decrypt it using a threshold decryption.
4. Parties set $\overline{c} = (a + \sum_{i \in S} d_i) \boxdot \overline{b} \boxminus ((\boxplus_{i \in S} \overline{d_i b}))$.

Intuitively, the protocol works because 1) honest parties have agreement on the ciphertext to decrypt after evaluating the circuit, and 2) only ciphertexts or random values are revealed.

When the above protocol is executed over an asynchronous network, all security guarantees are lost. This is because synchronous broadcast protocols do not necessarily give any guarantees when run over an asynchronous network. As a result, parties lose agreement in critical points in the protocol. For example, parties can receive different sets of encrypted inputs during input distribution, which can lead to privacy violations if the mismatching inputs are decrypted. Moreover, parties must reach agreement on S, and S must contain at least one honest party contributing to the reconstructed random value to ensure that the value is random and unknown to the adversary. For this, it is essential that parties have agreement on whether a zero-knowledge proof was successful or not. Finally, parties need to reach agreement on which ciphertext to decrypt, or whether to decrypt at all.

To solve the problems above, we replace the problematic sub-protocols with versions that achieve certain guarantees even when the network is asynchronous. More concretely, we will make use of broadcast, byzantine agreement and asynchronous common subset sub-protocols. The broadcast protocol will ensure that encrypted inputs from honest parties can only lead to correct ciphertexts. When used with the byzantine agreement protocol proposed in [8], it will allow parties to reach agreement on the set S for the multiplication gates. Finally, we make use of the enhanced asynchronous common subset sub-protocol in [9] at the end of the circuit computation to decide whether or not parties should proceed to decrypt a ciphertext, or output $\perp$.

1.2 Related Work

Despite being a very natural direction of research, protocols resilient to both synchronous and asynchronous networks have only begun to be studied in relatively recent works. The closest related work is the recent work by Blum et al. [8] which considers the problem of byzantine agreement in a 'hybrid' network model. The authors prove that byzantine agreement t_s-secure under a synchronous network and t_a-secure under an asynchronous network is possible if and only if $t_a + 2t_s < n$. The work was recently further extended to the problem of state-machine replication [9]. Our work extends both above works to the problem of secure multi-party computation, and in particular, introduces techniques to protect privacy of inputs in the hybrid network setting.

Another close related work is the work by Guo et al. [32], which considers a weakened variant of the classical synchronous model. Here, an attacker can temporarily disconnect a subset of parties from the rest of the network. Guo et al. gave byzantine agreement and multi-party computation protocols tolerating the optimal corruption threshold in this model, and Abraham et al. [1] achieve similar guarantees for state-machine replication. The main difference between these works and ours is that their protocols need to assume synchrony in part of the network. In contrast, our protocols give guarantees even if the network is fully asynchronous.

Further related work for the problem of byzantine agreement protocols include the work by Malkhi et al. [41] which considers protocols that provide guarantees when run in synchronous or partially synchronous networks, and the work by Liu et al. [38] which designs protocols resilient to malicious corruptions in a synchronous network, and fail-stop corruptions in an asynchronous network. Kursawe [37] shows a protocol for asynchronous byzantine agreement that reaches agreement more quickly in case the network is synchronous.

A line of works [44, 45, 40, 39] has recently investigated protocols that achieve *responsiveness*. These protocols operate under a synchronous network, but in addition give the guarantee that parties obtain output as fast as the actual network delay allows. None of these works provide security guarantees when the network is not synchronous.

2 Model

Our protocols are proven secure in the universally composable (UC) framework [13] (see Section A for a summary).

2.1 Setup

We consider a setting with n parties $\mathcal{P} = \{P_1, \dots, P_n\}$. We denote κ the security parameter.

Common reference string. We assume that the parties have a common reference string (CRS). The CRS is used to realize the bilateral zero-knowledge UC functionalities.

Digital signatures. We assume that parties have a public-key infrastructure available, i.e., all parties hold the same vector of public keys $(\mathtt{pk}_1, \ldots, \mathtt{pk}_n)$, and each party P_i holds the secret key $\mathtt{sk}_i$ associated with $\mathtt{pk}_i$. This allows parties to sign values.

Definition 1. *A digital signature scheme is a tuple of algorithms* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ *such that:*

- *Key generation: On input* 1^κ*, the key generation algorithm outputs* $(\mathtt{pk}, \mathtt{sk}) = \mathsf{Gen}(1^\kappa)$ *a pair of public and secret key.*
- *Signature: Given a secret key* $\mathtt{sk}$ *and a message* x*, the signing algorithm outputs* $\sigma = \mathsf{Sign}_{\mathtt{sk}}(x)$ *a signature of message* x*.*
- *Verification: Given a public key* $\mathtt{pk}$*, a message* x *and a signature* σ*, the verification algorithm outputs* $\mathsf{Ver}_{\mathtt{pk}}(x, \sigma) = 1$ *if and only if* σ *is a correct signature of* x*.*

We require that the signature scheme is correct and unforgeable against chosen message attacks.

Threshold encryption. We assume that parties have a threshold additively homomorphic encryption setup available. That is, it provides to each party P_i a global public key $\mathtt{ek}$ and a private key share $\mathtt{dk}_i$.

Definition 2. *A threshold homomorphic encryption scheme is a public-key encryption scheme which has the following properties:*

- *Key generation: The key generation algorithm is parameterized by* (t, n) *and outputs* $(\mathtt{ek}, \mathtt{dk}) = \mathsf{Gen}_{(t,n)}(1^\kappa)$*, where* $\mathtt{ek}$ *is the public key, and* $\mathtt{dk} = (\mathtt{dk}_1, \ldots, \mathtt{dk}_n)$ *is the list of private keys.*
- *Encryption: Given* $\mathtt{ek}$ *and a plaintext* a *one can compute an encryption* $\overline{a} = \mathtt{Enc}_{\mathtt{ek}}(a)$ *of* a*.*
- *Decryption: Given a ciphertext* c *and a secret key share* $\mathtt{dk}_i$*, there is an algorithm that outputs* $d_i = \mathtt{DecShare}_{\mathtt{dk}_i}(c)$*, such that* $(d_1, \ldots, d_n)$ *forms a* t*-out-of-*n *sharing of the plaintext* $m = \mathtt{Dec}_{\mathtt{dk}}(c)$*. Moreover, with* t *decryption shares* $\{d_i\}$*, one can reconstruct the plaintext* $m = \mathtt{Rec}(\{d_i\})$*.*
- *Additively homomorphic: Given* $\mathtt{ek}$ *and two encryptions* $\overline{a} \in \mathtt{Enc}_{\mathtt{ek}}(a)$ *and* $\overline{b} \in \mathtt{Enc}_{\mathtt{ek}}(b)$*, one can efficiently compute an encryption* $\overline{a+b} \in \mathtt{Enc}_{\mathtt{ek}}(a+b)$*. We write* $\overline{a+b} = \overline{a} + \overline{b}$*.*
- *Multiplication by constant: Given* $\mathtt{ek}$*, a plaintext* α *and an encryption* $\overline{a} \in \mathtt{Enc}_{\mathtt{ek}}(a)$*, one can efficiently compute a random encryption* $\overline{\alpha a} \in \mathtt{Enc}_{\mathtt{ek}}(\alpha a)$*. We write* $\overline{\alpha a} = \alpha \boxdot \overline{a}$*.*

Such a threshold encryption scheme can be based on, for example, the Paillier cryptosystem [43] (see Section B). We use the threshold encryption scheme as a basic tool in the MPC protocol, following the approach in [22, 27].

2.2 Communication Network and Adversary

We consider a complete network of authenticated channels. Our protocols operate in two possible settings: synchronous or asynchronous.

