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 $\varDelta$, 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* [6, 15, 50, 32, 2, 24], and of arbitrary number of corruptions for the case of *security with (unanimous) abort* and no fairness [27, 34]. On the downside, especially in real-world networks where the *actual* maximal network delay δ is hard to predict, $\varDelta$ 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 $\varDelta$; a notion that we shall refer to as *responsiveness* [46]. 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 [7]. 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 [12], 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 $T + 2t < n$.

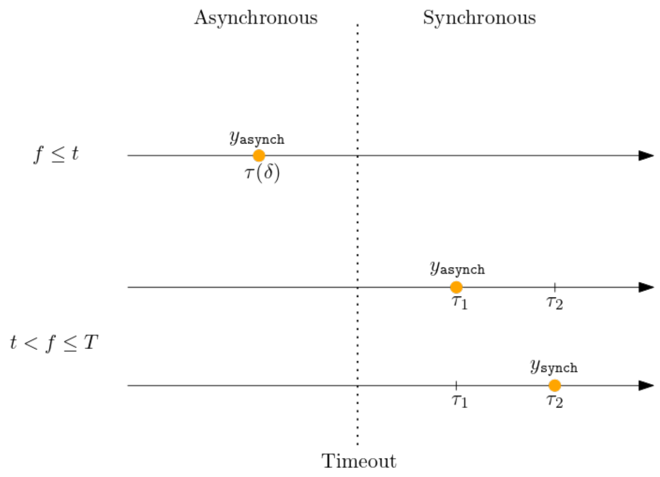

Fig. 1. The dotted vertical line separates the asynchronous and the synchronous phase. The orange dot shows the latest point in time when honest parties get output. The output $y_{\mathtt{asynch}}$ takes into account $n - t$ inputs, whereas $y_{\mathtt{sync}}$ takes into account all inputs. Up to t corruptions 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 $\perp$ for $\mathcal{F}^{\mathtt{ua}}_{\mathrm{HYB}}$.

For the first sub-protocol 1), we show how to modify the asynchronous MPC protocol by Cohen [20] 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. [27, 34]) and for $T < n/2$ for full security (e.g. [6, 15, 50, 32, 2, 24]).

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. [48, 17, 16, 20, 37]) can be adapted to the two-threshold setting, we choose to adapt the protocol in [20] for simplicity.

The protocol in [20] 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 Δ clock ticks, where Δ is a known upper bound on the network delay. In this case, the output is obtained at a time which depends on Δ. Note that Δ 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 Δ 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 $\Delta \gg \delta$. Hence, any synchronous protocol (even constant-round) is slow.

Notification Based. A well-known approach (see e.g. [44]) 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 Δ 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 Δ 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 trade-offs 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 [46] 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* [47]. 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 [45] 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. [35] 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 [33] 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 [43] 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 [8, 9, 10] 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, 18, 49] 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 [28, 39, 40, 29, 36] consider different thresholds to achieve more fine-grained security guarantees.

Worth mentioning, are the works of [39, 40], 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 [40], 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 [41, 42]. Parties have access to functionalities and global functionalities [13]. 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 $\texttt{buffer}$ 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 (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 $\texttt{buffer}$. Then, it sends the tuple $(\text{SENT}, P_i, P_j, m, \texttt{id}_m)$ to the adversary.
3: On input $(\text{FETCHMESSAGES}, i)$ from P_i, for each message tuple $(T_{\texttt{init}}, T_{\texttt{end}}, P_k, P_i, m, \texttt{id}_m)$ from $\texttt{buffer}$ where $T_{\texttt{end}} \leq \tau$, the functionality removes the tuple from $\texttt{buffer}$ and outputs (k, m) to P_i.
4: On input $(\text{DELAY}, D, \texttt{id})$ from the adversary, if there exists a tuple $(T_{\texttt{init}}, T_{\texttt{end}}, P_i, P_j, m, \texttt{id})$ in $\texttt{buffer}$ and $T_{\texttt{end}} + D \leq T_{\texttt{init}} + \delta$, then set $T_{\texttt{end}} = T_{\texttt{end}} + D$ and return (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 Δ. 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* $\mathsf{Tamper}_{t,T}^{\textsc{Hyb}}$*:*

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

// Flags indicating violation of c correctness, p privacy, r responsiveness
$(c, p, r) = \mathsf{Tamper}_{t,T}^{\textsc{Hyb}}$*, 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 Δ.

In addition, it keeps the following local variables:

– $\texttt{FastOutput}$ 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.
– $\texttt{sync}$ 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 $\texttt{FastOutput}$, τ, $\texttt{sync}$, x_i, y_i, w_i. These variables are initialized as $\texttt{FastOutput} = \texttt{false}$, $\tau = 0$, $\texttt{sync} = \texttt{false}$, $x_i = \perp$, 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 $\texttt{sync} = \texttt{true}$. If $\texttt{FastOutput} = \texttt{false}$, compute $y_1 = \cdots = y_n = f(x_1, \ldots, x_n)$.

