MPC with Synchronous Security and Asynchronous Responsiveness

Chen-Da Liu-Zhang[1], Julian Loss[2], Ueli Maurer[1], Tal Moran[3]⋆, and Daniel Tschudi[4] ⋆⋆

[1] `{lichen,maurer}@inf.ethz.ch`, ETH Zurich
[2] `lossjulian@gmail.com`, University of Maryland
[3] `talm@idc.ac.il`, IDC Herzliya
[4] `dt@concordium.com`, Concordium

Abstract. Two paradigms for secure MPC are synchronous and asynchronous protocols. While synchronous protocols tolerate more corruptions and allow every party to give its input, they are very slow because the speed depends on the conservatively assumed worst-case delay Δ of the network. In contrast, asynchronous protocols allow parties to obtain output as fast as the actual network allows, a property called *responsiveness*, but unavoidably have lower resilience and parties with slow network connections cannot give input.

It is natural to wonder whether it is possible to leverage synchronous MPC protocols to achieve responsiveness, hence obtaining the advantages of both paradigms: full security with responsiveness up to t corruptions, and *extended* security (full security or security with unanimous abort) with no responsiveness up to $T \geq t$ corruptions. We settle the question by providing matching feasibility and impossibility results:

- For the case of unanimous abort as extended security, there is an MPC protocol if and only if $T + 2t < n$.
- For the case of full security as extended security, there is an MPC protocol if and only if $T < \frac{n}{2}$ and $T + 2t < n$. In particular, setting $t = \frac{n}{4}$ allows to achieve a fully secure MPC for honest majority, which in addition benefits from having substantial responsiveness.

1 Introduction

In the context of multiparty computation (MPC), a set of mutually distrustful parties wish to jointly compute a function by running a distributed protocol. The protocol is deemed secure if every party obtains the correct output and if it does not reveal any more information about the parties' inputs than what can be inferred from the output. Moreover, these guarantees should be met even if some of the parties can maliciously deviate from the protocol description. Broadly speaking, MPC protocols exist in two regimes of synchrony. First, there are *synchronous* protocols which assume that parties share a common clock and

⋆ Supported in part by ISF grant no. 1790/13 and by the Bar-Ilan Cyber-center.
⋆⋆ Author was supported by advanced ERC grant MPCPRO.

messages sent by honest parties can be delayed by at most some a priori known bounded time. Synchronous protocols typically proceed in rounds of length Δ, ensuring that any message sent at the beginning of a round by an honest party will arrive by the end of that round at its intended recipient. On the upside, such strong timing assumptions allow to obtain protocols with an optimal resilience of $\frac{1}{2}n$ corruptions for the case of *full security* [5, 13, 45, 27, 2, 20], and of arbitrary number of corruptions for the case of *security with (unanimous) abort* and no fairness [23, 29]. On the downside, especially in real-world networks where the *actual* maximal network delay δ is hard to predict, Δ has to be chosen rather pessimistically, and synchronous protocols fail to take advantage of a fast network.

The second type of protocols that we will study in this work are *asynchronous* protocols. Such protocols do not require synchronized clocks or an a priori known bounded network delay to work properly. As such, they function correctly under much more realistic network assumptions. Moreover, asynchronous protocols have the benefit of running at the *actual speed of the network*, i.e., they run in time that depends only on δ, but *not* on Δ; a notion that we shall refer to as *responsiveness* [41]. This speed and robustness comes at a price, however: it can easily be seen that no asynchronous protocol that implements an arbitrary function can tolerate $\frac{1}{3}n$ maliciously corrupted parties [6]. We ask the natural question of whether it is possible to leverage synchronous MPC protocols to also achieve responsiveness:

Is there a (synchronous) MPC protocol that allows to simultaneously achieve full security with responsiveness up to t corruptions, and some form of extended security (full security, unanimous abort) up to $T \geq t$ corruptions?

We settle the question with tight feasibility and impossibility results:

– For the case where unanimous abort is required as extended security, this is possible if and only if $T + 2t < n$.
– For the case where full security is required as extended security, this is possible if and only if $T < \frac{n}{2}$ and $T + 2t < n$.

1.1 Technical Overview of Our Results

The Model. We first introduce a new composable model of functionalities in the UC framework [11], which captures the guarantees that protocols from both asynchronous and synchronous worlds achieve in a very general fashion. Our model allows to capture multiple distinct guarantees such as privacy, correctness, or responsiveness, each of which is guaranteed to hold for (possibly) different thresholds of corruption. In contrast to previous works, we do not capture the guarantees as protocol properties, but rather as part of the ideal functionality. This allows to use the ideal functionality as an assumed functionality in further steps of the composition, without the need to keep track of the properties of the real-world protocols.

Real World Functionalities. Our protocols work with public-key infrastructure (PKI) and common-reference string (CRS) as setup. Parties have access to a synchronized global clock functionality $\mathcal{G}_{\text{CLK}}$ and a communication network of authenticated channels with *unknown* upper bound δ, corresponding to the maximal network delay. This value is unknown to the honest parties. Instead, protocols make use of a conservatively assumed worst-case delay $\varDelta \gg \delta$. Within δ, the adversary can schedule the messages arbitrarily.

Ideal Functionality. In order to capture the guarantees that asynchronous and synchronous protocols achieve in a fine-grained manner, we describe an ideal functionality $\mathcal{F}_{\text{HYB}}$ which allows parties to jointly evaluate a function. At a high level, $\mathcal{F}_{\text{HYB}}$ is composed of two phases; an asynchronous and a synchronous phase, separated by some pre-defined time-out. Each party can obtain a unique identical output in either phase. As in asynchronous protocols, the outputs obtained during the asynchronous phase are obtained fast, i.e., at a time which depends on the actual maximal network delay δ, but not on the conservatively assumed worst-case network delay $\varDelta$. Let us describe the guarantees that $\mathcal{F}_{\text{HYB}}$ provides.

If there are up to t corruptions, $\mathcal{F}_{\text{HYB}}$ achieves full security with responsiveness. That is, honest parties obtain a correct and identical output, and honest parties' inputs remain private. Moreover, they obtain an output $y_{\texttt{asynch}}$ by a time proportional to the actual network delay δ. Unavoidably, this means that $\mathcal{F}_{\text{HYB}}$ may ignore up to t inputs from honest parties.

If there are up to $T \geq t$ corruptions, $\mathcal{F}_{\text{HYB}}$ can give output at two different points in time $\tau_1 \leq \tau_2$. Either all parties obtain $y_{\texttt{asynch}}$ before time τ_1 (there might be some parties which obtained $y_{\texttt{asynch}}$ in the asynchronous phase), or all parties obtain the output $y_{\texttt{sync}}$ by time τ_2, which is guaranteed to take into account all inputs from honest parties. For the output $y_{\texttt{sync}}$, we consider two versions: $\mathcal{F}^{\texttt{fs}}_{\text{HYB}}$ which guarantees full security up to T corruptions implying that $y_{\texttt{sync}}$ is the correct output, and $\mathcal{F}^{\texttt{ua}}_{\text{HYB}}$ which guarantees security with unanimous abort up to T corruptions, meaning that the adversary can set $y_{\texttt{sync}}$ to $\perp$.

We depict in Figure 1 a time-line showing the point in time at which the honest parties obtain the output, depending on the number of corruptions.

Black-Box Compiler. We give a generic black-box compiler that combines an asynchronous MPC protocol with a synchronous MPC protocol and gives a hybrid protocol that combines beneficial properties from both the synchronous and asynchronous regime, very roughly in the following way: Using threshold encryption and assuming 1) a *two-threshold* asynchronous protocol with full security up to t corruptions and security with no termination (correctness and privacy) up to $T \geq t$ corruptions, and 2) a synchronous protocol with *extended* security (full security or security with unanimous abort) up to T corruptions, the compiler provides full security with responsiveness up to t corruptions, and extended security up to T corruptions, for any $T + 2t < n$.

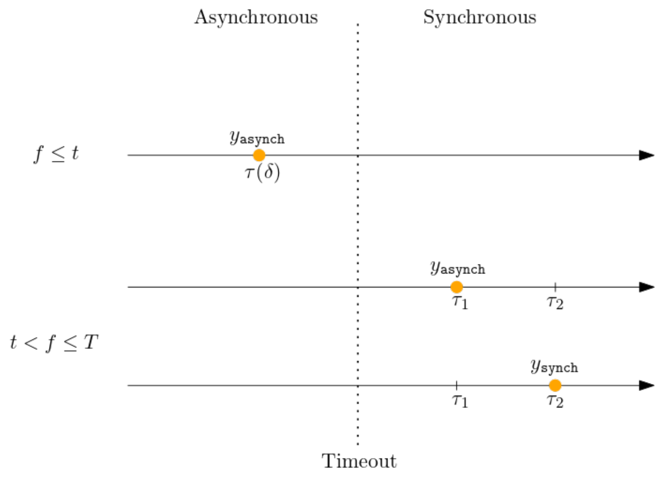

Fig. 1. The dotted vertical line separates the asynchronous and the synchronous phase. The orange dot shows the latest point in time when honest parties get output. The output $y_{\mathtt{asynch}}$ takes into account $n - t$ inputs, whereas $y_{\mathtt{sync}}$ takes into account all inputs. Up to t corruptions all parties obtain $y_{\mathtt{asynch}}$ fast. In the other case, either all parties obtain $y_{\mathtt{asynch}}$ by τ_1, or all parties obtain $y_{\mathtt{sync}}$ by τ_2, which is the correct output for $\mathcal{F}^{\mathtt{fs}}_{\mathrm{HYB}}$, and may be $\bot$ for $\mathcal{F}^{\mathtt{ua}}_{\mathrm{HYB}}$.

For the first sub-protocol 1), we show how to modify the asynchronous MPC protocol by Cohen [18] to obtain the trade-off mentioned above when used in our aforementioned compiler. We separate the termination threshold from all other security guarantees. That is, we achieve an asynchronous protocol that terminates (in a responsive and fully-secure manner) for any $t < \frac{1}{3}n$, and provides security without termination up to $T < n - 2t$ corruptions.

The second sub-protocol 2) can be achieved with known protocols; for $T < n$ in the case of security with unanimous abort (e.g. [23, 29]) and for $T < n/2$ for full security (e.g. [5, 13, 45, 27, 2, 20]).