In the synchronous setting, all parties have access to synchronized clocks and all messages are guaranteed to be delivered within some known upper bound delay Δ. Within Δ, the adversary can schedule the messages arbitrarily. In particular, the adversary is *rushing*, i.e., within the same round, the adversary is allowed to send its messages after seeing the honest parties' messages. Sometimes it is convenient to describe a protocol in rounds, where each round r refers to the interval of time $(r-1)\Delta$ to $r\Delta$. In such case, we say that a party receives a message in round r if it receives the message within that time interval. Moreover, we say a party sends a message in round r when it sends the message at the beginning of the round, i.e., at time $(r-1)\Delta$.

In the asynchronous setting, both assumptions above are removed. That is, parties do not have access to synchronized clocks, and the adversary is allowed to arbitrarily schedule the delivery of the messages. However, we assume that all messages are eventually delivered (i.e., the adversary cannot drop messages).

We consider a static adversary who corrupts parties in an arbitrary manner at the beginning of the protocol.

3 Definitions

3.1 Broadcast

Broadcast allows a designated party called the *sender* to consistently distribute a message among a set of parties.

Definition 3. *(Broadcast) Let Π be a protocol executed by parties $P_1, \ldots, P_n$, where a designated sender P_s initially holds an input v, and parties terminate upon generating output.*

- *Validity: Π is t-valid if the following holds whenever up to t parties are corrupted: if P_s is honest, then every honest party which outputs, outputs v.*
- *Weak-validity: Π is t-weakly valid if the following holds whenever up to t parties are corrupted: if P_s is honest, then every honest party which outputs, outputs v or $\perp$.*
- *Consistency: Π is t-consistent if the following holds whenever up to t parties are corrupted: every honest party which outputs, outputs the same value.*
- *Liveness: Π is t-live if the following holds whenever up to t parties are corrupted: every honest party outputs a value.*

If Π is t-valid, t-consistent and t-live, we say that it is t-secure.

In the asynchronous setting, one can formally prove that the strong broadcast guarantees as in Definition 3 cannot be achieved [10, 11]. Intuitively, the reason is that one cannot distinguish between a dishonest sender not sending messages, or an honest sender's messages being delayed. Hence, a useful primitive is a *reliable broadcast* protocol, which achieves the same guarantees as a broadcast protocol, except that the liveness property is relaxed and divided into two properties.

Definition 4. *(Reliable Broadcast) Let Π be a protocol executed by parties $P_1, \ldots, P_n$, where a designated sender P_s initially holds an input v, and parties terminate upon generating output.*

- *Validity: Π is t-valid if the following holds whenever up to t parties are corrupted: if P_s is honest, then every honest party outputs v.*
- *Consistency: Π is t-consistent if the following holds whenever up to t parties are corrupted: either no honest party terminates, or else all honest parties output the same value.*

Observe that, in contrast to Definition 3, when the sender is dishonest, it is allowed that no honest party terminates.

3.2 Byzantine Agreement

In a byzantine agreement protocol, each party P_i starts with a value v_i. The protocol allows the set of parties to agree on a common value. The achieved guarantees are the same as in broadcast (see Definition 3), except that validity is adapted accordingly.

Definition 5. *(Byzantine Agreement) Let Π be a protocol executed by parties $P_1, \ldots, P_n$, where each party P_i initially holds an input v_i, and parties terminate upon generating output.*

- *Validity: Π is t-valid if the following holds whenever up to t parties are corrupted: if every honest party has the same input value v, then every honest party that outputs, outputs v.*
- *Consistency: Π is t-consistent if the following holds whenever up to t parties are corrupted: every honest party which outputs, outputs the same value.*
- *Liveness: Π is t-live if the following holds whenever up to t parties are corrupted: every honest party outputs a value.*

If Π is t-valid, t-consistent and t-live, we say that it is t-secure.

3.3 Asynchronous Common Subset

A protocol for the asynchronous common subset (ACS) problem [7, 12, 42, 9] allows n parties, each with an initial input, to agree on a subset of the inputs. For this primitive, we do not assume that parties terminate upon generating output, that is, even after generating output parties are allowed to keep participating in the protocol indefinitely.

Definition 6. *(ACS) Let Π be a protocol executed by parties $P_1, \ldots, P_n$, where each party initially holds an input v, and parties output sets of size at most n.*

- *Validity: Π is t-valid if the following holds whenever up to t parties are corrupted: if all honest parties start with the same input v, then every honest party which outputs, outputs $\{v\}$.*
- *Consistency: Π is t-consistent if the following holds whenever up to t parties are corrupted: every honest party which outputs, outputs the same set.*
- *Liveness: Π is t-live if the following holds whenever up to t parties are corrupted: every honest party outputs.*
- *Validity liveness: Π is t-live valid if the following holds whenever up to t parties are corrupted: If all honest parties start with the same input, then every honest party outputs.*
- *Set quality: Π has (t, h)-set quality if the following holds whenever up to t parties are corrupted: if an honest party outputs a set, it contains the inputs of at least h honest parties.*

3.4 Multi-Party Computation

At a high level, a protocol for multi-party computation (MPC) allows n parties $P_1, \ldots, P_n$, where each party P_i has an initial input x_i, to jointly compute a function over the inputs $f(x_1, \ldots, x_n)$ in such a way that nothing beyond the output is revealed.

We consider different types of security guarantees for our MPC protocols. The first one is the strongest guarantee that an MPC protocol can offer: MPC with guaranteed output delivery, or full security (cf. [31, 6, 15, 46, 3, 22]). Here, honest parties are guaranteed to obtain the correct output. Formally, in UC this is modeled as the protocol realizing the ideal functionality where each party P_i inputs x_i to the functionality, and it then outputs $f(x_1, \ldots, x_n)$ to the parties.

When the network is asynchronous, it is provably impossible that the computed function takes into account all inputs from honest parties [7, 34, 4, 19, 17]. The reason is that one cannot distinguish between a dishonest party not sending its input, or an honest party's input being delayed. Hence, we say that a protocol achieves L-output quality, if the output to be computed contains the inputs from at least L parties. Traditional asynchronous protocols in the literature (e.g. [5, 7, 34]) achieve $(n-t)$-output quality under t corruptions, since the computed output ignores up to t inputs. Formally this is modelled in the ideal functionality as allowing the ideal adversary to choose a subset S of L parties. The functionality then computes $f(x_1, \ldots, x_n)$, where $x_i = v_i$ is the input of P_i in the case that $P_i \in S$, and otherwise $x_i = \bot$.

Functionality $\mathcal{F}_{\text{SFE}}^{\mathsf{sec},L}$

$\mathcal{F}_{\text{SFE}}$ is parameterized by a set $\mathcal{P}$ of n parties and a function $f : (\{0,1\}^* \cup \{\bot\})^n \to (\{0,1\}^*)^n$. For each $P_i \in \mathcal{P}$, initialize the variables $x_i = y_i = \bot$. Set $S = \mathcal{P}$.

1: On input $(\textsc{Input}, v)$ from $P_i \in \mathcal{P}$, if $P_i \in S$, set $x_i = v$ and send a message $(\textsc{Input}, P_i)$ to the adversary.
2: On input $(\textsc{OutputSet}, S')$ from the ideal adversary, where $S' \subseteq \mathcal{P}$ and $|S'| = L$, set $S = S'$ and $x_i = \bot$ for each $P_i \notin S$.
3: Once all inputs from honest parties in S have been input, set each $y_i = f(x_1, \ldots, x_n)$.
4: On input $(\textsc{GetOutput})$ from P_i, output $(\textsc{Output}, y_i, \text{sid})$ to P_i.