Asynchronous Phase If $\texttt{sync} = \texttt{false}$ do the following:

– At the onset of the execution, output δ and $\tau_{\texttt{asynch}}$ to the adversary.
– On input $(\textsc{Input}, v_i, \text{sid})$ 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 $(\textsc{Input}, P_i, \text{sid})$ to the adversary.
– On input $(\textsc{GetOutput}, \text{sid})$ from P_i do the following:
  • If the output has not been set yet or is blocked, i.e., $y_i = \perp$ or $w_i = \texttt{aBlocked}$, ignore this message.
  • If $\tau \geq \tau_{\texttt{asynch}}$ output $(\textsc{output}, y_i, \text{sid})$ to P_i and set $\texttt{FastOutput} = \texttt{true}$.
  • Otherwise, output $(\textsc{output}, P_i, \text{sid})$ to the adversary.

Synchronous Phase If $\texttt{sync} = \texttt{true}$ do the following:

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

Adversary

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

```
    // 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 $\mathcal{F}^{\texttt{ua}}_{\text{HYB}}$ 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 $y_{\texttt{sync}}$, 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 $\mathcal{F}_{\text{Setup}}$, 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:

- $\Pi_{\texttt{ZK}}$ is a bilateral zero-knowledge protocol which allows a party to prove knowledge of a witness corresponding to a statement.
- $\Pi_{\texttt{aMPC}}$ 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.
- $\Pi^{\texttt{fs}}_{\texttt{sMPC}}$ (resp. $\Pi^{\texttt{ua}}_{\texttt{sMPC}}$) is a synchronous MPC protocol with full security (resp. security with unanimous abort) up to T corruptions.
- $\Pi_{\texttt{sBC}}$ 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 $\mathcal{F}_{\text{Setup}}$. At a very high level, $\mathcal{F}_{\text{Setup}}$ 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 $\mathtt{ek}$ 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 $\mathcal{F}_{\textsc{Setup}} = [\mathcal{F}_{\mathrm{PKI}}, \mathcal{F}_{\mathrm{TE}}]$.

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 Section A for completeness). As shown in [25], 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 [6, 15, 50, 32, 2, 24], for $\Pi^{\mathtt{fs}}_{\mathtt{sMPC}}$ can be proven to UC-realize an ideal MPC functionality $\mathcal{F}^{\mathtt{fs}}_{\mathrm{SYNC}}$ (described in Section A 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 $T < n$ [27, 34].

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 [26] 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 $\mathcal{F}_{\text{sBC}}$ (which is a synchronous MPC functionality, where the output is the sender's input) for our setting with static corruptions [30, 41].

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 [41, 23]. 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 [23]) 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 [23], and similar to the network functionality $\mathcal{F}^{\delta}_{\text{NET}}$, 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 $\tau_{\texttt{asynch}}$. It initializes the variables $x_i = y_i = \bot$, $\tau_{\texttt{in}} = \bot$ 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 $(\text{INPUT}, v_i, \text{sid})$ from party P_i:
- If some party has received output, ignore this message. Otherwise, set $x_i = v_i$.
- If $x_i \neq \bot$ 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 $\tau_{\texttt{in}} = \tau$.
- Output $(\text{INPUT}, P_i, \text{sid})$ to the adversary.

2: On input $(\text{GETOUTPUT}, \text{sid})$ 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 $(\text{OUTPUT}, y_i, \text{sid})$ to P_i.

Adversary:

1: Upon receiving a message $(\text{NO-INPUT}, \mathcal{P}', \text{sid})$ 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 $(\text{SETOUTPUTTIME}, P_i, \tau', \text{sid})$ from the adversary, if $\tau_{\texttt{in}} \neq \bot$ and $\tau' < \tau_{\texttt{in}} + \tau_{\texttt{asynch}}$, set $\tau_i = \tau'$.

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

1: On input $(\text{TAMPEROUTPUT}, P_i, y'_i, \text{sid})$ 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 $(\text{BLOCKOUTPUT}, P_i, \text{sid})$ 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 $y_{\texttt{asynch}}$. We need to ensure (1) that if an honest party obtains an output $y_{\texttt{asynch}}$ 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 $\Pi_{\texttt{sMPC}}$ is not necessarily $y_{\texttt{asynch}}$. This is because in an asynchronous protocol $\Pi_{\texttt{aMPC}}$, up to t inputs from honest parties can be ignored. This is the reason why we require that $\Pi_{\texttt{aMPC}}$ evaluates the function $f' = \texttt{Enc}_{\texttt{ek}}(f)$. During the synchronous phase, parties agree on whether they execute the synchronous protocol $\Pi_{\texttt{sMPC}}$. The parties will invoke $\Pi_{\texttt{sMPC}}$ only if it is guaranteed that the adversary did not obtain $y_{\texttt{asynch}}$. Also, if the parties do not invoke $\Pi_{\texttt{sMPC}}$, it is guaranteed that they can jointly decrypt the output $y_{\texttt{asynch}}$.

Asynchronous Phase. In this phase, parties optimistically execute $\Pi_{\texttt{aMPC}}$. When a party P_i obtains as output a ciphertext $c = [y]$ from $\Pi_{\texttt{aMPC}}$, 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 $d_i = \texttt{Dec}_{\texttt{dk}_i}(c)$, and proves using $\Pi_{\texttt{ZK}}$ 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 $y_i = \texttt{Rec}(\{d_j\})$.

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 $\Pi_{\texttt{sMPC}}$.

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 $[y_{\texttt{asynch}}]$ from $\Pi_{\texttt{aMPC}}$, 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 $y_{\texttt{asynch}}$. 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 $y_{\texttt{asynch}}$, since no honest party sent its decryption share. However, it might be that the adversary collected a valid $([y_{\texttt{asynch}}], L')$. 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 $\Pi_{\texttt{sMPC}}$.

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 $\Pi_{\texttt{sMPC}}$. It can happen that the adversary is the only one who collected $([y], L)$.

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

The party stores the current time τ, a flag $\texttt{sync} = \texttt{false}$ and a variable $\tau_{\texttt{sync}} = \perp$. 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 $\Pi_{\texttt{aMPC}}$, $\Pi_{\texttt{ZK}}$ and a network transmission message. Also, let $T_{zk}(\Delta)$ denote an upper bound on the time to execute $\Pi_{\texttt{ZK}}$.

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 $\Pi_{\texttt{aMPC}}$ 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 $\Pi_{\texttt{sBC}}$ to broadcast $(c, L)$, for each pair $(c, L)$ received during the
           Asynchronous Phase.
    4:     Wait until $\Pi_{\texttt{sBC}}$ 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 Section B. 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 $P_1, \ldots, P_n$, 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.*

The first step is to obtain an asynchronous Byzantine Agreement protocol $\Pi_{\mathtt{aBA}}$ with higher consistency threshold. In Section C, 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 $t_l \leq t_v < \frac{n}{3}$ 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 $t_l \leq t_v$.

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 $t_l < \frac{n}{3}$, $t_l \leq t_v < \frac{n}{3}$ 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 [20, 37, 38], 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 [20], which we model with the functionality $\mathcal{F}_{\text{SETUP}}^{\texttt{FHE}}$, which is the same as $\mathcal{F}_{\text{SETUP}}$ from Section 4.1, except that the threshold encryption scheme is fully-homomorphic. For completeness, we review the definition of a FHE scheme in Section E.

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

– $\Pi_{\texttt{aBA}}$ is a Byzantine agreement protocol with liveness threshold $t_l = t < n/3$, validity $t \leq t_v < n/3$ and consistency $t_c = T < n - 2t$.
– $\Pi_{\texttt{ZK}}$ 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 $\texttt{ek}$:

1. *Proof of Plaintext Knowledge:* The statement consists of $\texttt{ek}$, and a ciphertext c. The witness consists of a plaintext m and randomness r such that $c = \texttt{Enc}_{\texttt{ek}}(m; r)$.
2. *Proof of Correct Decryption:* The statement consists of $\texttt{ek}$, a ciphertext c, and a decryption share d. The witness consists of a decryption key share $\texttt{dk}_i$, such that $d = \texttt{Dec}_{\texttt{dk}_i}(c)$.

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* $\texttt{dist}_i$, 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 $\Pi_{\texttt{aMPC}}^{\texttt{input}}(P_i)$

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 $(\textsc{GetKeys}, \text{sid})$ to $\mathcal{F}_{\textsc{Setup}}^{\texttt{FHE}}$. 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 $\texttt{vk}$.
```