Compiler Description. We now give an outline of our compiler. At a high level, the idea of our compiler is to first run an asynchronous protocol until some pre-defined timeout. Upon timing out, the parties switch to a synchronous computation. If sufficiently many parties are honest, the honest parties obtain their output at the actual speed of network. The main challenge is to ensure that if even a single party obtains output during the asynchronous phase, the output will not be changed during the synchronous phase. This would be problematic for two reasons: First, because the combined protocol would offer no improvement over a standard synchronous protocol in terms of responsiveness; if a party does not know if the output it obtains during the asynchronous phase will be later changed during the synchronous phase, then this output is essentially useless to that party. Therefore, if this were indeed the case, then one could run *just* the synchronous part of the protocol. Second, computing two different outputs may be problematic for privacy reasons, as two different outputs give the adversary more information about the honest parties' inputs than what it should be able to infer. Our solution to this problem is to have the asynchronous protocol output a *threshold ciphertext* $[y]$ *of the actual output* y. Prior to running the hybrid protocol, the parties each obtain a key share d_i such that k out of n parties can jointly decrypt the ciphertext by pooling their shares. This way, if we set $k = n - t$, where t is the responsiveness threshold, we are ensured that sufficiently many parties will pool their shares during the asynchronous phase, given that fewer than t parties are corrupt. Therefore, every honest party should be able to decrypt and learn the output during the asynchronous phase, thus ensuring responsiveness. On the other hand, our compiler ensures that if any honest party gives out its share during the asynchronous phase after seeing the ciphertext $[y]$ being output by the asynchronous protocol, then the only possible output during the synchronous phase can be y. Finally, our compiler has a mechanism to detect whether no honest party has made its share public yet. In this case, we can safely recompute the result during the synchronous phase of the hybrid protocol, as we can be certain that the adversary does not have sufficient shares to learn the output from the asynchronous phase.

Two-Threshold Asynchronous MPC Protocol. Finally, in Section 5, we show how to obtain an asynchronous MPC protocol to achieve trade-offs between termination and security (correctness and privacy). While many asynchronous MPC protocols (e.g. [43, 15, 14, 18, 32]) can be adapted to the two-threshold setting, we choose to adapt the protocol in [18] for simplicity.

The protocol in [18] achieves all guarantees simultaneously for the corruption threshold $\frac{1}{3}n$. At a high level, the idea of this protocol is to use a threshold fully homomorphic encryption scheme (TFHE) with threshold $k = \frac{1}{3}n$ and let parties distribute encryption shares of their inputs to each other. Then, parties agree on a common set of at least $\frac{2}{3}n$ parties, whose inputs will be taken into account during the function evaluation. In this step, n Byzantine Agreement protocols are run. Parties can then locally evaluate the function which is to be computed on their respective input shares by carrying out the corresponding (homomorphic) arithmetic operations on these shares. After this local computation has succeeded, parties pool their shares of the computation's result to decrypt the final output of the protocol. We modify the thresholds in this protocol in the following manner. Instead of setting $k = \frac{1}{3}n$, we set $k = \frac{3}{4}n$. Intuitively, assuming a perfect Byzantine Agreement (BA) functionality, this modification has the effect that the adversary needs to corrupt $\frac{3}{4}n$ parties to break privacy, but can prevent the protocol from terminating by withholding decryption shares whenever it corrupts more than $\frac{1}{4}n$ parties. However, one can see that if one realizes the BA functionality using a traditional protocol with validity and consistency thresholds $\frac{1}{3}n$, the overall statement will only have security $\frac{1}{3}n$.

We show how to improve the security threshold T of the protocol by using, as a sub-component, an asynchronous BA protocol which trades liveness for consistency without sacrificing validity. Our protocol inherits the thresholds of the improved BA protocol, achieving any $T < n - 2t$, where t is the termination threshold.

1.2 Synchronous Protocols over an Asynchronous Network

We argue that it is not trivial to enhance a synchronous MPC protocol to achieve responsiveness. Two ways to execute a synchronous protocol over a network with unknown delay δ are as follows:

Time-Out Based. Perhaps the easiest approach to execute a synchronous protocol over this network is to model each round using $\varDelta$ clock ticks, where $\varDelta$ is a known upper bound on the network delay. In this case, the output is obtained at a time which depends on $\varDelta$. Note that $\varDelta$ has to be set high enough to accommodate any conditions, and such that any honest party has enough time to perform its local computation; if an honest party is slightly later than $\varDelta$ in any round, it will be considered corrupted throughout the whole computation. In realistic settings where δ is hard to predict, we will have that $\varDelta \gg \delta$. Hence, any synchronous protocol (even constant-round) is slow.

Notification Based. A well-known approach (see e.g. [39]) to "speed up" a synchronous protocol is to let the parties simulate a synchronized clock in an event-based fashion over an asynchronous network. More concretely, the idea is that each party broadcasts a notification once it finishes a particular round i and only advances to round $i + 1$ upon receiving a notification for round i from all parties. It is not hard to see that this approach does not achieve the responsiveness guarantees we aim for. To this end, observe that a **single** corrupted party P_j can make all parties wait $\varDelta$ clock ticks in each round, simply by not sending a notification in this particular round. Note that parties cannot infer that P_i is corrupted, unless they wait for $\varDelta$ clock ticks, because δ is unknown. Hence, unless there are *no corruptions*, an approach along these lines can not ensure responsiveness. In contrast, our protocol guarantees that parties obtain fast outputs as long as there are up to t corruptions.

1.3 Related Work

Despite being a very natural direction of research, compilers for achieving tradeoffs between asynchronous and synchronous protocol have only begun to be studied in relatively recent works.

Pass and Shi study a hybrid type of state-machine replication (SMR) protocol in [41] which confirms transactions at an asynchronous speed and works in the model of *mildly adaptive* malicious corruptions; such corruptions take a short time to take effect and as such model a slightly weaker adversary than one that is fully adaptive. Subsequently, Pass and Shi show a general paradigm for SMR protocols with optimistic confirmation of transactions called *Thunderella* [42]. In their work, they show how to achieve optimistic transaction confirmation (at asynchronous network speed) as long as the majority of some designated committee and a party called the 'accelerator' are honest and faithfully notarize transactions for confirmation. If the committee or the accelerator become corrupted, the protocol uses a synchronous SMR protocol to recover and eventually switch back to the asynchronous path of the protocol. Their protocol achieves safety and liveness against a fully adaptive adversary, but can easily be kept on the slow, synchronous path forever in this case. Subsequently, Loss and Moran [40] showed how to obtain compilers for the simpler case of BA that achieve tradeoffs between responsiveness and safety against a fully adaptive adversary.

The work by Guo et al. [30] introduced a model which weakens classical synchrony. There, the adversary can interrupt the communication between certain sets of parties, as long as in each round there is a (possibly different) connected component with an honest majority of the nodes. Although their focus is not on responsive protocols, the authors include an MPC responsive protocol, based on threshold FHE for the case of full-security as extended security. Our protocols differ from theirs in various aspects: 1) In contrast to their protocol, our approach is conceptually simpler and allows to plug-in any asynchronous and synchronous protocol in a black-box manner and automatically inherit the thresholds for each of the guarantees, and the assumptions from each of the protocols. For example, we can plug-in a synchronous protocol with full security and unanimous abort, and obtain the corresponding guarantees; one could further consider other types of guarantees, or design MPC protocols from different types of assumptions which would all be inherited automatically from our compiler; 2) We phrase all our results in the UC framework and capture in a very general fashion the guarantees that the protocol provides as part of the ideal functionality. This leads to some differences, e.g. our ideal functionality allows to capture responsiveness guarantees; also allows to take into account in the computation the inputs from all parties in some cases.

Further Related Work. Best-of-both worlds compilers for distributed protocols (in particular MPC protocols) come in many flavours and we are only able to list an incomplete summary of related work. Goldreich and Petrank [28] give a black-box compiler for Byzantine agreement which focuses on achieving protocols which have expected constant round termination, but in the worst case terminate after a fixed number of rounds. Kursawe [38] gives a protocol for Byzantine agreement that has an optimistic *synchronous path* which achieves Byzantine agreement if every party behaves honestly and the network is well-behaved. If the synchronous path fails, then parties fall back to an asynchronous path which is robust to network partitions. However, the overall protocol tolerates only $\frac{1}{3}n$ corrupted parties in order to still achieve safety and liveness. A recent line of works [7, 8, 9] studied protocols resilient to t_2 corruptions when run in a synchronous network and also to t_1 corruptions if the network is asynchronous, for $0 < t_1 < \frac{1}{3}n \leq t_2 < \frac{1}{2}n$. A line of works [3, 4, 16, 44] consider the setting where parties have a few synchronous rounds before switching to fully asynchronous computation. Here, one can achieve protocols with better security guarantees than purely asynchronous ones. Finally, the line of works [24, 34, 35, 25, 31] consider different thresholds to achieve more fine-grained security guarantees.

Worth mentioning, are the works of [34, 35], which consider MPC protocols with full security up for an honest majority t, and security with abort for a dishonest majority T. Our protocols achieve results in this direction as well, except that our threshold t includes responsiveness as well. Note that the impossibility of [35], where it is shown that $T + t \geq n$ is impossible does not apply to our work, since we consider a *weaker* trade-off $T + 2t < n$. Moreover, the fact that our threshold t for full security case includes responsiveness as well is essential to prove that the bound $T + 2t < n$ is tight.

2 Preliminaries

Threshold Encryption Scheme. We assume the existence of a secure public-key encryption scheme which enables threshold decryption.

Definition 1. *A threshold encryption scheme is a public-key encryption scheme which has the following two additional properties:*

- *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.*
- *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 $m = \mathtt{Rec}(\{d_i\})$.*

Digital Signature Scheme. We assume the existence of a digital signature scheme unforgeable against adaptively chosen message attacks. Given a signing key $\mathtt{sk}$ and a verification key $\mathtt{vk}$, let $\mathsf{Sign}_{\mathtt{sk}}$ and $\mathsf{Ver}_{\mathtt{vk}}$ the signing and verification functions. We write $\sigma = \mathsf{Sign}_{\mathtt{sk}}(m)$ meaning using $\mathtt{sk}$, sign a plaintext m to obtain a signature σ. Moreover, we write $\mathsf{Ver}_{\mathtt{vk}}(m, \sigma) = 1$ to indicate that σ is a valid signature on m.

3 Model

Notation. We denote by κ the security parameter, $\mathcal{P} = \{P_1, \ldots, P_n\}$ the set of n parties and by $\mathcal{H}$ the set of honest parties.

3.1 Adversary

We consider a static adversary, who can corrupt up to f parties at the onset of the execution and make them deviate from the protocol arbitrarily. The adversary is also computationally bounded.