In addition to MPC with full security, we also consider weaker notions of security. In MPC with selective output [35, 18], the ideal world adversary can choose any subset of parties to receive $\bot$, instead of the correct output. The last type of security we consider is called MPC with unanimous output [31, 29]. Under this definition, the adversary is permitted to choose whether all honest parties receive the correct output or all honest parties receive $\bot$ as output; as such it is is slightly stronger than MPC with selective output, but weaker than full security.

Let us denote the functionality $\mathcal{F}_{\text{SFE}}^{\texttt{sout},L}$ (resp. $\mathcal{F}_{\text{SFE}}^{\texttt{uout},L}$), the above functionality, where the adversary can selectively choose any subset of parties to obtain $\perp$ as the output (resp. choose that either all honest parties receive $f(x_1,\ldots,x_n)$ or $\perp$).

Definition 7. *A protocol π achieves full security (resp. selective output; unanimous output) with L output-quality if it UC-realizes functionality $\mathcal{F}_{\text{SFE}}^{\texttt{sec},L}$ ($\mathcal{F}_{\text{SFE}}^{\texttt{sout},L}$; $\mathcal{F}_{\text{SFE}}^{\texttt{uout},L}$).*

Since protocols run in a synchronous network typically achieve n-output quality, we implicitly assume that all synchronous protocols discussed achieve n-output quality (unless otherwise specified).

Weak termination. In general, traditional protocols for MPC require that the protocol terminates (halts). In this paper, we capture a slightly weaker version as a property of a protocol: we say that a protocol has weak termination, if parties are guaranteed to terminate upon receiving an output different than $\perp$, but do not necessarily terminate if the output is $\perp$.

4 Synchronous MPC with Asynchronous Unanimous Output and Weak Termination

In this section, we show a protocol $\Pi_{\texttt{smpc}}^{t_s,t_a}$ that achieves full security up to t_s corruptions when the network is synchronous, and achieves unanimous output with weak termination up to t_a corruptions when the network is asynchronous, for any $0 \le t_a < \frac{n}{3} \le t_s < \frac{n}{2}$ satisfying $t_a + 2t_s < n$. The protocol relies on a number of primitives:

- $\Pi_{\texttt{bc}}^{t_s,t_a}$ is a broadcast protocol that is t_s-secure when run in a synchronous network, and is t_a-weakly valid and t_a-live when run in an asynchronous network.
- $\Pi_{\texttt{ba}}^{t_s,t_a}$ is a byzantine agreement protocol that is t_s-secure when run in a synchronous network, and is t_a-secure when run in an asynchronous network.
- $\Pi_{\texttt{acs}}^{t_s,t_a}$ is an asynchronous common subset protocol that is t_s-valid and t_s-live valid when run in a synchronous network, and is t_a-consistent, t_a-live and has $(t_a,1)$-set quality when run in an asynchronous network.
- $\Pi_{\texttt{zk}}^{t_s,t_a}$ is a multi-party zero-knowledge protocol that allows a party P_i to prove knowledge of a witness w for a statement x satisfying a certain relation R towards all parties. The protocol achieves full security up to t_s corruptions when the network is synchronous, and achieves security with selective abort up to t_a corruptions when the network is asynchronous.

In the following, we show instantiations for each of the sub-protocols.

4.1 Broadcast

We use the Dolev-Strong protocol [28, 8] to achieve a broadcast protocol that is t_s-secure when run in a synchronous network, and is t_a-weakly valid and t_a-live when run in an asynchronous network. The idea is quite simple: we run the Dolev-Strong protocol for t_s+1 rounds and output v if v is the only value accepted, and otherwise $\perp$. In the protocol, we say that a message (v,Σ) at round r is valid if Σ contains r signatures, where one of them is from the sender and the other $r-1$ from distinct additional parties.

Protocol $\Pi_{\texttt{bc}}^{t_s,t_a}$

Sender P_s has input v. Each party P_i keeps local variables $\Sigma_i, \Omega_i := \varnothing$.

Round 1. P_s signs its input v to obtain a signature σ_s, and sends $(v,\{\sigma_s\})$ to all parties.

Round $1 \le r \le t_s$. Each P_i does: Upon receiving a valid message (v,Σ), add v to Ω_i. Compute a signature σ_i on v and let $\Sigma_i := \Sigma_i \cup \{\sigma_i\}$. Send (v,Σ_i) to all parties in the next round.

Output determination

Round t_s+1. Each P_i does: Upon receiving a valid message (v,Σ), add v to Ω_i. Then, if Ω_i contains exactly one value v', output v' and terminate. Otherwise, output $\perp$ and terminate.

Lemma 1. *Let n, t_s, t_a be such that $t_a, t_s < n$. $\Pi_{\mathtt{bc}}^{t_s,t_a}$ is a broadcast protocol that is t_s-secure when run in a synchronous network, and is t_a-weakly valid and t_a-live when run in an asynchronous network.*

Proof. Security under a synchronous network is achieved via the standard analysis of the Dolev-Strong protocol: If the sender is honest, each honest party P_i adds the sender's input v to Ω_i, and no honest party adds any other value. Moreover, if an honest P_i adds v to Ω at round $r \leq t_s$, every honest P_j adds v at round $r + 1$. And if P_i adds v at round $t_s + 1$, then there are $t_s + 1$ signatures on v and hence an honest P_k added v at some round $r' \leq t_s$ and every honest party added v at round $r' + 1$. If the network is asynchronous, t_a-liveness is trivial, since every honest party outputs at (local) time $(t_s + 1)\Delta$. The protocol is also t_a-weakly valid because the adversary cannot forge signatures from the sender P_s.

□

4.2 Byzantine Agreement

In [8], the authors show a byzantine agreement protocol that is t_s-secure when run in a synchronous network, and is t_a-secure when run in an asynchronous network. We briefly sketch the construction here.

At a high level, their protocol consists of two phases: a round-based BA followed by an event-based BA. An honest party P_i with input v_i uses v_i as their input for the round-based phase. If the round-based phase terminates with output $v' \in \{0, 1\}$ within some (local) time limit, P_i uses v' as input for the event-based phase. (The timeout is chosen such that the honest parties are guaranteed to receive output from the round-based BA before the timeout when the network is synchronous and at most t_s parties are corrupted.) Otherwise, if the round-based phase times out without producing boolean output, P_i proceeds directly to the event-based phase, using their original input v_i as their input. P_i then outputs the output they receive from the event-based phase.

Intuitively, when the network is synchronous and there are t_s corruptions, the security guarantees for the full protocol are primarily inherited from the round-based BA sub-protocol (with the caveat that the event-based BA sub-protocol guarantees t_s-validity and therefore preserves the results of the first phase). When the network is asynchronous and there are t_a corruptions, the round-based BA protocol need only be t_a-weakly valid, after which the desired security guarantees follow from the security properties of the event-based BA sub-protocol. We state the following lemma. The proof can be found in [8].

Lemma 2. *Let n, t_s, t_a be such that $0 \leq t_a < \frac{n}{3} \leq t_s < \frac{n}{2}$ and $t_a + 2t_s < n$. There is a protocol $\Pi_{\mathtt{ba}}^{t_s,t_a}$ satisfying the following properties:*

1. *When run in a synchronous network, it is t_s-secure.*
2. *When run in an asynchronous network, it is t_a-secure.*

4.3 Asynchronous Common Subset

We describe the protocol $\Pi_{\mathtt{acs}}^{t_s,t_a}$ presented in [9], which is an asynchronous common subset protocol that is t_s-valid and t_s-live valid when run in a synchronous network, and is t_a-consistent, t_a-live and has $(t_a, 1)$-set quality when run in an asynchronous network.