Plaintext Knowledge and Distribution:

```
1: Compute $c_i = \texttt{Enc}_{\texttt{ek}}(x_i)$.
2: Prove to each $P_j$ knowledge of the plaintext of $c_i$, using $\Pi_{\texttt{ZK}}$.
3: Upon receiving a correct proof of plaintext knowledge for a ciphertext $c_j$ from
   $P_j$, send $\sigma_i^{\texttt{popk}} = \mathsf{Sign}_{\texttt{sk}_i}(c_j)$ to $P_j$.
4: Upon receiving $n-t$ signatures $\{\sigma_j^{\texttt{popk}}\}$, compute $\pi_i = \{\sigma_j^{\texttt{popk}}\}$ and send $(c_i, \pi_i)$
   to all parties.
5: Upon receiving a message $(c_j, \pi_j)$ from $P_j$, send $\sigma_i^{\texttt{dist}} = \mathsf{Sign}_{\texttt{sk}_i}((c_j, \pi_j))$ to
   $P_j$. Add $(j, (c_j, \pi_j))$ to $S_i$.
6: Upon receiving $n-t$ signatures $\{\sigma_j^{\texttt{dist}}\}$, compute $\texttt{dist}_i = \{\sigma_j^{\texttt{dist}}\}$ and send
   $((c_i, \pi_i), \texttt{dist}_i)$ to all parties.
7: Upon receiving $((c_j, \pi_j), \texttt{dist}_j)$ 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 $\Pi_{\texttt{aBA}}$ 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 $\texttt{CoreSet} \coloneqq \{j | w_j = 1\}$.
6: For each $j \in \texttt{CoreSet}$ with $(j, (c_j, \pi_j)) \in R$, send $(j, (c_j, \pi_j))$ to all parties.
   Wait until each tuple $(j, (c_j, \pi_j)), j \in \texttt{CoreSet}$ is received.
```