3.2 Communication Network and Clocks

We borrow ideas from a standard model for UC synchronous communication [36, 37]. Parties have access to functionalities and global functionalities [12]. More concretely, parties have access to a synchronized global clock functionality $\mathcal{G}_{\text{CLK}}$, and a network functionality $\mathcal{F}^{\delta}_{\text{NET}}$ of pairwise authenticated channels with an unknown upper bound on the message delay δ.

At a high level, the model captures the two guarantees that parties have in the synchronous model of communication. First, every party must be activated each clock tick, and second, every party is able to perform all its local computation before the next tick. Both guarantees are captured via the clock functionality $\mathcal{G}_{\text{CLK}}$. It maintains the global time τ, initially set to 0, and a round-ready flag $d_i = 0$, for each party P_i. Each clock tick, $\mathcal{G}_{\text{CLK}}$ sets the flag to $d_i = 1$ whenever a party sends a confirmation (that it is ready) to the clock. Once the flag is set for every honest party, the clock counter is increased and the flags are reset to 0 again. This ensures that all honest parties are activated in each clock tick.

Functionality $\mathcal{G}_{\text{CLK}}$

The clock functionality stores a counter τ, initially set to 0. For each honest party P_i it stores flag d_i, initialized to 0.

ReadClock:

```
1: On input (ReadClock), return $\tau$.
```

Ready:

```
1: On input (ClockReady) from honest party $P_i$ set $d_i = 1$ and notify the
   adversary.
```

ClockUpdate: Every activation, the functionality runs the following code before doing anything else:

```
1: if for every honest party $P_i$ it holds $d_i = 1$ then
2:     Set $d_i = 0$ for every honest party $P_i$ and $\tau = \tau + 1$.
3: end if
```

The UC standard communication network does not consider any delivery guarantees. Hence, we consider the functionality $\mathcal{F}^{\delta}_{\text{NET}}$ which models a complete network of pairwise authenticated channels with an *unknown* upper bound δ corresponding to the real delay in the network. The network is connected to the clock functionality $\mathcal{G}_{\text{CLK}}$. It works in a *fetch-based* mode: parties need to actively query for the messages in order to receive them. For each message m sent from P_i to P_j, $\mathcal{F}_{\text{NET}}$ creates a unique identifier $\texttt{id}_m$ for the tuple $(T_{\texttt{init}}, T_{\texttt{end}}, P_i, P_j, m)$. This identifier is used to refer to a message circulating the network in a concise way. The field $T_{\texttt{init}}$ indicates the time at which the message was sent, whereas $T_{\texttt{end}}$ is the time at which the message is made available to the receiver. At first, the time $T_{\texttt{end}}$ is initialized to $T_{\texttt{init}} + 1$.

Whenever a new message is input to the buffer of $\mathcal{F}_{\text{NET}}$, the adversary is informed about both the content of the message and its identifier. It is then allowed to modify the delivery time $T_{\texttt{end}}$ by any finite amount. For that, it inputs an integer value T along with some corresponding identifier $\texttt{id}_m$ with the effect that the corresponding tuple $(T_{\texttt{init}}, T_{\texttt{end}}, P_i, P_j, m)$ is modified to $(T_{\texttt{init}}, T_{\texttt{end}} + T, P_i, P_j, m)$. Moreover, to capture that there is an upper bound on the delay of the messages, the network does not accept more than δ accumulated delay for any identifier $\texttt{id}_m$. That is, $\mathcal{F}_{\text{NET}}$ checks that $T_{\texttt{end}} \leq T_{\texttt{init}} + \delta$. Also, observe that the adversary has the power to schedule the delivery of messages: we allow it to input delays more than once, which are added to the current amount of delay. If the adversary wants to deliver a message during the next activation, it can input a negative delay. We remark, that the traditional model of an asynchronous network with eventual delivery can be modeled by setting $\delta = \infty$.

Functionality $\mathcal{F}_{\text{NET}}^{\delta}$

The functionality is connected to a clock functionality $\mathcal{G}_{\text{CLK}}$. It is parameterized by a positive constant δ (the real delay upper bound only known to the adversary). It also stores the current time τ and keeps a buffer of messages `buffer` which initially is empty.
Each time the functionality is activated it first queries $\mathcal{G}_{\text{CLK}}$ for the current time and updates τ accordingly.

Message transmission:

1: At the onset of the execution, output δ to the adversary.
2: On input $(\textsc{Send}, i, j, m)$ from party P_i, $\mathcal{F}_{\text{NET}}$ creates a new identifier $\texttt{id}_m$ and records the tuple $(\tau, \tau+1, P_i, P_j, m, \texttt{id}_m)$ in `buffer`. Then, it sends the tuple $(\textsc{Sent}, P_i, P_j, m, \texttt{id}_m)$ to the adversary.
3: On input $(\textsc{FetchMessages}, i)$ from P_i, for each message tuple $(T_{\texttt{init}}, T_{\texttt{end}}, P_k, P_i, m, \texttt{id}_m)$ from `buffer` where $T_{\texttt{end}} \leq \tau$, the functionality removes the tuple from `buffer` and outputs (k, m) to P_i.
4: On input $(\textsc{Delay}, D, \texttt{id})$ from the adversary, if there exists a tuple $(T_{\texttt{init}}, T_{\texttt{end}}, P_i, P_j, m, \texttt{id})$ in `buffer` and $T_{\texttt{end}} + D \leq T_{\texttt{init}} + \delta$, then set $T_{\texttt{end}} = T_{\texttt{end}} + D$ and return $(\textsc{Delay-ok})$ to the adversary. Otherwise, ignore the message.

3.3 Ideal World

We introduce ideal functionality $\mathcal{F}_{\text{HYB}}^{\texttt{fs}}$ (resp. $\mathcal{F}_{\text{HYB}}^{\texttt{ua}}$) which allows to capture the guarantees that asynchronous and synchronous protocols for secure function evaluation offer in a fine-grained manner. The functionality has access to the global functionality $\mathcal{G}_{\text{CLK}}$, and allows parties to evaluate a function f. The idea is that up to t corruptions, parties have full security and responsiveness. Moreover, in the case of $\mathcal{F}_{\text{HYB}}^{\texttt{fs}}$, if up to $t \leq T < n/2$ parties are corrupted, full security is guaranteed, i.e. all honest parties obtain the correct and identical output, and the inputs from honest parties remain secret. The functionality $\mathcal{F}_{\text{HYB}}^{\texttt{ua}}$ is the same, except that it guarantees security with unanimous abort up to $t \leq T < n$ corruptions instead of full security, i.e., honest parties obtain the correct output or unanimously obtain $\perp$.

The number of inputs that the function is guaranteed to take into account and the time at which it provides output depends the number of corruptions. The time-out divides the execution into two phases: an asynchronous and a synchronous phase.

- If there are up to t corruptions, parties are guaranteed to obtain an output at time $\tau_{\texttt{asynch}}$, which depends on δ. This fast output is identical to every party and is guaranteed to take into account at least $n-t$ inputs, i.e. can ignore the inputs from up to t honest parties.
- Otherwise, the parties are guaranteed to obtain the same output, but at a time which depends on $\varDelta$. More concretely, there are two latest points in time at which parties can obtain an output after the time-out occurs: $\tau_{\texttt{OD}} < \tau_{\texttt{OND}}$. Either all parties obtain the output by $\tau_{\texttt{OD}}$, which is guaranteed to take into account $n-t$ inputs, or all parties obtain output at a later time $\tau_{\texttt{OND}}$, which is guaranteed to take into account all inputs.

The adversary can in addition gain certain capabilities depending on the amount of corruption it performs. More technically, we introduce a tamper function Tamper, parametrized by a tuple of thresholds (t, T). This allows to naturally capture the different guarantees for the two corruption thresholds t and T. Basically, if the number of corruptions is greater than t, the adversary can prevent the parties to obtain fast outputs. And beyond T, no security guarantee is ensured, as the adversary learns the inputs from the honest parties and can choose the outputs as well.

Tamper Function. The ideal functionality is parameterized by a tamper function, which indicates the adversary's capabilities depending on the threshold. We consider two thresholds: T for full security, and t for responsiveness.

Definition 2. *We define the ideal functionality with parameters* (t, T) *if it has the following tamper function* $\mathsf{Tamper}^{\textsc{Hyb}}_{t,T}$*:*

Function $\mathsf{Tamper}^{\textsc{Hyb}}_{t,T}$

// Flags indicating violation of c correctness, p privacy, r responsiveness $(c, p, r) = \mathsf{Tamper}^{\textsc{Hyb}}_{t,T}$*, where:*
- $c = 1$*,* $p = 1$ *if and only if* $|\mathcal{P} \setminus \mathcal{H}| > T$.
- $r = 1$ *if and only if* $|\mathcal{P} \setminus \mathcal{H}| > t$

The ideal functionality has in addition a set of parameters. It contains a parameter $\tau_{\texttt{asynch}}$ which models the maximum output delay in the asynchronous phase, and parameters $\tau_{\texttt{OD}}$ and $\tau_{\texttt{OND}}$ which model the output delays for an output that takes into account $n-t$ inputs, or an output with all the inputs. One can think of $\tau_{\texttt{asynch}} = O(\delta)$, and $\tau_{\texttt{OD}} < \tau_{\texttt{OND}}$ are times which depend on $\varDelta$.

In addition, it keeps the following local variables:

- `FastOutput` indicates if the output contains $n-t$ inputs or all inputs.
- τ keeps the current time.
- $\tau_{\texttt{tout}}$ is the pre-defined time-out to switch between the two phases.
- `sync` indicates the phase being executed (asynchronous or synchronous).
- x_i, y_i the input and output for party P_i.
- w_i indicates if the adversary decided to not deliver output y_i in the asynchronous phase. The adversary can only use this capability if the number of corruptions is larger than t.
- $\mathcal{I}$ keeps the set of parties whose input are taken into account for the fast output.

Functionality $\mathcal{F}_{\text{HYB}}^{\texttt{fs}}$

The functionality is connected to a global clock $\mathcal{G}_{\text{CLK}}$.
The functionality is parametrized by δ, $\tau_{\texttt{asynch}}$, $\tau_{\texttt{OD}}$, $\tau_{\texttt{OND}}$, Tamper, $\tau_{\texttt{tout}}$ and the function to evaluate f.
The functionality stores variables `FastOutput`, τ, `sync`, x_i, y_i, w_i. These variables are initialized as `FastOutput` = `false`, $\tau = 0$, `sync` = `false`, $x_i = \perp$, and $y_i = w_i = \perp$.
It keeps $\mathcal{I} = \mathcal{H}$, where $\mathcal{H}$ is the set of honest parties, and a set $\mathcal{C} = \varnothing$.