The protocol is based on previous asynchronous common subset protocols [7, 12, 42], but the output decision differs. The general idea is that parties run n executions of Bracha's reliable broadcast protocol [10], where each party P_i acts as the sender in each execution, followed by n executions of byzantine agreement to agree on a subset of parties that finished the reliable broadcast protocol. If a party sees $n - t_s$ broadcasts terminate on the same value, it outputs this value. Otherwise, it waits until all byzantine agreement protocols have terminated and then outputs based on the set C of senders for whom the corresponding BA output 1: If there is a majority v of broadcasted values from parties in C, output v, and otherwise output the union of all broadcasted values from parties in C.

In order to achieve the guarantees described above, the protocol needs a reliable broadcast protocol which, under an asynchronous network, achieves validity up to t_s corruptions, and consistency up to t_a corruptions. Let us denote $\mathtt{RBC}_i$ the reliable broadcast protocol where P_i acts as the sender, and $\mathtt{BA}_i$ the byzantine agreement protocol which outputs whether $\mathtt{RBC}_i$ has terminated or not.

Protocol $\Pi_{\mathtt{acs}}^{t_s,t_a}(P_i)$

Participate in each protocol $\mathtt{RBC}_j$, $j \neq i$, as the receiver, and participate in $\mathtt{RBC}_i$ as the sender.
On output from $\mathtt{RBC}_j$, if an input has not yet been provided to $\mathtt{BA}_j$, then input 1 to $\mathtt{BA}_j$.
When $n - t_a$ of the protocols $\mathtt{BA}_j$ have output 1, provide input 0 to each instance $\mathtt{BA}_j$ that has not yet been provided input.

Output determination

```
if at least n − t_s executions of RBC_j output a value v then
    Output {v}.
else
    let C := {j | BA_j output 1}. Once all instances BA_j have been completed and |C| ≥ n − t_a, wait for
    the output v_j of each RBC_j, j ∈ C.
    if A majority of the executions {RBC_j}_{j∈C} output a value v then
        Output {v}.
    else
        Output ⋃_{j∈C} {v_j}.
    end if
end if
```

We state the following lemma. The proof can be found in [9].

Lemma 3. *Let n, t_s, t_a be such that $0 \leq t_a < \frac{n}{3} \leq t_s < \frac{n}{2}$ and $t_a + 2t_s < n$. Protocol $\Pi_{\mathtt{acs}}^{t_s,t_a}$ satisfies the following properties:*

1. *When run in a synchronous network, it is t_s-valid and t_s-live valid.*
2. *When run in an asynchronous network, it is t_a-consistent, t_a-live and has $(t_a, 1)$-set quality.*

4.4 Zero-Knowledge

Let us assume a binary relation R, consisting of pairs (x, w), where x is the statement, and w is a witness to the statement. A zero-knowledge proof allows a prover P to prove to a verifier V knowledge of w such that $R(x, w) = 1$. We are interested in zero-knowledge proofs for three types of relations, parameterized by a threshold encryption scheme with public encryption key $\mathtt{ek}$:

1. *Proof of Plaintext Knowledge:* The statement consists of $\mathtt{ek}$, and a ciphertext c. The witness consists of a plaintext m and randomness r such that $c = \mathtt{Enc}_{\mathtt{ek}}(m, r)$.
2. *Proof of Correct Multiplication:* The statement consists of $\mathtt{ek}$, and ciphertexts c_1, c_2 and c_3. The witness consists of a plaintext m_1 and randomness r_1, r_3 such that $c_1 = \mathtt{Enc}_{\mathtt{ek}}(m_1, r_1)$ and $c_3 = m_1 \cdot c_2 + \mathtt{Enc}_{\mathtt{ek}}(0; r_3)$.
3. *Proof of Correct Decryption:* The statement consists of $\mathtt{ek}$, a ciphertext c, and a decryption share d. The witness consists of a decryption key share $\mathtt{dk}_i$, such that $d = \mathtt{Dec}_{\mathtt{dk}_i}(c)$.

Examples of bilateral zero-knowledge proofs of knowledge can be found for example in [23, 22]. The bilateral UC zero-knowledge functionality $\mathcal{F}_{\text{ZK}}$ for a relation R and a pair prover P and a verifier V is defined as follows: P inputs a pair (x, w) instance-witness, and the functionality outputs (x, b) to the verifier, where $b = 1$ if and only if $R(x, w) = 1$. It is known that assuming a CRS, one can realize a bilateral UC zero-knowledge functionality $\mathcal{F}_{\text{ZK}}$ [14, 26, 17].

Multi-party zero-knowledge protocols. A multi-party zero-knowledge protocol allows a prover P to prove towards all parties knowledge of a witness w for a statement x such that $R(x, w) = 1$. The ideal functionality can be seen as a special case of secure function evaluation, where the prover inputs (x, w), and the parties obtain the statement x and 1 if and only if $R(x, w) = 1$.

Assuming a bilateral UC zero-knowledge functionality $\mathcal{F}_{\text{ZK}}$, one can construct a UC multi-party zero-knowledge functionality $\mathcal{F}_{\text{MZK}}$ using so-called *certificates* [34] as follows: The prover bilaterally performs the zero-knowledge proofs towards each of the recipients, who upon a successful proof, send a signature that the proof was correct. Once the prover collects a list L of $t_s + 1$ signatures, the list works as a certificate that proves non-interactively that at least one honest party accepted the proof. The prover can hence broadcast the list L to let all honest parties know that the proof is correct. If the last broadcast is executed with the protocol $\Pi_{\mathsf{bc}}^{t_s,t_a}$, it is easy to see that under t_s corruptions and a synchronous network the multi-party zero-knowledge functionality achieves full security. Moreover, if there are up to t_a corruptions and an asynchronous network, broadcast guarantees weak validity, so the protocol achieves security with selective abort (in the last step, if the prover has a certificate, it is guaranteed that parties receive the certificate or $\perp$, and a dishonest party who did not collect such certificate cannot make the parties accept the proof).

Protocol $\Pi_{\mathsf{zk}}^{t_s,t_a}$

Prover P proves knowledge of a witness w for a statement x satisfying a certain relation R towards all parties.

1: P inputs (x, w) to each bilateral $\mathcal{F}_{\text{ZK}}$.
2: Each P_i does: Upon a successful proof, compute $\sigma_i = \mathsf{Sign}_{sk_i}(x)$ and send σ_i to P.
3: P collects a list L of $t_s + 1$ signatures and broadcasts using protocol $\Pi_{\mathsf{bc}}^{t_s,t_a}$ the list L.
4: Each P_i does: Upon receiving a list L as output of the broadcast protocol, if L contains t_s+1 signatures on the same instance x, output $(x, 1)$. In any other case, output $\perp$.

Lemma 4. *Let R be a relation. Let n, t_s, t_a be such that $t_a, t_s < n$. $\Pi_{\mathsf{zk}}^{t_s,t_a}$ realizes the multi-party zero-knowledge functionality for P as prover with the following guarantees:*

1. *When run in a synchronous network, it achieves full security up to t_s corruptions.*
2. *When run in an asynchronous network, it achieves security with selective abort up to t_a corruptions.*

Proof. We prove each of the cases separately. We simulate in the hybrid where there is a trusted setup generating the keys in the real world. In the ideal world, the simulator $\mathcal{S}$ generates the PKI keys, and outputs the public keys to the adversary along with its secret keys.

Synchronous network and up to t_s corruptions. We describe the simulator $\mathcal{S}$ for the case where the network is synchronous and there are up to t_s corruptions. Let us first consider the case where the prover P is honest.

– $\mathcal{S}$ forwards the result from $\mathcal{F}_{\text{MZK}}$ to the adversary. If the result is positive, generate a signature σ_i on behalf of each honest party. Let L be list of signatures.
– On input correct signatures from the dishonest parties, it adds it to L.
– $\mathcal{S}$ emulates the messages of the broadcast protocol.

Now assume that P is dishonest.