Computation and Threshold-Decryption Stage. After input stage, parties have agreed on a common subset $\texttt{CoreSet}$ 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 $\texttt{CoreSet}$. 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 $\mathtt{CoreSet}$ 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 $\sigma^{\mathtt{pocs}}_i = \mathsf{Sign}_{\mathtt{sk}_i}((d_j, c'))$ to $P_j$.
4: Upon receiving $n-t$ signatures $\{\sigma^{\mathtt{pocs}}_j\}$ on the same pair $(d_i, c')$, compute $\mathtt{ProofShare}_i = \{\sigma^{\mathtt{pocs}}_j\}$ 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 [11] 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 Section D, 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 $\bot$).

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

In addition, classical bounds in synchronous MPC with full security, show that full security for dishonest majority $T \geq n/2$ is impossible [19]. 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 [25], which uses CRS.
- A synchronous MPC with full security (resp. unanimous abort) for $T < n/2$ (resp. $T < n$), using a protocol such as [6, 32] (resp. [27, 34]).
- A synchronous broadcast protocol for $T < n$ such as [26] 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 $T_{\mathtt{sync}}(\Delta)$ and $T_{\mathtt{BC}}(\Delta)$ are the running times for the synchronous MPC protocol and the synchronous broadcast:

Corollary 1. *There exists a protocol parametrized by $\Delta \geq \delta$, which realizes $\mathcal{F}_{\text{HYB}}^{\mathtt{fs}}$ 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 $(\mathcal{G}_{\text{CLK}}, \mathcal{F}_{\text{NET}}^{\delta}, \mathcal{F}_{\text{PKI}}, \mathcal{F}_{\text{CRS}})$-hybrid world. The expected maximum delay of the asynchronous phase is $\tau_{\mathtt{asynch}} = O(\delta)$, and the maximum delay of the synchronous phase is $\tau_{\mathtt{OD}} = T_{\mathtt{BC}}(\Delta) + T_{\mathtt{zk}}(\Delta)$ if an output was delivered in the asynchronous phase, and otherwise is $\tau_{\mathtt{OND}} = T_{\mathtt{BC}}(\Delta) + T_{\mathtt{sync}}(\Delta)$.*

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

Corollary 2. *There exists a protocol parametrized by $\Delta \geq \delta$, which realizes $\mathcal{F}_{\text{HYB}}^{\mathtt{ua}}$ 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 $(\mathcal{G}_{\text{CLK}}, \mathcal{F}_{\text{NET}}^{\delta}, \mathcal{F}_{\text{PKI}}, \mathcal{F}_{\text{CRS}})$-hybrid world. The expected maximum delay of the asynchronous phase is $\tau_{\mathtt{asynch}} = O(\delta)$, and the maximum delay of the synchronous phase is $\tau_{\mathtt{OD}} = T_{\mathtt{BC}}(\Delta) + T_{\mathtt{zk}}(\Delta)$ if an output was delivered in the asynchronous phase, and otherwise is $\tau_{\mathtt{OND}} = T_{\mathtt{BC}}(\Delta) + T_{\mathtt{sync}}(\Delta)$.*

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 and Ran El-Yaniv. Resilient-optimal interactive consistency in constant time. *Distributed Computing*, 16(4):249–262, 2003.

[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, John Katz, and Julian Loss. Synchronous consensus with optimal asynchronous fallback guarantees. In *Theory of Cryptography Conference*, 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] Erica Blum, Chen-Da Liu-Zhang, and Julian Loss. Always have a backup plan: Fully secure synchronous mpc with asynchronous fallback. In *CRYPTO, to appear*, 2020.

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

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

[13] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] Ran Cohen, Sandro Coretti, Juan Garay, and Vassilis Zikas. Round-preserving parallel composition of probabilistic-termination cryptographic protocols. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 80. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[22] Ran Cohen, Sandro Coretti, Juan A. Garay, and Vassilis Zikas. Probabilistic termination and composability of cryptographic protocols. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part III*, volume 9816 of *LNCS*, pages 240–269. Springer, Heidelberg, August 2016.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] Craig Gentry. *A fully homomorphic encryption scheme.* PHD. Thesis, 2009.

[32] 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.

[33] 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.

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

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

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

[44] 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.

[45] 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`.

[46] 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.

[47] 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.

[48] 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.

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

[50] 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.

Supplementary Material

The following supplementary material is divided in sections labeled by latin letters and appropriately referred to in the body.