Timeout/Clock :

Each time the functionality is activated, query $\mathcal{G}_{\text{CLK}}$ for the current time and update τ accordingly.
If $\tau \geq \tau_{\texttt{tout}}$, set `sync` = `true`. If `FastOutput` = `false`, compute $y_1 = \cdots = y_n = f(x_1, \ldots, x_n)$.

Asynchronous Phase If `sync` = `false` do the following:

- At the onset of the execution, output δ and $\tau_{\texttt{asynch}}$ to the adversary.
- On input $(\text{Input}, v_i, \text{sid})$ from party P_i:
  - If some party has received output, ignore this message. Otherwise, set $x_i = v_i$.
  - If $x_i \neq \perp$ for each $P_i \in \mathcal{I}$, set each output to $y_j = f(x'_1, \ldots, x'_n)$, where $x'_i = x_i$ for each $P_i \in \mathcal{I} \cup (\mathcal{P} \setminus \mathcal{H})$ and $x'_i = \perp$ otherwise.
  - Output $(\text{Input}, P_i, \text{sid})$ to the adversary.
- On input $(\text{GetOutput}, \text{sid})$ from P_i do the following:
  - If the output has not been set yet or is blocked, i.e., $y_i = \perp$ or w_i = `aBlocked`, ignore this message.
  - If $\tau \geq \tau_{\texttt{asynch}}$ output $(\text{output}, y_i, \text{sid})$ to P_i and set `FastOutput` = `true`.
  - Otherwise, output $(\text{output}, P_i, \text{sid})$ to the adversary.

Synchronous Phase If `sync` = `true` do the following:

- On input $(\text{GetOutput}, \text{sid})$ from party P_i
  - If `FastOutput` = `true` and $\tau \geq \tau_{\texttt{tout}} + \tau_{\texttt{OD}}$, it outputs $(\text{Output}, y_i, \text{sid})$ to P_i.
  - If `FastOutput` = `false` and $\tau \geq \tau_{\texttt{tout}} + \tau_{\texttt{OND}}$, it outputs $(\text{Output}, y_i, \text{sid})$ to P_i.

Adversary

```
Upon each party corruption, update $(c, p, r) = \mathsf{Tamper}^{\textsc{Hyb}}_{t,T}$.
    // Core Set and Delivery of Outputs
1: Upon receiving a message $(\textsc{No-Input}, \mathcal{P}', \mathrm{sid})$ from the adversary, if $\texttt{sync} = \texttt{false}$, $\mathcal{P}'$ is a subset of $\mathcal{P}$ of size $|\mathcal{P}'| \leq t_r$ and $y_1 = \cdots = y_n = \bot$, set $\mathcal{I} = \mathcal{H} \setminus \mathcal{P}'$.
2: On input $(\textsc{DeliverOutput}, i, \mathrm{sid})$ from the adversary, if $y_i \neq \bot$ and $\texttt{sync} = \texttt{false}$, output $(\textsc{output}, y_i, \mathrm{sid})$ to $P_i$ and set $\texttt{FastOutput} = \texttt{true}$.
    // Adversary's capabilities
3: On input $(\textsc{TamperOutput}, P_i, y'_i, \mathrm{sid})$ from the adversary, if $c = 1$, set $y_i = y'_i$.
4: If $p = 1$, output $(x_1, \ldots, x_n)$ to the adversary.
5: On input $(\textsc{BlockAsynchOutput}, P_i, \mathrm{sid})$ from the adversary, if $r = 1$ and $\texttt{sync} = \texttt{false}$, set $w_i = \texttt{aBlocked}$.
```

In the version where $\mathcal{F}^{\mathtt{ua}}_{\textsc{hyb}}$ provides security with unanimous abort and no fairness, the adversary can in addition choose to set the output to $\bot$ for all honest parties and learn the output $y_{\mathtt{sync}}$, in the case $\texttt{FastOutput} = \texttt{false}$.

4 Compiler

In this section, we present a protocol which realizes the ideal functionality presented in the previous section. The protocol works with a setup $\mathcal{F}_{\textsc{Setup}}$, where parties have access to a public-key infrastructure used to sign values, and keys for a threshold encryption scheme.

The protocol uses a number of sub-protocols:

- $\Pi_{\mathtt{ZK}}$ is a bilateral zero-knowledge protocol which allows a party to prove knowledge of a witness corresponding to a statement.
- $\Pi_{\mathtt{aMPC}}$ is an asynchronous MPC protocol that provides full security up to t corruptions, and security without termination (correctness and privacy) up to $T \geq t$ corruptions.
- $\Pi^{\mathtt{fs}}_{\mathtt{sMPC}}$ (resp. $\Pi^{\mathtt{ua}}_{\mathtt{sMPC}}$) is a synchronous MPC protocol with full security (resp. security with unanimous abort) up to T corruptions.
- $\Pi_{\mathtt{sBC}}$ is a synchronous broadcast protocol secure up to T corruptions.

4.1 Key-Distribution Setup

The compiler works with a key distribution setup. The setup can be computed once for multiple instances of the protocol, without knowing the parties' inputs nor the function to evaluate.

As usual, we describe our compiler in a hybrid model where parties have access to an ideal functionality $\mathcal{F}_{\textsc{Setup}}$. At a very high level, $\mathcal{F}_{\textsc{Setup}}$ allows to distribute the keys for a threshold encryption scheme and a digital signature scheme. The threshold encryption scheme here does not need to be homomorphic. More concretely, it provides to each party P_i a global public key $\texttt{ek}$ and a private key share $\mathtt{dk}_i$. Moreover, it gives a PKI infrastructure. That is, it gives to each party P_i a signing key $\mathtt{sk}_i$ and the verification keys of all parties $(\mathtt{vk}_1, \ldots, \mathtt{vk}_n)$.

We describe the two setups, PKI setup $\mathcal{F}_{\mathrm{PKI}}$ and threshold encryption setup $\mathcal{F}_{\mathrm{TE}}$ independently. The setup of the protocol consists of includes both functionalities $\mathcal{F}_{\mathrm{SETUP}} = [\mathcal{F}_{\mathrm{PKI}}, \mathcal{F}_{\mathrm{TE}}]$.

Digital Signature Setup. The protocol assumes a signature setup. That is, each party P_i has a pair secret key and verification key $(\mathtt{sk}_i, \mathtt{vk}_i)$, where $\mathtt{vk}_i$ is known to all parties.

Threshold Encryption Setup. The protocol assumes also a threshold encryption setup, which allows each party to access a global public key $\mathtt{ek}$ and a private key share $\mathtt{dk}_i$.

4.2 Zero-Knowledge

The protocol $\Pi_{\mathtt{ZK}}$ is a bilateral zero-knowledge protocol which allows a party to prove knowledge of a witness corresponding to a statement. The protocol must be UC-secure, meaning that it has to UC-realize the $\mathcal{F}_{\mathrm{ZK}}$ functionality, described in the full version for completeness. As shown in [21], such a protocol exists in the $\mathcal{F}_{\mathrm{CRS}}$-hybrid model for any relation. For this protocol, we need *proofs of correct decryption*, where the relation is parametrized by a threshold encryption scheme. The statement consists of $\mathtt{ek}$, a ciphertext c, and a decryption share d. The witness is a decryption key share $\mathtt{dk}_i$ such that $d = \mathtt{Dec}_{\mathtt{dk}_i}(c)$.

4.3 Synchronous MPC

Classical synchronous MPC protocols [5, 13, 45, 27, 2, 20], for $\Pi^{\mathtt{fs}}_{\mathtt{sMPC}}$ can be proven to UC-realize an ideal MPC functionality $\mathcal{F}^{\mathtt{fs}}_{\mathrm{SYNC}}$ (described in the full version for completeness) up to $T < n/2$ corruptions, which allows a set of n parties to evaluate a specific function f. For the case of unanimous abort, where the adversary is allowed to set the output $\perp$, one can instantiate $\Pi^{\mathtt{ua}}_{\mathtt{sMPC}}$ for any $T < n$ [23, 29].

4.4 Synchronous Byzantine Broadcast

A Byzantine broadcast primitive allows a party P_s, called the sender, to consistently distribute a message among a set of parties $\mathcal{P}$.

Definition 3. *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. Π is a T-secure broadcast protocol if the following conditions hold up to T corruptions:*

- *Validity: If the sender P_s is honest, every honest party outputs the sender's message v.*
- *Consistency: All honest parties output the same message.*

The classical result of Dolev-Strong [22] shows that synchronous broadcast protocol $\Pi_{\texttt{sBC}}$ can be achieved for any $T < n$, assuming a public-key infrastructure. The protocol UC-realizes the synchronous broadcast functionality $\mathcal{F}_{\mathrm{sBC}}$ (which is a synchronous MPC functionality, where the output is the sender's input) for our setting with static corruptions [26, 36].

4.5 Asynchronous MPC

In this section we formally define what it means for a protocol $\Pi_{\texttt{aMPC}}$ to achieve full security up to t corruptions and security without termination (correctness and privacy) up to $T \geq t$ corruptions. In Section 5.2 we show how to achieve such a protocol.

In a nutshell, the idea is that the protocol realizes an ideal MPC functionality which is parametrized with the two thresholds (t, T). If the adversary corrupts up to t parties, all honest parties obtain all the security guarantees as a conventional asynchronous MPC functionality. If the adversary corrupts $t \leq f \leq T$ parties, it is allowed to block any party from obtaining output; however, those parties that obtain output, are ensured to obtain the correct output, and privacy is still guaranteed. Finally, if the adversary corrupts $f > T$ parties, no guarantees remain: the adversary learns the inputs from all honest parties and can choose the outputs to be anything.

To model formally an asynchronous MPC functionality, we borrow ideas from [36, 19]. In traditional asynchronous protocols, the parties are guaranteed to *eventually* receive output, meaning that the adversary can delay the output of honest parties in an arbitrary but finite manner. The reason for this is that the assumed network guarantees eventual delivery. One can make the simple observation that if the network has an unknown upper bound δ, then the adversary can delay the outputs of honest parties up to time $\tau_{\texttt{asynch}} = \tau(\delta)$, which is a function of δ. The guarantee obtained in an asynchronous MPC with eventual delivery (e.g. as in [19]) is a special case of our functionality, namely when $\tau_{\texttt{asynch}} = \infty$. We describe it for the case where $\tau_{\texttt{asynch}}$ is a fixed time, but one can model $\tau_{\texttt{asynch}}$ to be probabilistic as well.