– $\mathcal{S}$ gets the instance-witness pairs that P inputs to prove to each party. To the dishonest parties, output the instance and the bit 1 if and only if the witness is correct.
– For each of the pairs, forward a signature on behalf of the honest party if the witness is a correct witness to the corresponding instance.
– $\mathcal{S}$ receives a list L of $t_s + 1$ signatures on the same instance: input the instance and the witness to $\mathcal{F}_{\text{MZK}}$.

Asynchronous network and up to t_a corruptions. The only difference with respect to the case where the network is synchronous, is that the protocol $\Pi_{\mathsf{bc}}^{t_s,t_a}$ only provides weak-validity. In the simulation, it implies that the simulator will also need to simulate the $\perp$ messages from the broadcast protocols.

It is easy to see that the simulation goes through. In the case of a synchronous network and t_s corruptions, an honest prover collects at least $t_s + 1$ signatures and every honest receiver outputs 1. In the case the prover is dishonest, it cannot collect $t_s + 1$ signatures for an instance without having succeeded in one of the proofs, and hence each honest party outputs $\perp$. If the network is asynchronous, when the prover is honest, every honest party outputs 1 or $\perp$, where the set of parties that output $\perp$ is chosen by the adversary. In the case the prover is dishonest, the case is analogous as the synchronous case and every honest party outputs $\perp$.

□

4.5 Description of the Synchronous MPC Protocol

We start from the MPC protocol that uses homomorphic encryption presented in [22, 27]. The protocol was originally designed for the synchronous setting and guarantees full security up to $t_s < \frac{n}{2}$ corruptions. We modify the protocol to also achieve unanimous output up to t_a corruptions even when the network is asynchronous, as long as $0 \le t_a < \frac{n}{3} \le t_s < \frac{n}{2}$ satisfies $t_a + 2t_s < n$.

We assume that the computation is specified as a circuit with addition and multiplication gates. We assume that the plaintext space does not contain a special symbol $\perp$. For example, we can assume that the plaintext space is $\mathbf{Z}_N$ for some RSA modulus N and that we use a threshold version of the Paillier cryptosystem (see Section B).

When the network is synchronous, we need to ensure that parties start simultaneously in each of the sub-protocols in order to ensure that the security guarantees are preserved. For example, in $\Pi_{\mathtt{ba}}^{t_s,t_a}$ there is a timeout chosen such that honest parties are guaranteed to receive output when the network is synchronous. As a consequence, if parties start at different times, we lose the security guarantees in the synchronous case. In order to solve this, we wait at least for an upper bound on the running time of each sub-protocol. This allows parties to simultaneously start at each sub-protocol when the network is synchronous. Let us denote T_{bc}, T_{zk}, T_{ba}, T_{dec} upper bounds on the running time of $\Pi_{\mathtt{bc}}^{t_s,t_a}$, $\Pi_{\mathtt{zk}}^{t_s,t_a}$, n parallel executions of $\Pi_{\mathtt{ba}}^{t_s,t_a}$, and the Threshold Decryption sub-protocols respectively, in the case the network is synchronous.

Protocol $\Pi_{\mathtt{smpc}}^{t_s,t_a}(P_i)$

Let x_i denote the input value of party P_i. Let `abort` $= 0$.

Input Distribution

1: P_i computes $\overline{x_i}$ and broadcasts using $\Pi_{\mathtt{bc}}^{t_s,t_a}$ the ciphertext $\overline{x_i}$ and uses the multi-party zero-knowledge functionality $\mathcal{F}_{\text{MZK}}$ to prove knowledge of the plaintext of $\overline{x_i}$ towards all parties. Wait until $\max\{T_{bc}, T_{zk}\}$ clock ticks passed.
2: If there is a broadcast or zero-knowledge proof that has not terminated, or the number of correct encryptions received is less than $n - t_s$ inputs, set `abort` $= 1$. Continue participating in the sub-protocols, but do not compute any ciphertext.

Addition Gates Input: $\overline{a}, \overline{b}$. Output: $\overline{c}$.

1: P_i locally computes $\overline{c} = \overline{a} \boxplus \overline{b}$.

Multiplication Gates Input: $\overline{a}, \overline{b}$. Output: $\overline{c}$.

1: P_i chooses a random plaintext d_i and broadcasts using $\Pi_{\mathtt{bc}}^{t_s,t_a}$ the ciphertexts $\overline{d_i}$ and $\overline{d_i b}$ and uses the multi-party zero-knowledge functionality $\mathcal{F}_{\text{MZK}}$ to prove knowledge of d_i and that $\overline{d_i b}$ is a correct encryption of the multiplication. Wait for $\max\{T_{bc}, T_{zk}\}$.
2: Let S_i be the subset of the parties succeeding with both proofs. Run n times the protocol $\Pi_{\mathtt{ba}}^{t_s,t_a}$, each one to decide for each party P_j's proof. Input 1 to party j's BA if and only if $j \in S_i$. Wait for T_{ba}. // Crucial to agree on the same S, otherwise privacy breaks.
3: Let S be the subset of the parties for which $\Pi_{\mathtt{ba}}^{t_s,t_a}$ outputs 1.
4: **if** $|S| > t_s$ **then**
5: P_i computes $\overline{a} \boxplus \left(\boxplus_{i \in S} \overline{d_i}\right)$. P_i executes the Threshold Decryption sub-protocol on this ciphertext. Wait for T_{dec}.
6: P_i learns $a + \sum_{i \in S} d_i$ and computes $\overline{c} = \left(a + \sum_{i \in S} d_i\right) \boxdot \overline{b} \boxminus \left(\boxplus_{i \in S} \overline{d_i b}\right)$.
7: **else**
8: Set `abort` $= 1$.
9: **end if**

Output Determination Input x, where $x = c_i$ is the output ciphertext of the circuit if `abort` $= 0$, and otherwise $x = \perp$.

1: P_i executes the protocol $\Pi_{\mathtt{acs}}^{t_s,t_a}$ with x as input. Let S_i be the output of the protocol.
2: **if** $S_i = \{c\}$ **then**
3: Execute the Threshold Decryption sub-protocol on c.
4: After an output is given, terminate.
5: **else**
6: Output $\perp$. // Observe that parties do not terminate, since $\Pi_{\mathtt{acs}}^{t_s,t_a}$ does not guarantee termination.
7: **end if**

Threshold Decryption Input: ciphertext c.

1: P_i computes its decryption share s_i sends it to every other party.
2: P_i proves that the value s_i is a correct decryption share of c bilaterally.
3: Once $t_s + 1$ correct decryption shares are collected, send the list to every party and output the corresponding plaintext.

Theorem 1. *Let n, t_s, t_a be such that $0 \leq t_a < \frac{n}{3} \leq t_s < \frac{n}{2}$ and $t_a + 2t_s < n$. Protocol $\Pi_{\mathtt{smpc}}^{t_s,t_a}$ satisfies the following properties:*

1. *When run in a synchronous network, it achieves full security up to t_s corruptions.*
2. *When run in an asynchronous network, it achieves unanimous output with weak termination up to t_a corruptions and has $n - t_s$ output quality.*

Proof. We prove each of the cases individually. We simulate in the hybrid where there is a trusted setup generating the keys for the PKI, the threshold encryption scheme and the CRS in the real world. In the ideal world, the simulator $\mathcal{S}$ generates the PKI keys, threshold encryption keys and CRS, and outputs the corresponding public keys and CRS to the adversary along with its secret keys.

Case 1: Synchronous network. We describe the simulator $\mathcal{S}$ for the case where the network is synchronous and there are up to t_s corruptions.