A UC Zero-Knowledge and Synchronous MPC

For completeness, we formally describe the UC functionalities for zero-knowledge [14] and synchronous MPC [41].

A.1 Zero-Knowledge

We formally describe the UC $\mathcal{F}_{\text{ZK}}$ functionality, which allows a prover to prove knowledge of a certain witness w for a statement x satisfying a relation R.

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

$\mathcal{F}_{\text{ZK}}$ is connected to a global clock functionality $\mathcal{G}_{\text{CLK}}$. It is parameterized by a prover P, verifier V, a relation R, and a delay time $\tau_{\texttt{zk}}$. It also stores the current time τ and keeps a buffer `buffer` of messages containing the proofs that is initially empty.
Each time the functionality is activated, it first queries $\mathcal{G}_{\text{CLK}}$ for the current time and updates τ accordingly.

Zero-Knowledge Proof:

1: On input (x, w) from P, if $R(x, w) = 1$, create a new identifier sid and record the tuple $(\tau, \tau + 1, (x, w), \text{sid})$.Then sends (x, sid) to the adversary.
2: On input $(\text{GetProof}, \text{sid})$ from V, for each tuple $(T_{\texttt{init}}, T_{\texttt{end}}, (x, w), \text{sid})$ such that $T_{\texttt{end}} \leq \tau$, remove it from `buffer` and output (x, sid) to V.
3: On input $(\text{Delay}, T, \text{sid})$ from the adversary, if there is a tuple $(T_{\texttt{init}}, T_{\texttt{end}}, (x, w), \text{sid})$ in `buffer` and $T_{\texttt{end}} + T \leq T_{\texttt{init}} + \tau_{\texttt{zk}}$, then set $T_{\texttt{end}} = T_{\texttt{end}} + T$ and return (Delay-OK) to the adversary. Otherwise, ignore the message.

A.2 Synchronous MPC

We describe the ideal functionality $\mathcal{F}^{\texttt{fs}}_{\text{SYNC}}$ for full security and $\mathcal{F}^{\texttt{ua}}_{\text{SYNC}}$. The functionality is connected to a global clock $\mathcal{G}_{\text{CLK}}$ and is parametrized by the delay time $\tau_{\texttt{sync}}$ for which the honest parties obtain the output. For simplicity, we model the synchronous functionality with deterministic termination, but one can extend this to probabilistic termination using the frameworks presented in [22, 21].

Tamper Function for Synchronous SFE. The tamper function $\mathsf{Tamper}^{\text{Synch}}_{T}$ models the adversary's capabilities for a SFE functionality secure up to a single threshold T. The adversary can tamper with the output and learn the inputs from honest parties if and only if the number of corruptions is larger than T.

Definition 6. *We define a synchronous SFE functionality secure up to T corruptions if it has the following tamper function* $\mathsf{Tamper}_T^{\textsc{Synch}}$*:*

Function $\mathsf{Tamper}_T^{\textsc{Synch}}$

$(c, p) = \mathsf{Tamper}_T^{\textsc{Synch}}$*, where:*
- $c = 1$*,* $p = 1$ *if and only if* $|\mathcal{P} \setminus \mathcal{H}| > T$*.*

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

$\mathcal{F}_{\text{SYNC}}$ is connected to a global clock $\mathcal{G}_{\text{CLK}}$. $\mathcal{F}_{\text{SYNC}}$ is parameterized by a set $\mathcal{P}$ of n parties, a function f and a tamper function $\mathsf{Tamper}_T^{\textsc{Synch}}$, and a delay time at which the parties obtain output $\tau_{\texttt{sync}}$. Additionally, it initializes $\tau = 0$ and, for each party P_i, $x_i = y_i = \bot$. It keeps the set of honest parties $\mathcal{H}$.
Upon receiving input from any party or the adversary, it queries $\mathcal{F}_{\text{CLOCK}}$ for the current time and updates τ accordingly.

Party:

1: On input $(\textsc{Input}, v_i, \text{sid})$ from each party $P_i \in \mathcal{H}$ at a fixed time τ':
   - If $x_i = \bot$, it sets $x_i = v_i$.
   - Set $\tau_{\texttt{out}} = \tau' + \tau_{\texttt{sync}}$.

2: If for each party $P_i \in \mathcal{H}$ $x_i \neq \bot$, set each $y_i = f(x_1, \dots, x_n)$.

3: On input $(\textsc{GetOutput}, \text{sid})$ from honest party P_i or the adversary (for corrupted P_i), if $\tau \geq \tau_{\texttt{out}}$, it outputs $(\textsc{Output}, y_i, \text{sid})$ to P_i.

Adversary: Upon party corruption, set $(c, p) = \mathsf{Tamper}_T^{\textsc{Synch}}((x_1, \dots, x_n), \mathcal{H})$.

1: On input $(\textsc{TamperOutput}, P_i, y_i', \text{sid})$ 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 $(\textsc{Input}, v_i, \text{sid})$ from the adversary on behalf of P_i, set $x_i = v_i$.