It is known that asynchronous protocols cannot achieve simultaneously fast termination (at a time which depends on δ) and input completeness. This is because δ is unknown and hence it is impossible to distinguish between an honest slow party and an actively corrupted party. If fast termination must be ensured even when up to t parties are corrupted, the parties can only wait for $n - t$ inputs. Since the adversary is able to schedule the delivery of messages from honest parties, it can also typically choose exactly a set of parties $\mathcal{P}' \subseteq \mathcal{P}$, $|\mathcal{P}'| \leq t$, whose input is not considered. Therefore, the ideal functionality also allows the simulator to choose this set. As in [19], and similar to the network functionality $\mathcal{F}^{\delta}_{\mathrm{NET}}$, we use a "fetch-based" mode functionality and allow the simulator to specify a delay on the delivery to every party.

Functionality $\mathcal{F}_{\text{ASYNC}}$

$\mathcal{F}_{\text{ASYNC}}$ is connected to a global clock functionality $\mathcal{G}_{\text{CLK}}$. It is parameterized by a set $\mathcal{P}$ of n parties, a function f, a tamper function $\mathsf{Tamper}_{t,T}$, a delay δ, and a maximum delay $\tau_{\mathtt{asynch}}$. It initializes the variables $x_i = y_i = \bot$, $\tau_{\mathtt{in}} = \bot$ and $\tau_i = 0$ for each party $P_i \in \mathcal{P}$ and the variable $\mathcal{I} = \mathcal{H}$, where $\mathcal{H}$ is the set of honest parties.
Upon receiving input from any party or the adversary, it queries $\mathcal{F}_{\text{CLOCK}}$ for the current time and updates τ accordingly.

Party P_i:

```
1: On input (INPUT, $v_i$, sid) from party $P_i$:
     – If some party has received output, ignore this message. Otherwise, set $x_i = v_i$.
     – If $x_i \neq \bot$ for each $P_i \in \mathcal{I}$, set each output to $y_j = f(x'_1, \dots, x'_n)$, where $x'_i = x_i$ for each $P_i \in \mathcal{I} \cup (\mathcal{P} \setminus \mathcal{H})$ and $x'_i = \bot$ otherwise. Set $\tau_{\mathtt{in}} = \tau$.
     – Output (INPUT, $P_i$, sid) to the adversary.
2: On input (GETOUTPUT, sid) from $P_i$, if the output is not set or is blocked, i.e., $y_i \in \{\bot, \top\}$, ignore the message. Otherwise, if the current time is larger than the time set by the adversary, $\tau \geq \tau_i$, output (OUTPUT, $y_i$, sid) to $P_i$.
```

Adversary:

```
1: Upon receiving a message (NO-INPUT, $\mathcal{P}'$, sid) from the adversary, if $\mathcal{P}'$ is a subset of $\mathcal{P}$ of size $|\mathcal{P}'| \leq t$ and $y_1 = \dots = y_n = \bot$, set $\mathcal{I} = \mathcal{H} \setminus \mathcal{P}'$.
2: On input (SETOUTPUTTIME, $P_i$, $\tau'$, sid) from the adversary, if $\tau_{\mathtt{in}} \neq \bot$ and $\tau' < \tau_{\mathtt{in}} + \tau_{\mathtt{asynch}}$, set $\tau_i = \tau'$.
```

Upon each party corruption, update $(c, p, l) = \mathsf{Tamper}^{\text{ASYNCH}}_{t,T}$.

```
1: On input (TAMPEROUTPUT, $P_i$, $y'_i$, sid) from the adversary, if $c = 1$, set $y_i = y'_i$.
2: If $p = 1$, output $(x_1, \dots, x_n)$ to the adversary.
3: On input (BLOCKOUTPUT, $P_i$, sid) from the adversary, if $l = 1$, set $y_i = \top$.
```

Similar to $\mathcal{F}_{\text{HYB}}$, we parametrize the functionality by a tamper function to capture the guarantees depending on the set of corrupted parties. The $\mathcal{F}_{\text{ASYNC}}$ functionality has the tamper function $\mathsf{Tamper}^{\text{ASYNCH}}_{t,T}$, where the adversary can tamper with the output value and learn the inputs if the number of corruptions is larger than T, and is allowed to block the delivery of the outputs if the number of corruptions is larger than t.

Definition 4. *We define an asynchronous MPC functionality with full security t and security without termination T, if it has the tamper function* $\mathsf{Tamper}^{\text{ASYNCH}}_{t,T}$*:*

Function $\mathsf{Tamper}^{\text{ASYNCH}}_{t,T}$

$(c, p, l) = \mathsf{Tamper}^{\text{ASYNCH}}_{t,T}$, *where:*
- $c = 1$, $p = 1$ *if and only if* $|\mathcal{P} \setminus \mathcal{H}| > T$.
- $l = 1$ *if and only if* $|\mathcal{P} \setminus \mathcal{H}| > t$.

4.6 Protocol Compiler

The protocol has two phases: an asynchronous phase and a synchronous phase, separated by a pre-defined timeout. The timeout is set large enough (using Δ and the number of asynchronous rounds) so that the asynchronous phase should have supposedly terminated if there were not too many corruptions.

During the asynchronous phase, parties may obtain an output $y_{\mathtt{asynch}}$. We need to ensure (1) that if an honest party obtains an output $y_{\mathtt{asynch}}$ during the asynchronous phase, then every other honest party obtains this output as well; and (2) that the adversary does not learn two outputs. We remark that even if the function to evaluate is the same, the output obtained from the synchronous MPC protocol $\Pi_{\mathtt{sMPC}}$ is not necessarily $y_{\mathtt{asynch}}$. This is because in an asynchronous protocol $\Pi_{\mathtt{aMPC}}$, up to t inputs from honest parties can be ignored. This is the reason why we require that $\Pi_{\mathtt{aMPC}}$ evaluates the function $f' = \mathtt{Enc}_{\mathtt{ek}}(f)$. During the synchronous phase, parties agree on whether they execute the synchronous protocol $\Pi_{\mathtt{sMPC}}$. The parties will invoke $\Pi_{\mathtt{sMPC}}$ only if it is guaranteed that the adversary did not obtain $y_{\mathtt{asynch}}$. Also, if the parties do not invoke $\Pi_{\mathtt{sMPC}}$, it is guaranteed that they can jointly decrypt the output $y_{\mathtt{asynch}}$.

Asynchronous Phase. In this phase, parties optimistically execute $\Pi_{\mathtt{aMPC}}$. When a party P_i obtains as output a ciphertext $c = [y]$ from $\Pi_{\mathtt{aMPC}}$, it sends a signature of c and collects a list L of $n-t$ signatures on the same c. Once such list L is collected, it runs a robust threshold decryption protocol. For that, P_i computes a decryption share $d_i = \mathtt{Dec}_{\mathtt{dk}_i}(c)$, and proves using $\Pi_{\mathtt{ZK}}$ to each P_j that d_i is a correct decryption share of c. Upon receiving d_i and a correct proof of decryption share for c from $n-t$ parties, compute and output $y_i = \mathtt{Rec}(\{d_j\})$.

Synchronous Phase. After the timeout, parties execute a synchronous broadcast protocol to send a pair list-ciphertext (c, L), where L contains at least $n-t$ signatures on c, if such a list was collected during the asynchronous phase. If a party receives via broadcast any valid L, then it sends its decryption share d_i and runs the same robust threshold decryption protocol as above. Otherwise, parties execute the synchronous MPC $\Pi_{\mathtt{sMPC}}$.

Observe that if an honest party collects a list L of $n-t$ signatures on a ciphertext $[y]$ during the asynchronous phase, it broadcasts the pair $([y], L)$ during the synchronous phase. Then, every honest party obtains at least a valid pair $([y], L)$ after the broadcast round finishes. By a standard quorum argument, if there are up to $T < n-2t$ corruptions, there cannot be two signature lists of size $n-t$ on different values. Given that honest parties only sign the correct output ciphertext $[y_{\mathtt{asynch}}]$ from $\Pi_{\mathtt{aMPC}}$, this is the only value that can gather a list of signatures. Hence, all parties are instructed to run the robust threshold decryption protocol, and if there are up to t corruptions, every honest party is guaranteed to receive enough decryption shares to obtain the output $y_{\mathtt{asynch}}$. On the other hand, if no honest party obtained such a pair during the asynchronous phase, it is guaranteed that the adversary did not learn $y_{\mathtt{asynch}}$, since no honest party sent its decryption share. However, it might be that the adversary collected a valid $([y_{\mathtt{asynch}}], L')$. The adversary can then decide whether to broadcast a valid pair. If it does, every party will hold this pair and everyone outputs $y_{\texttt{asynch}}$ as before. And if it does not, no honest party holds a valid pair after the broadcast round, and every party can safely run the synchronous MPC protocol $\mathit{\Pi}_{\texttt{sMPC}}$.

We remark that it is not enough that upon the timeout parties simply *send* $([y], L)$, because the parties need to have agreement on whether or not to invoke $\mathit{\Pi}_{\texttt{sMPC}}$. It can happen that the adversary is the only one who collected $([y], L)$.

Protocol $\mathit{\Pi}^{\Delta}_{\texttt{hyb}}(P_i)$

The party stores the current time τ, a flag $\texttt{sync} = \texttt{false}$ and a variable $\tau_{\texttt{sync}} = \bot$. Let $\tau_{\texttt{tout}} = T_{\texttt{asynch}}(\Delta) + T_{\texttt{zk}}(\Delta) + \Delta$ be a known upper bound on the time to execute the asynchronous phase, composed of protocols $\mathit{\Pi}_{\texttt{aMPC}}$, $\mathit{\Pi}_{\texttt{ZK}}$ and a network transmission message. Also, let $T_{zk}(\Delta)$ denote an upper bound on the time to execute $\mathit{\Pi}_{\texttt{ZK}}$.