- *Input Distribution*: Emulate the messages of the broadcast protocol. This means that, on behalf of each honest party, emulate the broadcast protocol using an encryption of 0 as the input. Also, emulate the $\mathcal{F}_{\text{MZK}}$ functionality by outputting 1 on behalf of each honest parties, and from each corrupted party, on input $(c,(x,r))$ check that $c = \mathtt{Enc}_{\mathtt{ek}}(x,r)$ and output 1 to the adversary and 0 otherwise. The simulator waits for $\max\{T_{bc}, T_{zk}\}$. For each honest party P_i, it keeps track of the correct encrypted inputs I_i that P_i received. If the number of correct ciphertexts is less than $n - t_s$, the simulator does not compute on its ciphertexts on his behalf and sets a local variable $\mathtt{abort}_i = 1$.
- *Addition Gates*: $\mathcal{S}$ simply adds the corresponding ciphertexts locally.
- *Multiplication Gates*: $\mathcal{S}$ emulates the broadcast protocols on random encryptions, and outputs 1 when emulating $\mathcal{F}_{\text{MZK}}$ on behalf of them. For each honest party P_i, keep track of the set of parties S_i succeeding in the proofs. The simulator waits for $\max\{T_{bc}, T_{zk}\}$. Then, emulate the messages in the byzantine agreement protocols and compute the set S. Then it waits for waits for T_{ba}. If the set S is greater than t_s, it computes $\overline{a} \boxplus \left(\boxplus_{i \in S} \overline{d_i}\right)$ and emulates the threshold decryption sub-protocol. After waiting for T_{dec}, it computes the output ciphertext of the multiplication gate. Otherwise, it sets $\mathtt{abort}_i = 1$.
- *Output Determination*: For each party P_i, emulate the messages in the asynchronous common subset protocol with the corresponding input (either a ciphertext, which is the result of the computation, or $\perp$ in the case $\mathtt{abort}_i = 1$). If the output is a single ciphertext c, emulate the threshold decryption sub-protocol.
- *Threshold Decryption*: In a multiplication gate, simply compute the decryption shares and emulate the sending messages. In the Output Determination stage, $\mathcal{S}$ obtains the output y of the computation, and adjusts the shares such that the shares decrypt to y. In both cases, the simulator always outputs 1 on behalf of the honest parties indicating that the proofs of correct decryptions are correct.

Case 2: Asynchronous network. The only difference with respect to the case where the network is synchronous, is that the protocol $\Pi_{\mathtt{bc}}^{t_s,t_a}$ only provides weak-validity. In the simulation, it implies that the simulator will also need to simulate the $\perp$ messages from the broadcast protocols, and not simulate on behalf of the honest parties which stop participating in the protocol after they aborted.

We define a series of hybrids to argue that no environment can distinguish between the real world and the ideal world.

Hybrids and security proof.

Hybrid 1. This corresponds to the real world execution. Here, the simulator knows the inputs and keys of all honest parties.

Hybrid 2. We modify the real-world execution in the zero-knowledge proofs. In the case of a synchronous network, when a corrupted party requests a proof of any kind from an honest party, the simulator simply gives a valid response without checking the witness from the honest party. In the case of an asynchronous network, the simulator is allowed to set outputs to $\perp$ as the real-world adversary.

Hybrid 3. This is similar to Hybrid 2, but the computation of the decryption shares is different. Here, the simulator obtains the output y from the ideal functionality, and if it is not $\perp$, it computes the decryption shares of corrupted parties, and then adjusts the decryption shares of honest parties such that the decryption shares $(d_1, \ldots, d_n)$ form a secret sharing of the output value y.

Hybrid 4. We modify the previous hybrid in the Input Stage. Here, the honest parties, instead of sending an encryption of the actual input, they send an encryption of 0.

Hybrid 5. This corresponds to the ideal world execution.

In order to prove that no environment can distinguish between the real world and the ideal world, we prove that no environment can distinguish between any two consecutive hybrids.

Claim 1. No efficient environment can distinguish between Hybrid 1 and Hybrid 2.
Proof: This follows trivially, since the honest parties always send a valid witness to $\mathcal{F}_{\text{MZK}}$ in the case of a synchronous network. In the case of an asynchronous network, the simulator chooses the set of parties that get $\perp$ as the real-world adversary. ■

Claim 2. No efficient environment can distinguish between Hybrid 2 and Hybrid 3.
Proof: This follows from properties of a secret sharing scheme and the security of the threshold encryption scheme. Given that the threshold is $t_s + 1$, any number corrupted decryption shares below $t_s + 1$ does not reveal anything about the output y. Moreover, one can find shares for honest parties such that $(d_1, \ldots, d_n)$ is a sharing of y. ■

Claim 4. No efficient environment can distinguish between Hybrid 3 and Hybrid 4.
Proof: This follows from the semantic security of the used threshold encryption scheme. ■

Claim 5. No efficient environment can distinguish between Hybrid 4 and Hybrid 5.
Proof: The simulator in the ideal world and the simulator in Hybrid 4 emulate the joint behavior of the ideal functionalities exactly in the same way. ■

We conclude that the real world and the ideal world are indistinguishable.

Finally, let us argue why the protocol has weak termination. Observe that when the protocol outputs $\perp$, parties do not terminate. This is because the protocol $\Pi_{\mathtt{acs}}^{t_s,t_a}$ does not guarantee termination, i.e. might need to run forever (see [9]). However, when parties have agreement on a ciphertext to decrypt (in particular, this is the case when the network is synchronous), the threshold decryption sub-protocol ensures that honest parties can jointly collect $t_s + 1 \leq n - t_s \leq n - t_a$ decryption shares, decrypt the ciphertext and terminate.

□

5 Main Protocol

In this section, we present the protocol $\Pi_{\mathtt{mpc}}^{t_s,t_a}$ for secure function evaluation which tolerates up to t_s (resp. t_a) corruptions when the network is synchronous (resp. asynchronous), for any $0 \leq t_a < \frac{n}{3} \leq t_s < \frac{n}{2}$ satisfying $t_a + 2t_s < n$. The protocol is based on two sub-protocols:

- $\Pi_{\mathtt{smpc}}^{t_s,t_a}$ is a secure function evaluation protocol which gives full security up to t_s corruptions when run in a synchronous network, and achieves unanimous output with weak termination up to t_a corruptions and has $n - t_s$ output quality when run in an asynchronous network.
- $\Pi_{\mathtt{ampc}}^{t_a}$ is a secure function evaluation protocol which gives full security up to t_a corruptions and has $n - t_a$ output quality when run in an asynchronous network.

Protocol $\Pi_{\mathtt{mpc}}^{t_s,t_a}(P_i)$

Let x_i denote the input value of party P_i.

1: Run $\Pi_{\mathtt{smpc}}^{t_s,t_a}$ using x_i as input. Let y_i be the output of P_i.

2: If $y_i \neq \bot$, output y_i and terminate. Otherwise, run $\Pi_{\mathtt{ampc}}^{t_a}$ using x_i as input, output the result and terminate.

Theorem 2. *Let n, t_s, t_a be such that $0 \leq t_a < \frac{n}{3} \leq t_s < \frac{n}{2}$ and $t_a + 2t_s < n$. Protocol $\Pi_{\mathtt{mpc}}^{t_s,t_a}$ satisfies the following properties:*

1. *When run in a synchronous network, it achieves full security up to t_s corruptions.*
2. *When run in an asynchronous network, it achieves full security up to t_a corruptions and has $n - t_s$ output quality.*

Proof. The case where the network is synchronous and there are up to t_s corruptions is trivial, since $\Pi_{\mathtt{smpc}}^{t_s,t_a}$ is guaranteed to provide full security, and $\Pi_{\mathtt{ampc}}^{t_a}$ is never executed. In the other case where the network is asynchronous and there are up to t_a corruptions, observe that after $\Pi_{\mathtt{smpc}}^{t_s,t_a}$ gives output (which is guaranteed to happen), in the case where there is a non-$\bot$ output, every honest party is guaranteed to get this output (which take into account at least $n - t_s$ inputs) and also terminate. If the output is $\bot$, the adversary learned no information so far about the inputs, so it is safe to execute $\Pi_{\mathtt{ampc}}^{t_a}$. In this case, since $\Pi_{\mathtt{ampc}}^{t_a}$ has output quality $n - t_a$, the overall protocol also has $n - t_s \leq n - t_a$ output quality. Observe that in this case the honest parties terminate as soon as $\Pi_{\mathtt{ampc}}^{t_a}$ terminates, since $\Pi_{\mathtt{ampc}}^{t_a}$ guarantees termination.