In the version where $\mathcal{F}_{\text{SYNC}}^{\texttt{ua}}$ provides security with unanimous abort, the adversary can in addition choose to set the output to $\bot$ for all parties after learning the output.

B Proof of the Protocol Compiler

In this section, we show the proof of the Theorem 1, stated in Section 4.

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

Proof. **Completeness.** We first show that the protocol is complete. That is, if there are no corruptions, no environment can distinguish the real world from the ideal world. To this end, we need to argue that the output the parties obtain in both worlds are exactly the same. Observe that even if the adversary does not corrupt any party, it can still delay messages.

Given that the time-out occurs after $\tau_{\texttt{asynch}} = T_{\texttt{asynch}}(\delta) + T_{zk}(\delta) + \delta$ clock ticks and there are no corruptions, every honest party obtains output during the asynchronous phase. More concretely, each honest party obtains an output $[y_{\texttt{asynch}}]$ from $\Pi_{\texttt{aMPC}}$ and manages to collect a list L of $n-t$ signatures on this ciphertext during the asynchronous phase, decrypts $[y_{\texttt{asynch}}]$ and obtains the output $y_{\texttt{asynch}}$.

Soundness. To argue soundness, we first describe the simulator. The simulator $\mathcal{S}_{hyb}$ has to simulate the view of the dishonest parties during the protocol execution.

Algorithm $\mathcal{S}_{hyb}$

Clock / Timeout At every activation, the simulator does the following:

```
1: Query $\mathcal{G}_{\text{CLK}}$ for the current time and updates $\tau$ accordingly.
2: Send $(\text{CheckTimeOut}, \text{sid})$ to $\mathcal{G}_{\text{TIMEOUT}}$. If the response is
   $(\text{CheckTimeOut}, \texttt{true}, \text{sid})$, set $\texttt{sync} = \texttt{true}$, $\tau_{\texttt{sync}} = \tau$.
```

Network Messages:

The simulator prepares a set $\texttt{buffer} = \varnothing$ to simulate the messages that are sent to corrupted parties throughout the simulation (recall the variable $\texttt{buffer}$ in $\mathcal{F}_{\text{NET}}$). More concretely, it does the following:

```
1: On input $\delta$ from $\mathcal{F}_{\text{HYB}}$, output $\delta$ to the adversary.
2: On input $(\text{FetchMessages}, i)$ from $P_i$, for each message tuple
   $(T_{\texttt{init}}, T_{\texttt{end}}, P_k, P_i, m, \texttt{id}_m)$ from $\texttt{buffer}$ where $T_{\texttt{end}} \leq \tau$, output $(k, m)$
   to $P_i$.
3: On input $(\text{Delay}, D, \texttt{id})$ from the adversary, if there exists a tuple
   $(T_{\texttt{init}}, T_{\texttt{end}}, P_i, P_j, m, \texttt{id})$ in $\texttt{buffer}$ and $T_{\texttt{end}} + D \leq T_{\texttt{init}} + \delta$, then set
   $T_{\texttt{end}} = T_{\texttt{end}} + D$ and return $(\text{Delay-ok})$ to the adversary. Otherwise, ignore
   the message.
```

Setup:

```
1: The simulator generates the keys at the beginning of the execution. That
   is, it computes $(\texttt{ek}, \texttt{dk}) \leftarrow \mathsf{Gen}_{(n-t,n)}(1^\kappa)$, where $\texttt{dk} = (\texttt{dk}_1, \ldots, \texttt{dk}_n)$, and
   $(\texttt{vk}_j, \texttt{sk}_j) \leftarrow \mathsf{SigGen}(1^\kappa)$ for each party $P_j$. Then, it records the tuple
   $(\text{sid}, \texttt{ek}, \texttt{dk}, \texttt{vk}, \texttt{sk})$, where $\texttt{vk} = (\texttt{vk}_1, \ldots, \texttt{vk}_n)$ and $\texttt{sk} = (\texttt{sk}_1, \ldots, \texttt{sk}_n)$.
2: On input $(\text{GetKeys}, \text{sid})$ from a corrupted party $P_i$, send output
   $(\text{sid}, \texttt{ek}, \texttt{dk}_i, \texttt{vk}, \texttt{sk}_i)$ to $P_i$.
```

Asynchronous Phase:

It receives the time output $\tau_{\texttt{asynch}}$ from $\mathcal{F}^{\texttt{fs}}_{\text{HYB}}$. It keeps a variable τ_i for each party P_i.

```
    // Internal emulation of $\Pi_{\text{aMPC}}$
```

```
1: Emulate the messages of the protocol $\mathit{\Pi}_{\texttt{aMPC}}$. If a corrupted party $P_i$ is supposed to gets an output from the protocol, output $c_0 = [0]$, an encryption of 0.
   // Internal emulation of $\mathcal{F}_{\text{NET}}$.
2: As soon as $\tau_i = 0$ for an honest party $P_i$, input to buffer, the tuple $(\tau, \tau+1, i, j, [y], \mathsf{Sign}([y], \mathtt{sk}_i)), \mathtt{id})$, for each party $P_j$ and freshly generated id. Output $(\text{SENT}, i, j, ([y], \mathsf{Sign}([y], \mathtt{sk}_i))), \mathtt{id})$ to the adversary.
3: As soon as the current time $\tau$ is such that there are $n-t$ tuples $(\tau_1, \tau_2, j, i, ([y], \mathsf{Sign}([y], \mathtt{sk}_j)))$ such that $\tau_2 \leq \tau$ for the same $i$ in buffer, input to buffer the tuple $(\tau, \tau+1, i, j, ([y], L'), \mathtt{id})$, for each $P_j$, where $L'$ contains the list of signatures.
   // Internal emulation of $\mathit{\Pi}_{\text{ZK}}$.
4: The simulator internally emulate the delays of $\mathit{\Pi}_{\texttt{ZK}}$. Upon receiving an output $y$ from $\mathcal{F}^{\texttt{fs}}_{\text{HYB}}$, it computes an encryption $[y]$ under the key ek. Then, it computes the decryption shares of the corrupted parties $d_i = \mathtt{DecShare}_{\mathtt{sk}_i}(c_0)$, and sets the decryption shares from honest parties such that $(d_1, \ldots, d_n)$ forms a secret sharing of the output value $y$.
5: Every time the adversary requests validity of the decryption share $d_i$ from an honest party $P_i$, it responds with a confirmation of the validity of $d_i$.
6: Every time a corrupted $P_i$ provides a proof of correct decryption $(c', d')$, check whether $c' = c_0$ and $d' = d_i$. If so, record that a correct proof of decryption was input by $P_i$.
   // Delivery of honest parties' outputs.
7: On input $(\text{OUTPUT}, P_i, \text{sid})$ from $\mathcal{F}^{\texttt{fs}}_{\text{HYB}}$, where $P_i$ is an honest party, if $P_i$ obtained $n-t$ correct decryption shares, input $(\text{DELIVEROUTPUT}, P_i, \text{sid})$ to $\mathcal{F}^{\texttt{fs}}_{\text{HYB}}$.
```

Synchronous Phase:

```
   // Internal emulation of $\mathit{\Pi}_{\text{sBC}}$
1: The simulator emulates the messages from the broadcast protocol. For each emulated honest party $P_i$ that received a valid pair $([y], L)$ in the asynchronous phase, output $([y], L)$ to the adversary after $T_{\texttt{BC}}(\varDelta)$ clock ticks.
2: On input a valid pair $([y], L)$ from the adversary, after $T_{\texttt{BC}}(\varDelta)$ start the emulation of $\mathit{\Pi}_{\texttt{ZK}}$.
   // Internal emulation of $\mathit{\Pi}_{\text{ZK}}$
3: Output the message $(c_0, d_i)$ at the corresponding time, for each honest $P_i$. That is, keep a local delay $u_i$ for each honest party, which can be updated on input $(\text{DELAY}, D, \mathtt{id})$, and output the message if $\tau \geq \tau_{\mathsf{sync}} + T_{\texttt{BC}}(\varDelta) + u_i$.
   // Internal emulation of $\mathit{\Pi}_{\text{sMPC}}$.
4: If no valid pair was received from the adversary, and no honest party received a valid pair in the asynchronous phase, emulate the messages of the synchronous MPC protocol.
```

Tamper Function:

```
1: On input $(\text{TAMPEROUTPUT}, P_i, y_i', \text{sid})$ from the adversary, forward the input to $\mathcal{F}^{\texttt{fs}}_{\text{HYB}}$.
2: On input $(x_1, \ldots, x_n)$ from $\mathcal{F}^{\texttt{fs}}_{\text{HYB}}$, output it to the adversary.
```

```
3: When the adversary blocks an output from the asynchronous
   MPC protocol in the real world, the simulator forwards the input
   $(\textsc{BlockAsynchOutput}, P_i, \text{sid})$ to $\mathcal{F}_{\text{HYB}}^{\texttt{fs}}$.
```

We need to prove that the real and ideal worlds are indistinguishable. First, we remark that the simulator emulates the network by keeping a variable `buffer` which stores the messages that are sent. If a corrupted party inputs a message to $\mathcal{F}_{\text{NET}}$ in the real world, the simulator inputs the corresponding tuple to `buffer` exactly the same way as $\mathcal{F}_{\text{NET}}$. Moreover, the simulator have to input to `buffer` all messages that are sent from honest parties to corrupted parties in the real world. One can see that such messages correspond to signatures on an encrypted output and lists of such signatures. All these messages can be simulated. Observe that the simulator uses an encryption of 0 instead of $[y]$ in all the messages above. By the security of the threshold encryption scheme, both messages are indistinguishable. We remark that the simulator has knowledge of all the keys from the parties, since it simulates the setup functionality $\mathcal{F}_{\text{SETUP}}$.