Clock / Timeout Each time the party is activated do the following:

```
1: Query $\mathcal{G}_{\text{CLK}}$ for the current time and updates $\tau$ accordingly.
2: If $\tau \geq \tau_{\texttt{tout}}$, set $\texttt{sync} = \texttt{true}$ and $\tau_{\texttt{sync}} = \tau$.
```

Setup:

```
1: If activated for the first time input (GETKEYS, sid) to $\mathcal{F}_{\text{SETUP}}$. We denote the
   public key $\texttt{ek}$, a $(n-t, n)$-share $\texttt{dk}_i$ of the corresponding secret key $\texttt{dk}$, the
   signing key $\texttt{sk}$ and the verification key $\texttt{vk}$.
```

Asynchronous Phase: If $\texttt{sync} = \texttt{false}$ handle the following commands.

```
– On input (INPUT, $x_i$, sid) (and following activations) do
    1: Execute $\mathit{\Pi}_{\texttt{aMPC}}$ with input $x_i$ and wait until an output $c$ is received.
    2: Send $(c, \mathsf{Sign}(c, \texttt{sk}))$ to every other party using $\mathcal{F}_{\text{NET}}$.
    3: Receive signatures and values via $\mathcal{F}_{\text{NET}}$ until you received $n-t$ signatures
       $L = (\sigma_1, \ldots, \sigma_l)$ on a value $c$.
    4: Send $(c, L)$ to every party using $\mathcal{F}_{\text{NET}}$.
    5: Receive message lists $(c, L')$. For each such list send $(c, L')$ to every party
       using $\mathcal{F}_{\text{NET}}$.
    6: Once done with the above, compute $d_i = \texttt{Dec}_{\texttt{dk}_i}(c)$, and prove, using $\mathcal{F}_{\text{ZK}}$,
       to each $P_j$, that $d_i$ is a correct decryption share of $c$.
    7: Upon receiving $n-t$ correct decryption shares for $c$, compute and output
       $y = \texttt{Rec}(\{d_j\})$.
– At every clock tick, if it is not possible to progress with the list above, send
  (CLOCKREADY) to $\mathcal{G}_{\text{CLK}}$.
```

Synchronous Phase: If $\texttt{sync} = \texttt{true}$ and $\tau \geq \tau_{\texttt{sync}}$, stop all previous steps and do the following commands.

```
– On input (CLOCKREADY) do:
    1: Send (CLOCKREADY) to $\mathcal{G}_{\text{CLK}}$.
    2: if $\tau \geq \tau_{\texttt{sync}}$ then
    3:     Use $\mathit{\Pi}_{\texttt{sBC}}$ to broadcast $(c, L)$, for each pair $(c, L)$ received during the
           Asynchronous Phase.
    4:     Wait until $\mathit{\Pi}_{\texttt{sBC}}$ terminated. If a pair $(c, L)$ was received as output,
           compute $d_i = \texttt{Dec}_{\texttt{dk}_i}(c)$, and prove, using $\mathcal{F}_{\text{ZK}}$, to each $P_j$, that $d_i$ is a
```

```
        correct decryption share of $c$. Otherwise, if no pair $(c, L)$ was received,
        run the synchronous MPC protocol $\Pi^{\mathtt{fs}}_{\mathtt{sMPC}}$ with input $x_i$.
    5: end if
```

– If there was an output (c', L') from $\Pi_{\mathtt{sBC}}$, wait for $T_{zk}(\Delta)$ clock ticks. After that, if $n - t$ correct decryption shares d_j are received from $\mathcal{F}_{\text{NET}}$, compute and reconstruct the value $y = \mathtt{Rec}(\{d_j\})$ from c, and output y. Otherwise, if there was no output (c, L') from $\Pi_{\mathtt{sBC}}$, output the output received from $\Pi^{\mathtt{fs}}_{\mathtt{sMPC}}$.

Let $T_{zk}(\delta)$, $T_{\mathtt{sync}}(\Delta)$, $T_{\mathtt{BC}}(\Delta)$, $T_{\mathtt{asynch}}(\delta)$ be the corresponding time to execute the protocols $\Pi_{\mathtt{ZK}}$, $\Pi_{\mathtt{sMPC}}$, $\Pi_{\mathtt{sBC}}$ and $\Pi_{\mathtt{aMPC}}$, respectively. We state the following theorem, and the proof is formally described in the full version. The communication complexity is inherited from the corresponding sub-protocols.

Theorem 1. *Assuming PKI and CRS, for any $\Delta \geq \delta$, $\Pi^{\Delta}_{\mathtt{hyb}}$ realizes $\mathcal{F}^{\mathtt{fs}}_{\text{HYB}}$ with full security with responsiveness t and full security $\min\{T, n-2t\}$. The maximum delay of the asynchronous phase is $\tau_{\mathtt{asynch}} = T_{\mathtt{asynch}}(\delta) + T_{\mathtt{zk}}(\delta) + \delta$, and of the synchronous phase is $\tau_{\mathtt{OD}} = T_{\mathtt{BC}}(\Delta) + T_{\mathtt{zk}}(\Delta)$ for a fast output with $n - t$ inputs, and otherwise is $\tau_{\mathtt{OND}} = T_{\mathtt{BC}}(\Delta) + T_{\mathtt{sync}}(\Delta)$ for an output with all the inputs.*

By replacing the invocation of $\Pi^{\mathtt{fs}}_{\mathtt{sMPC}}$ to $\Pi^{\mathtt{ua}}_{\mathtt{sMPC}}$, one realizes $\mathcal{F}^{\mathtt{ua}}_{\text{HYB}}$ for the same parameters. Let $\Pi^{\Delta}_{\mathtt{hyb\text{-}ua}}$ denote the same protocol as $\Pi^{\Delta}_{\mathtt{hyb}}$, except that the invocation of $\Pi^{\mathtt{fs}}_{\mathtt{sMPC}}$ is replaced by $\Pi^{\mathtt{ua}}_{\mathtt{sMPC}}$.

Theorem 2. *Assuming PKI and CRS, for any $\Delta \geq \delta$, $\Pi^{\Delta}_{\mathtt{hyb\text{-}ua}}$ realizes $\mathcal{F}^{\mathtt{ua}}_{\text{HYB}}$ with full security with responsiveness t and security with unanimous abort $\min\{T, n-2t\}$. The maximum delay of the asynchronous phase is $\tau_{\mathtt{asynch}} = T_{\mathtt{asynch}}(\delta) + T_{\mathtt{zk}}(\delta) + \delta$, and of the synchronous phase is $\tau_{\mathtt{OD}} = T_{\mathtt{BC}}(\Delta) + T_{\mathtt{zk}}(\Delta)$ for a fast output with $n-t$ inputs, and otherwise is $\tau_{\mathtt{OND}} = T_{\mathtt{BC}}(\Delta) + T_{\mathtt{sync}}(\Delta)$ for an output with all the inputs.*

5 Asynchronous Protocols

In this section, we show how to obtain $\Pi_{\mathtt{aMPC}}$ with full security with responsiveness up to t corruptions and security (correctness and privacy) up to T corruptions, for any $t < \frac{n}{3}$ and any $T < n - 2t$.

Technical Remark. In our model, parties have access to a synchronized clock. The asynchronous protocols do not read the clock, but in our model they need to specify at which point the parties send a (ClockReady) message to $\mathcal{G}_{\text{CLK}}$, so that the clock advances. Observe that we do not model time within a single asynchronous round (between fetching and sending messages), or computation time. Hence, in an asynchronous protocol, at every activation, each party P_i *fetches* the messages from the assumed functionalities, and then checks whether it has any message available that it can send. If so, it sends the corresponding message. Otherwise, it sends a (ClockReady) message to $\mathcal{G}_{\text{CLK}}$.

5.1 Asynchronous Byzantine Agreement

The goal of Byzantine agreement is to allow a set of parties to agree on a common value.

Definition 5. *Let Π be a protocol executed by parties $P_1, \dots, P_n$, 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.*

The first step is to obtain an asynchronous Byzantine Agreement protocol $\Pi_{\mathtt{aBA}}$ with higher consistency threshold. In the full version, we formally prove security of such a protocol $\Pi_{\mathtt{aBA}}$ in the UC framework for any validity t_v, consistency t_c and termination t_l, such that $t_l \le t_v < \frac{n}{3}$ and $t_c + 2t_l < n$.

The general idea is to trade termination by consistency, while keeping validity. The protocol is quite simple. First, each party P_i runs with input x_i a regular Byzantine agreement protocol secure up to a single threshold $t' = t_v < n/3$. Once an output x is obtained from the BA, it computes a signature $\sigma = \mathsf{Sign}(x, \mathtt{sk})$ and sends it to every other party. Once $n - t_l$ signatures on a value x' are collected, the party sends the list containing the signatures along with the value x' to every other party, and terminates with output x'. Since there cannot be two lists of $n - t_l$ signatures on different values if there are up to $t_c < n - 2t_l$ corruptions, this prevents parties to output different values if there are up to $t_c < n - 2t_l$ corruptions. On the other hand, termination is reduced to t_l. One can also verify that validity is inherited from the regular BA protocol: if every honest party starts with input x, no honest party signs any other value $x' \neq x$, and hence there cannot be a list of $n - t_l$ signatures on x', given that $t_l \le t_v$.

Lemma 1. *There is a Byzantine agreement protocol $\Pi_{\mathtt{aBA}}$ with validity, consistency and termination parameters (t_v, t_c, t_l), for any $t_l < \frac{n}{3}$, $t_l \le t_v < \frac{n}{3}$ and $t_c < n - 2t_l$, assuming a PKI infrastructure setup $\mathcal{F}_{\mathrm{PKI}}$. The expected maximum delay for the output is $\tau_{aba} = O(\delta)$.*

5.2 Two-Threshold Asynchronous MPC

In order to realize $\mathcal{F}_{\mathrm{ASYNC}}$ with full security up to t and security with no termination (correctness and privacy) up to T, where $t < \frac{n}{3}$ and $T + 2t < n$, we follow the ideas from [18, 32, 33], and replace the single-threshold asynchronous BA protocol for the one that we obtained in Section 5.1 with increased consistency $t_c < n - 2t_l$.

The protocol works with a threshold FHE setup, similar to [18], which we model with the functionality $\mathcal{F}^{\mathtt{FHE}}_{\textsc{Setup}}$, which is the same as $\mathcal{F}_{\textsc{Setup}}$ from Section 4.1, except that the threshold encryption scheme is fully-homomorphic. For completeness, we review the definition in the full version.

The protocol uses in addition a number of sub-protocols:

- $\Pi_{\mathtt{aBA}}$ is a Byzantine agreement protocol with liveness threshold $t_l = t < n/3$, validity $t \leq t_v < n/3$ and consistency $t_c = T < n - 2t$.
- $\Pi_{\mathtt{ZK}}$ is a bilateral zero-knowledge protocol, similar to the one in Section 4.