□

6 Impossibility Proof

We now discuss two lower bounds in this setting. Our first result shows that our feasibility result in Section 5 is tight with respect to the output quality. More concretely, we show that there are basic functions for which it is impossible to achieve both (1) full security up to t corruptions in a synchronous network and (2) $(n - t + 1)$-output quality for even 0 *corruptions* in an asynchronous network. Put simply, a protocol secure against t corruptions cannot rely on receiving more than $n - t$ inputs, even in executions in which all participants happen to be honest.

Our second result shows that the construction presented in Section 5 is tight with respect to the corruption thresholds. That is, we show that there is no protocol for secure function evaluation achieving the guarantees of Theorem 2 when $t_a + 2 \cdot t_s \geq n$. As an example, we show that the majority function cannot be computed with full security up to t_s corruptions in a synchronous network as well as security up to t_a corruptions in an asynchronous network (in fact, in an asynchronous network, it cannot be computed even if we require only unanimous output).

Theorem 3. *Fix any t. There is no protocol Π for MPC with the following properties:*

- *When run in a synchronous network, it achieves full security up to t corruptions.*
- *When run in an asynchronous network, it achieves $(n - t + 1)$-output quality when every party is honest.*

Proof. We show the proof for the case of the OR function. More concretely, the function computes the OR of all the inputs that are received by the ideal functionality (i.e. all inputs that are not $\bot$).

We partition the n parties into two sets S_t, S_{n-t}, where $|S_t| = t$ and $|S_{n-t}| = n - t$. Consider an execution of Π in a synchronous network where parties in S_t are corrupted and abort, and parties in S_{n-t} input 0. In this case, since the protocol achieves full security, all honest parties obtain 0 as output and terminate by some time T.

Next consider an execution of Π in an asynchronous network where all parties are honest, parties in S_t have input 1, and parties in S_{n-t} have input 0. All communication between S_t and S_{n-t} is delayed for more than T clock ticks. Since the view of the parties in S_{n-t} is exactly the same, these parties output 0. This contradicts the fact that Π achieves $(n - t + 1)$-output quality.

□

Theorem 4. *Fix any t_a, t_s such that $t_a + 2 \cdot t_s \geq n$. There is no protocol Π for MPC with the following properties:*

- *When run in a synchronous network, it achieves full security up to t_s corruptions.*
- *When run in an asynchronous network, it achieves unanimous output up to t_a corruptions.*

Proof. **Case 1:** $t_s \geq n/2$ or $t_a \geq n/3$. These bounds follow from classical impossibility results for synchronous and asynchronous MPC protocols with full security (c.f. [16, 7]).

Case 2: $t_s < n/2$, $t_a < n/3$, and $t_a + 2 \cdot t_s \geq n$.

Assume without loss of generality that $t_a + 2 \cdot t_s = n$. We prove the impossibility for the case of the majority function. Partition the n parties into three sets, $S^0_{t_s}$, $S^1_{t_s}$, and S_{t_a}, where $|S^0_{t_s}| = |S^1_{t_s}| = t_s$ and $|S_{t_a}| = t_a$.

First, consider an execution of Π in which the network is synchronous and the t_s parties in $S^1_{t_s}$ are corrupted and crash, and furthermore the honest parties all input 0. Since t_s is less than $n/2$, the protocol must output 0.

Next, consider an execution of Π in which the network is asynchronous, the t_a parties in S_{t_a} are corrupted, and the parties in $S^0_{t_s}$ and $S^1_{t_s}$ input 0 and 1, respectively. In the real world, the adversary can use the following attack: block all messages between $S^0_{t_s}$ and $S^1_{t_s}$ throughout, and have all corrupted parties simulate an honest protocol execution with input $b \in \{0, 1\}$ with the parties in $S^b_{t_s}$. A party in $S^0_{t_s}$ cannot distinguish between this execution and the first execution, and thus the protocol outputs 0; for the same reason a party in $S^1_{t_s}$ outputs 1. By contrast, in the ideal world, the output will of course be the same for all parties. This proves that there is no protocol for the majority function Π that achieves both properties.

References

[1] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync HotStuff: Simple and practical synchronous state machine replication. Cryptology ePrint Archive, Report 2019/270, 2019. `https://eprint.iacr.org/2019/270`.

[2] Judit Bar-Ilan and Donald Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In Piotr Rudnicki, editor, *8th ACM PODC*, pages 201–209. ACM, August 1989.

[3] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *22nd ACM STOC*, pages 503–513. ACM Press, May 1990.

[4] Zuzana Beerliová-Trubíniová and Martin Hirt. Simple and efficient perfectly-secure asynchronous MPC. In Kaoru Kurosawa, editor, *ASIACRYPT 2007*, volume 4833 of *LNCS*, pages 376–392. Springer, Heidelberg, December 2007.

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

[6] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10. ACM Press, May 1988.

[7] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience (extended abstract). In Jim Anderson and Sam Toueg, editors, *13th ACM PODC*, pages 183–192. ACM, August 1994.

[8] Erica Blum, Jonathan Katz, and Julian Loss. Synchronous consensus with optimal asynchronous fallback guarantees. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part I*, volume 11891 of *LNCS*, pages 131–150. Springer, Heidelberg, December 2019.

[9] Erica Blum, Jonathan Katz, and Julian Loss. Network-agnostic state machine replication. Cryptology ePrint Archive, Report 2020/142, 2020. `https://eprint.iacr.org/2020/142`.

[10] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

[11] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

[12] Ran Canetti. Studies in secure multiparty computation and applications. *pp73-79, March*, 1996.

[13] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[14] Ran Canetti and Marc Fischlin. Universally composable commitments. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 19–40. Springer, Heidelberg, August 2001.

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

[16] Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *18th ACM STOC*, pages 364–369. ACM Press, May 1986.

[17] Ran Cohen. Asynchronous secure multiparty computation in constant time. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016, Part II*, volume 9615 of *LNCS*, pages 183–207. Springer, Heidelberg, March 2016.

[18] Ran Cohen and Yehuda Lindell. Fairness versus guaranteed output delivery in secure multiparty computation. *Journal of Cryptology*, 30(4):1157–1186, October 2017.

[19] Sandro Coretti, Juan A. Garay, Martin Hirt, and Vassilis Zikas. Constant-round asynchronous multiparty computation based on one-way functions. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 998–1021. Springer, Heidelberg, December 2016.

[20] Ronald Cramer, Ivan Damgård, Stefan Dziembowski, Martin Hirt, and Tal Rabin. Efficient multiparty computations secure against an adaptive adversary. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 311–326. Springer, Heidelberg, May 1999.

[21] Ronald Cramer, Ivan Damgård, and Ueli M. Maurer. General secure multi-party computation from any linear secret-sharing scheme. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 316–334. Springer, Heidelberg, May 2000.

[22] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. Multiparty computation from threshold homomorphic encryption. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 280–299. Springer, Heidelberg, May 2001.

[23] Ivan Damgård. Efficient concurrent zero-knowledge in the auxiliary string model. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 418–430. Springer, Heidelberg, May 2000.