Very roughly, the protocol asks each party P_i to encrypt its input x_i and distribute it to all parties. Then, parties homomorphically evaluate the function over the encrypted inputs to obtain an encrypted output, and jointly decrypt the output. Of course, the protocol does not work like that. In order to achieve robustness, we need that every party proves in zero-knowledge the correctness of essentially every value provided during the protocol execution.

We are interested in zero-knowledge proofs for two relations, parametrized 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 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 $d = \mathtt{Dec}_{\mathtt{dk}_i}(c)$.

The protocol proceeds in three phases: the input stage, the computation and threshold-decryption stage, and the termination stage.

Input Stage. The goal of the input stage is to define an encrypted input for each party. In order to ensure that the inputs are independent, the parties are required to perform a proof of plaintext knowledge of their ciphertext. It is known that input completeness and guaranteed termination cannot be simultaneously achieved in asynchronous networks, since one cannot distinguish between an honest slow party and an actively corrupted party. Given that we only guarantee termination up to t corruptions, we can take into account $n - t$ input providers.

The input stage is as follows: each party P_i encrypts its input to obtain a ciphertext c_i. It then constructs a certificate π_i that P_i knows the plaintext of c_i and that c_i is the only input of P_i, using bilateral zero-knowledge proofs and signatures. It then sends (c_i, π_i) to every other party, and constructs a *certificate of distribution* $\mathtt{dist}_i$, which works as a non-interactive proof that (c_i, π_i) was distributed to at least $n - t$ parties. This certificate is sent to every party.

After P_i collects $n - t$ certificates of distribution, it knows that at least $n - t$ parties have proved knowledge of the plaintext of their input ciphertext and distributed the ciphertext correctly to $n - t$ parties. If the number of corruptions is smaller than $n - t$, this implies that each of the $n - t$ parties have proved knowledge of the plaintext of their input ciphertext and also have distributed the ciphertext to at least 1 honest party. At this point, if each party is instructed to echo the certified inputs they saw, then every honest party will end up holding the $n-t$ certified inputs. To determine who they are, the parties compute a common set of input providers. For that, n asynchronous Byzantine Agreement protocols are run, each one to decide whether a party's input will be taken into account. To ensure that the size of the common set is at least $n-t$, each party P_i inputs 1 to the BAs of those parties for which it saw a certified input. It then waits until there are $n-t$ ones from the BAs before inputting any 0.

Protocol $\Pi_{\mathtt{aMPC}}^{\mathtt{input}}(P_i)$

The protocol keeps sets S_i and D_i, initially empty. Let x_i be the input for P_i.

Setup:

```
1: If activated for the first time input (GetKeys, sid) to $\mathcal{F}_{\text{Setup}}^{\text{FHE}}$. We denote the
   public key $\mathtt{ek}$, a $(n-t,n)$-share $\mathtt{dk}_i$ of the corresponding secret key $\mathtt{dk}$, the
   signing key $\mathtt{sk}$ and the verification key $\mathtt{vk}$.
```

Plaintext Knowledge and Distribution:

```
1: Compute $c_i = \mathtt{Enc}_{\mathtt{ek}}(x_i)$.
2: Prove to each $P_j$ knowledge of the plaintext of $c_i$, using $\Pi_{\mathtt{ZK}}$.
3: Upon receiving a correct proof of plaintext knowledge for a ciphertext $c_j$ from
   $P_j$, send $\sigma_i^{\mathtt{popk}} = \mathsf{Sign}_{\mathtt{sk}_i}(c_j)$ to $P_j$.
4: Upon receiving $n-t$ signatures $\{\sigma_j^{\mathtt{popk}}\}$, compute $\pi_i = \{\sigma_j^{\mathtt{popk}}\}$ and send $(c_i, \pi_i)$
   to all parties.
5: Upon receiving a message $(c_j, \pi_j)$ from $P_j$, send $\sigma_i^{\mathtt{dist}} = \mathsf{Sign}_{\mathtt{sk}_i}((c_j, \pi_j))$ to
   $P_j$. Add $(j, (c_j, \pi_j))$ to $S_i$.
6: Upon receiving $n-t$ signatures $\{\sigma_j^{\mathtt{dist}}\}$, compute $\mathtt{dist}_i = \{\sigma_j^{\mathtt{dist}}\}$ and send
   $((c_i, \pi_i), \mathtt{dist}_i)$ to all parties.
7: Upon receiving $((c_j, \pi_j), \mathtt{dist}_j)$ from $P_j$, add $j$ to $D_i$.
```

Select Input Providers: Once $|D_i| > n-t$, stop the above rules and proceed as follows:

```
1: Send $S_i$ to every party.
2: Once $n-t$ sets $\{S_j\}$ are collected, let $R = \bigcup_j S_j$ and enter $n$ asynchronous
   Byzantine agreement protocols $\Pi_{\mathtt{aBA}}$ with inputs $v_1, \dots, v_n \in \{0,1\}$, where
   $v_j = 1$ if $\exists (j, (c_j, \pi_j)) \in R$. Keep adding possibly new received sets to $R$.
3: Wait until there are at least $n-t$ outputs which are one. Then, input 0 for
   the BAs which do not have input yet.
4: Let $w_1, \dots, w_n$ be the outputs of the BAs.
5: Let $\mathtt{CoreSet} := \{j | w_j = 1\}$.
6: For each $j \in \mathtt{CoreSet}$ with $(j, (c_j, \pi_j)) \in R$, send $(j, (c_j, \pi_j))$ to all parties.
   Wait until each tuple $(j, (c_j, \pi_j)), j \in \mathtt{CoreSet}$ is received.
```

Computation and Threshold-Decryption Stage. After input stage, parties have agreed on a common subset CoreSet of size at least $n-t$ parties, and each party holds the $n-t$ ciphertexts corresponding to the encryption of the input from each party in CoreSet. In the computation stage, the parties homomorphically evaluate the function, resulting on the ciphertext c encrypting the output. In the threshold-decryption stage, each party P_i computes the decryption share

$d_i = \mathtt{Dec}_{\mathtt{dk}_i}(c)$, and proves in zero-knowledge simultaneously towards all parties that the decryption share is correct. Once $n-t$ correct decryption shares on the same ciphertext are collected, P_i reconstructs the output y_i.

Protocol $\Pi^{\mathtt{comp}}_{\mathtt{aMPC}}(P_i)$

Start once $\Pi^{\mathtt{input}}_{\mathtt{aMPC}}(P_i)$ is completed. Let `CoreSet` be the resulting set of at least $n-t$ parties, and let the input ciphertexts be c_j, for each $j \in \mathtt{CoreSet}$.

Function Evaluation:

```
1: For each $j \notin \mathtt{CoreSet}$, assume a default valid ciphertext $c_j$ for $P_j$.
2: Locally compute the homomorphic evaluation of the function $c = f_{\mathtt{ek}}(c_1, \ldots, c_n)$.
```

Threshold Decryption:

```
1: Compute a decryption share $d_i = \mathtt{Dec}_{\mathtt{dk}_i}(c)$.
2: Prove, using $\Pi_{\mathtt{ZK}}$, to each $P_j$ that $d_i$ is a correct decryption share of $c$.
3: Upon receiving a correct proof of decryption share for a ciphertext $c'$ and decryption share $d_j$ from $P_j$, send $\sigma_i^{\mathtt{pocs}} = \mathsf{Sign}_{\mathtt{sk}_i}((d_j, c'))$ to $P_j$.
4: Upon receiving $n-t$ signatures $\{\sigma_j^{\mathtt{pocs}}\}$ on the same pair $(d_i, c')$, compute $\mathtt{ProofShare}_i = \{\sigma_j^{\mathtt{pocs}}\}$ and send $((d_i, c'), \mathtt{ProofShare}_i)$ to all parties.
5: Upon receiving $n-t$ valid pairs $((d_j, c'), \mathtt{ProofShare}_j)$ for the same $c'$, compute the output $y_i = \mathtt{Rec}(\{d_j\})$.
```

Termination Stage. The termination stage ensures that all honest parties terminate with the same output. This stage is essentially a Bracha broadcast [10] of the output value. The idea is that each party P_i votes for one output y_i and continuously collects outputs votes. More concretely, P_i sends y_i to every other party. If P_i receives $n-2t$ votes on the same value y, it knows that y is the correct output (because at least an honest party obtained the value y as output if the security threshold $T < n-2t$ is satisfied). Hence, if no output was computed yet, it sets $y_i = y$ as its output and sends y_i to every other party. Observe that if the security threshold is not satisfied, the adversary can tamper the outputs, but so can the simulator. Once $n-t$ votes on the same value y are collected, terminate with output y. If a party receives $n-t$ votes on y, and termination should be guaranteed ($f \leq t$), there are $n-2t$ honest parties that voted for y, and hence every honest party which did not output will at some point collect $n-2t$ votes on y, and hence will also vote for y. Since each honest party which terminated voted for y and each honest party which did not terminated voted for y as well, this means that all honest parties which did not terminate will receive $n-t$ votes for y.

Protocol $\Pi^{\mathtt{term}}_{\mathtt{aMPC}}(P_i)$

During the overall protocol, execute this protocol concurrently.

Waiting for Output:

```
1: Wait until the output $c$ is computed from $\Pi^{\mathtt{comp}}_{\mathtt{aMPC}}(P_i)$.
```

```
Adopt Output:
1: Wait until receiving $n - 2t$ votes for the same value $y$.
2: Adopt $y$ as output, and send $y$ to every other party.
Termination:
1: Wait until receiving $n - t$ votes for the same value $y$.
2: Terminate.
```

Let us denote $\Pi_{\texttt{aMPC}}$ the protocol that executes concurrently the protocols $\Pi^{\texttt{input}}_{\texttt{aMPC}}$, $\Pi^{\texttt{comp}}_{\texttt{aMPC}}$ and $\Pi^{\texttt{term}}_{\texttt{aMPC}}$. Each party, at every activation, tries to progress with any of the subprotocols. If they cannot, they output (ClockReady) to $\mathcal{G}_{\text{CLK}}$ so that the clock advances. In full version we prove the following theorem.

Theorem 3. *The protocol $\Pi_{\texttt{aMPC}}$ uses $\mathcal{F}^{\texttt{FHE}}_{\text{SETUP}}$ as setup and realizes $\mathcal{F}_{\text{ASYNC}}$ on any function f on the inputs, with full security up to t corruptions and security without termination up to T, for any $t < n/3$ and $T+2t < n$. The total maximum delay for the honest parties to obtain output is $\tau_{\texttt{asynch}} = \tau_{\texttt{aba}}(\delta) + 2\tau_{\texttt{zk}}(\delta) + 9\delta$.*

6 Impossibility Results

In this section we argue that the obtained trade-offs are optimal. We prove that any MPC protocol that achieves full security with responsiveness up to t corruptions, and extended security with unanimous abort up to T corruptions needs to satisfy $T + 2t < n$. Since full security is stronger than security with unanimous abort, these bounds also hold for the case where the extended security is full security.

Lemma 2. *Let t, T be such that $T + 2t \geq n$. There is no MPC protocol Π that achieves full security with responsiveness up to t corruptions, and extended security with unanimous abort up to $T \geq t$ corruptions.*

Proof. Let δ be the unknown delay upper bound. Moreover, let $\delta' \ll \delta$ be such that the time to execute Π when messages are scheduled within δ' is $\tau(\delta') < \delta$.

Assume without loss of generality that $3t = n$. We prove impossibility for the case where the function to be computed is the majority function. Consider three sets S_0, S_1 and S, where $|S_0| = |S_1| = t$ and $|S| = T$.

First, consider an execution where parties in S_0 and S are honest and have input 0, and parties in S_1 are corrupted and crash. Moreover, the adversary *instantly* delivers the messages between S_0 and S (within δ'). Since full security with responsiveness is guaranteed, parties in S_0 output 0 at time $\tau(\delta')$. Similarly, in an execution where parties in S_1 and S are honest and have input 1, the parties in S_1 output at time $\tau(\delta')$.

Now, consider an execution where S is corrupted, and the parties in S_0 and S_1 have inputs 0 and 1 respectively. The corrupted parties in S emulate an honest protocol execution with input $b \in \{0, 1\}$ with the parties in S_b. Moreover, the adversary delays δ the messages between S_0 and S_1. A party in S_0 (resp.

S_1) cannot distinguish between the two executions, because it outputs at time $\tau(\delta') < \delta$, and hence outputs 0 (resp. 1).

However, since T parties are corrupted, the protocol provides security with unanimous abort meaning that in the ideal world all honest parties output the same value (which may be $\perp$).

This contradicts the fact that Π achieves full security with responsiveness up to t corruptions and unanimous abort up to T corruptions. □

In addition, classical bounds in synchronous MPC with full security, show that full security for dishonest majority $T \geq n/2$ is impossible [17]. As a consequence, MPC with extended full security is impossible for dishonest majority.

7 Conclusions

We summarize all our results. Using the compiler from Section 4 and the following instantiations:

– A bilateral zero-knowledge protocol like in [21], which uses CRS.
– A synchronous MPC with full security (resp. unanimous abort) for $T < n/2$ (resp. $T < n$), using a protocol such as [5, 27] (resp. [23, 29]).
– A synchronous broadcast protocol for $T < n$ such as [22] from PKI.
– An asynchronous MPC with full security up to $t < n/3$ and security without termination up to $T < n - 2t$, as described in Section 5.2, based on PKI and threshold FHE (achievable from CRS [1]).

We obtain the following corollaries, where $T_{\texttt{sync}}(\varDelta)$ and $T_{\texttt{BC}}(\varDelta)$ are the running times for the synchronous MPC protocol and the synchronous broadcast:

Corollary 1. *There exists a protocol parametrized by $\varDelta \geq \delta$, which realizes $\mathcal{F}^{\texttt{fs}}_{\text{HYB}}$ on any function f, with full security with responsiveness t and full security T for any $t < \frac{n}{3}$ and $T < \min\{n/2, n - 2t\}$, in the $(\mathcal{G}_{\text{CLK}}, \mathcal{F}^{\delta}_{\text{NET}}, \mathcal{F}_{\text{PKI}}, \mathcal{F}_{\text{CRS}})$-hybrid world. The expected maximum delay of the asynchronous phase is $\tau_{\texttt{asynch}} = O(\delta)$, and the maximum delay of the synchronous phase is $\tau_{\texttt{OD}} = T_{\texttt{BC}}(\varDelta) + T_{\texttt{zk}}(\varDelta)$ if an output was delivered in the asynchronous phase, and otherwise is $\tau_{\texttt{OND}} = T_{\texttt{BC}}(\varDelta) + T_{\texttt{sync}}(\varDelta)$.*

For $t_r = \frac{n}{4}$, we obtain $\mathcal{F}^{\texttt{fs}}_{\text{HYB}}$ with correctness with privacy for any $t_s < \frac{n}{2}$.

Corollary 2. *There exists a protocol parametrized by $\varDelta \geq \delta$, which realizes $\mathcal{F}^{\texttt{ua}}_{\text{HYB}}$ on any function f, with full security with responsiveness t and full security T for any $t < \frac{n}{3}$ and $T < n - 2t$, in the $(\mathcal{G}_{\text{CLK}}, \mathcal{F}^{\delta}_{\text{NET}}, \mathcal{F}_{\text{PKI}}, \mathcal{F}_{\text{CRS}})$-hybrid world. The expected maximum delay of the asynchronous phase is $\tau_{\texttt{asynch}} = O(\delta)$, and the maximum delay of the synchronous phase is $\tau_{\texttt{OD}} = T_{\texttt{BC}}(\varDelta) + T_{\texttt{zk}}(\varDelta)$ if an output was delivered in the asynchronous phase, and otherwise is $\tau_{\texttt{OND}} = T_{\texttt{BC}}(\varDelta) + T_{\texttt{sync}}(\varDelta)$.*

References

[1] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 483–501. Springer, Heidelberg, April 2012.

[2] 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.

[3] Zuzana Beerliova-Trubiniova, Martin Hirt, and Jesper Buus Nielsen. Almost-asynchronous MPC with faulty minority. Cryptology ePrint Archive, Report 2008/416, 2008. `http://eprint.iacr.org/2008/416`.

[4] Zuzana Beerliová-Trubíniová, Martin Hirt, and Jesper Buus Nielsen. On the theoretical gap between synchronous and asynchronous MPC protocols. In *PODC, Zurich, Switzerland*, 2010.

[5] 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.

[6] 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.

[7] Erica Blum, John Katz, and Julian Loss. Synchronous consensus with optimal asynchronous fallback guarantees. In *Theory of Cryptography Conference*, 2019.

[8] 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`.

[9] Erica Blum, Chen-Da Liu-Zhang, and Julian Loss. Always have a backup plan: Fully secure synchronous mpc with asynchronous fallback. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020*, pages 707–731, Cham, 2020. Springer International Publishing.

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

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

[12] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 61–85. Springer, Heidelberg, February 2007.

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

[14] Ashish Choudhury. Optimally-resilient unconditionally-secure asynchronous multi-party computation revisited. Cryptology ePrint Archive, Report 2020/906, 2020. `https://eprint.iacr.org/2020/906`.

[15] Ashish Choudhury and Arpita Patra. Optimally resilient asynchronous mpc with linear communication complexity. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking*, pages 1–10, 2015.

[16] Ashish Choudhury, Arpita Patra, and Divya Ravi. Round and communication efficient unconditionally-secure MPC with t<n / 3 in partially synchronous network. In *ICITS 2017*, 2017.

[17] 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.

[18] 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.

[19] Sandro Coretti, Juan A. Garay, Martin Hirt, and Vassilis Zikas. Constant-round asynchronous multi-party 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, 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.

[21] Alfredo De Santis, Giovanni Di Crescenzo, Rafail Ostrovsky, Giuseppe Persiano, and Amit Sahai. Robust non-interactive zero knowledge. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 566–598. Springer, Heidelberg, August 2001.

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

[23] 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.

[24] Matthias Fitzi, Martin Hirt, Thomas Holenstein, and Jürg Wullschleger. Two-threshold broadcast and detectable multi-party computation. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 51–67. Springer, Heidelberg, May 2003.

[25] Matthias Fitzi, Thomas Holenstein, and Jürg Wullschleger. Multi-party computation with hybrid security. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 419–438. Springer, Heidelberg, May 2004.

[26] Juan A Garay, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. Adaptively secure broadcast, revisited. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 179–186, 2011.

[27] 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.

[28] Oded Goldreich and Erez Petrank. The best of both worlds: Guaranteeing termination in fast randomized byzantine agreement protocols. Technical report, Computer Science Department, Technion, 1990.

[29] Shafi Goldwasser and Yehuda Lindell. Secure computation without a broadcast channel. In *16th International Symposium on Distributed Computing (DISC)*. Citeseer, 2002.

[30] Yue Guo, Rafael Pass, and Elaine Shi. Synchronous, with a chance of partition tolerance. In *Annual International Cryptology Conference*, pages 499–529. Springer, 2019.

[31] Martin Hirt, Christoph Lucas, and Ueli Maurer. A dynamic tradeoff between active and passive corruptions in secure multi-party computation. In Ran Canetti

and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 203–219. Springer, Heidelberg, August 2013.

[32] 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.

[33] Martin Hirt, Jesper Buus Nielsen, and Bartosz Przydatek. Asynchronous multi-party computation with quadratic communication. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 473–485. Springer, Heidelberg, July 2008.

[34] Yuval Ishai, Eyal Kushilevitz, Yehuda Lindell, and Erez Petrank. On combining privacy with guaranteed output delivery in secure multiparty computation. In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 483–500. Springer, Heidelberg, August 2006.

[35] Jonathan Katz. On achieving the "best of both worlds" in secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *39th ACM STOC*, pages 11–20. ACM Press, June 2007.

[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] Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. Fair and robust multi-party computation using a global transaction ledger. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 705–734. Springer, Heidelberg, May 2016.

[38] Klaus Kursawe. Optimistic asynchronous byzantine agreement. 2000.

[39] Eyal Kushilevitz, Yehuda Lindell, and Tal Rabin. Information-theoretically secure protocols and security under composition. In Jon M. Kleinberg, editor, *38th ACM STOC*, pages 109–118. ACM Press, May 2006.

[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] 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.

[42] Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 3–33. Springer, 2018.

[43] Arpita Patra, Ashish Choudhary, and C Pandu Rangan. Communication efficient statistical asynchronous multiparty computation with optimal resilience. In *International Conference on Information Security and Cryptology*, pages 179–197. Springer, 2009.

[44] Arpita Patra and Divya Ravi. On the power of hybrid networks in multi-party computation. *IEEE Trans. Information Theory*, 2018.

[45] 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.