[24] Ivan Damgård and Yuval Ishai. Constant-round multiparty computation using a black-box pseudorandom generator. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 378–394. Springer, Heidelberg, August 2005.

[25] Ivan Damgård and Mats Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In Kwangjo Kim, editor, *PKC 2001*, volume 1992 of *LNCS*, pages 119–136. Springer, Heidelberg, February 2001.

[26] Ivan Damgård and Jesper Buus Nielsen. Perfect hiding and perfect binding universally composable commitment schemes with constant expansion factor. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 581–596. Springer, Heidelberg, August 2002.

[27] Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 247–264. Springer, Heidelberg, August 2003.

[28] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

[29] Matthias Fitzi, Daniel Gottesman, Martin Hirt, Thomas Holenstein, and Adam Smith. Detectable byzantine agreement secure against faulty majorities. In Aleta Ricciardi, editor, *21st ACM PODC*, pages 118–126. ACM, July 2002.

[30] Matthias Fitzi, Martin Hirt, and Ueli M. Maurer. Trading correctness for privacy in unconditional multiparty computation (extended abstract). In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 121–136. Springer, Heidelberg, August 1998.

[31] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

[32] Yue Guo, Rafael Pass, and Elaine Shi. Synchronous, with a chance of partition tolerance. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 499–529. Springer, Heidelberg, August 2019.

[33] Martin Hirt and Ueli M. Maurer. Robustness for free in unconditional multi-party computation. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 101–118. Springer, Heidelberg, August 2001.

[34] Martin Hirt, Jesper Buus Nielsen, and Bartosz Przydatek. Cryptographic asynchronous multi-party computation with optimal resilience (extended abstract). In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 322–340. Springer, Heidelberg, May 2005.

[35] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure multi-party computation with identifiable abort. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 369–386. Springer, Heidelberg, August 2014.

[36] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 477–498. Springer, Heidelberg, March 2013.

[37] Klaus Kursawe. Optimistic byzantine agreement. In *21st IEEE Symposium on Reliable Distributed Systems, 2002. Proceedings.*, pages 262–267. IEEE, 2002.

[38] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolić. Xft: Practical fault tolerance beyond crashes. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 485–500, 2016.

[39] Chen-Da Liu-Zhang, Julian Loss, Ueli Maurer, Tal Moran, and Daniel Tschudi. Robust MPC: Asynchronous responsiveness yet synchronous security. Cryptology ePrint Archive, Report 2019/159, 2019. `https://eprint.iacr.org/2019/159`.

[40] Julian Loss and Tal Moran. Combining asynchronous and synchronous byzantine agreement: The best of both worlds. Cryptology ePrint Archive, Report 2018/235, 2018. `https://eprint.iacr.org/2018/235`.

[41] Dahlia Malkhi, Kartik Nayak, and Ling Ren. Flexible byzantine fault tolerance. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1041–1053, 2019.

[42] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 31–42. ACM Press, October 2016.

[43] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 223–238. Springer, Heidelberg, May 1999.

[44] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 91. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[45] Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 3–33. Springer, Heidelberg, April / May 2018.

[46] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In *21st ACM STOC*, pages 73–85. ACM Press, May 1989.

[47] Victor Shoup. Practical threshold signatures. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 207–220. Springer, Heidelberg, May 2000.

Supplementary Material

A Universal Composability

We prove security of our protocols in the universal composability (UC) framework [13]. As many other composable frameworks, it follows the real/ideal paradigm. All entities, including parties and the adversary, are modelled via interactive Turing machines.

The goal of a protocol is to emulate an *ideal functionality*, which models a trusted party that receives inputs and provides outputs to the parties. Intuitively, a protocol is proven secure if one shows that for any attack that an adversary can perform in the real protocol, one can construct a corresponding ideal adversary which can perform the same attack in the ideal world via what is called the simulator. The simulator runs in the ideal world, interacting only with the ideal functionality and the real adversary, and has to be such that the distributions of messages seen in the real world and ideal world executions are indistinguishable from the point of view of an external entity called *the environment*. The environment has total control over the adversary, and can choose the inputs, and see the outputs of all parties.

Let us denote by $\mathbf{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}(1^\kappa, z)$ the output distribution of the environment $\mathcal{Z}$ in the real world execution of protocol Π, with n parties and real-world adversary $\mathcal{A}$, and κ is the security parameter and z is the auxiliary input to $\mathcal{Z}$. Similarly, we denote the output distribution of $\mathcal{Z}$ when interacting with the ideal world as $\mathbf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(1^\kappa, z)$, where $\mathcal{F}$ is the ideal functionality and $\mathcal{S}$ is the simulator. Additionally, we denote the hybrid execution of a protocol Π, which is given access to an ideal functionality $\mathcal{G}$, by $\mathbf{HYB}^{\mathcal{G}}_{\Pi,\mathcal{A},\mathcal{Z}}(1^\kappa, z)$. This is defined similarly to the real execution, and is known as the $\mathcal{G}$-hybrid model. Security of a protocol is then defined as follows.

Definition 8. *A protocol Π UC-securely realizes an ideal functionality $\mathcal{F}$ in the $\mathcal{G}$-hybrid model if for any PPT adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that for any PPT environment $\mathcal{Z}$, it holds that:*

$$\mathbf{HYB}^{\mathcal{G}}_{\Pi,\mathcal{A},\mathcal{Z}} \approx_c \mathbf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}.$$

The composition theorem provides security guarantees when protocols are composed in an arbitrary way. This means that if ρ is a UC-secure protocol realizing $\mathcal{G}$, then the protocol Π in the $\mathcal{G}$-hybrid model can be replaced by the composition $\Pi \circ \rho$. Informally, the composition theorem then guarantees that $\mathbf{REAL}_{\Pi\circ\rho,\mathcal{A},\mathcal{Z}}$ is indistinguishable from $\mathbf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$.

The default real-world in the UC framework is inherently an asynchronous setting, where the adversary is allowed to drop messages. We are, however, interested in a synchronous setting and an asynchronous setting with *eventual delivery*. For such settings, proper extended hybrid UC-functionalities have been made in the literature. We include some of the references to synchronous UC [36, 13, 39], and asynchronous UC with eventual delivery [19, 39].

B Paillier Cryptosystem

In this section we describe the Paillier cryptosystem [43]. The public key $\mathtt{pk}$ is a k-bit RSA modulus $N = pq$, where p, q have $\frac{k}{2}$ bits and are such that $p = 2p' + 1$, $q = 2q' + 1$ for p', q' primes. The secret key is $\mathtt{sk} = \phi(N)(\phi(N)^{-1} \mod N)$.

In order to encrypt a message $a \in \mathbf{Z}_N$, one computes the ciphertext $\overline{a} = \mathtt{Enc}_{\mathtt{pk}}(a, r) = g^a r^N \mod N^2$, where $r \in \mathbf{Z}^*_N$ is chosen uniformly at random, and $g = N+1$. To decrypt a message, one simply computes $c^{\mathtt{sk}} \mod N^2 = Na + 1$, from which $a \mod N$ can be obtained.

The encryption scheme is additively homomorphic in the sense that $\overline{a} \boxplus \overline{b} = \mathtt{Enc}_{\mathtt{pk}}(a, r_a) \cdot \mathtt{Enc}_{\mathtt{pk}}(b, r_b) = \mathtt{Enc}_{\mathtt{pk}}(a + b, r_a r_b)$.

Semantic security can be shown under the so-called decisional composite residual assumption (DCRA), which states that random elements in $\mathbf{Z}^*_{N^2}$ are computationally indistinguishable from random elements of the form r^N.

A threshold version of this cryptosystem can be found in [25], based on a variant of Shoup's technique [47] for threshold RSA.