Now we analyze each phase individually.

Setup Phase. It is straightforward to see that the messages that the adversary sees during the setup phase are identical in both worlds. This is because the simulator executes the key generation algorithms for both the threshold encryption and the digital signature scheme as the functionality $\mathcal{F}_{\text{SETUP}}$ in the ideal world.

Asynchronous Phase. We argue that the view of the adversary is indistinguishable in both worlds.
Internal emulation of $\Pi_{\texttt{aMPC}}$. The simulator keeps a delay variable τ_i for each party P_i, which it sets the same way as the adversary. When $\tau_i = 0$, a corrupted party P_i gets the encryption $[y]$ in the real world. In the ideal world, the simulator outputs an encryption of 0, $c_0 = [0]$, when $\tau_i = 0$ as well.
Internal emulation of $\mathcal{F}_{\text{NET}}$. In the real world, the corrupted parties obtain two types of messages after obtaining the ciphertext $[y]$: signatures on $[y]$ and lists of signatures. Once an honest party obtains $[y]$ from the asynchronous MPC protocol, it inputs to $\mathcal{F}_{\text{NET}}$ a signature of $[y]$ towards every party. Then, when $n - t$ signatures are collected, the honest party inputs the list to $\mathcal{F}_{\text{NET}}$ towards every party.

The simulator maintains a variable `buffer` which stores the messages that are sent via the network. It then inputs signatures of $[0]$ on behalf of each honest party P_i to `buffer`, towards every party (in particular, towards corrupted parties), and at the corresponding time. Once $n-t$ signatures are collected with destination P_i, the simulator emulates internally the protocol of P_i, and inputs to `buffer` the corresponding list, towards every party.
Internal emulation of $\Pi_{\texttt{ZK}}$. The simulator keeps a delay variable u_i for each party P_i, which it sets the same way as the adversary. When the delay is met, a corrupted party P_i gets a proof of correct decryption $([y], d_i)$, where $d_i = \texttt{DecShare}_{\texttt{sk}_i}([y])$ from $\Pi_{\texttt{ZK}}$ in the real world. In the ideal world, the simulator

outputs a pair (c_0, d_i), where $d_i = \texttt{DecShare}_{\texttt{sk}_i}(c_0)$, where the decryption shares from honest parties are set such that they reconstruct the value y.
Delivery of honest parties' outputs. The simulator has the power to deliver the outputs of honest parties in the ideal world. Hence, it delivers the outputs at the corresponding time. Namely, when the honest party has the output ciphertext $[y]$ and collects $n - t$ decryption shares in the real world.

Synchronous Phase. We argue again that the view of the adversary is exactly the same in both worlds.
Internal emulation of $\Pi_{\texttt{sBC}}$. In the real world, the parties broadcast all valid pairs $([y], L)$ that were received in the Asynchronous phase. This behavior is emulated by the simulator as follows: the simulator keeps track of the honest parties that obtained a valid pair $([y], L)$ during the asynchronous phase. The simulator then internally emulates $\Pi_{\texttt{sBC}}$ and outputs the valid pairs $([y], L)$ at the end of the broadcast round, after $T_{\texttt{BC}}$ clock ticks. Also, if the adversary inputs a valid pair $([y], L)$ during the broadcast round, it also outputs the valid pair $([y], L)$ to each party at the corresponding time.
Internal emulation of $\Pi_{\texttt{ZK}}$. After the round of synchronous broadcasts terminated, if a valid pair $([y], L)$ was received, then in the real world the honest parties send the decryption shares along with proofs of correct decryption using $\Pi_{\texttt{ZK}}$. In the ideal world, the simulator the internal emulation of $\Pi_{\texttt{ZK}}$ is similar to the one during the asynchronous phase.
Internal emulation of $\Pi_{\texttt{sMPC}}$. If no valid pair was received, in the real world the parties execute $\Pi_{\texttt{sMPC}}$, whose behavior is directly emulated by the $\mathcal{F}^{\texttt{fs}}_{\text{HYB}}$ functionality in the ideal world. That is, the simulator forwards the output from $\mathcal{F}^{\texttt{fs}}_{\text{HYB}}$ to the adversary.

All that is left to do is to argue about the messages the adversary obtains from breaking the correctness, privacy and termination thresholds.

Full Security. In the real world, if the adversary corrupts more than T parties, it can set the output of the asynchronous protocol $\Pi_{\texttt{aMPC}}$ to any output y, and it can also obtain the inputs from the honest parties. In this case, the simulator learns the inputs from the honest parties as well and can set the output correspondingly.

Similarly, if the adversary corrupts more than $n - 2t$ parties, it can forge a list of signatures on any value and choose the output, potentially violating security. But in this case the simulator can also set the output of $\mathcal{F}^{\texttt{fs}}_{\text{HYB}}$ in the ideal world and learn the inputs from $\mathcal{F}^{\texttt{fs}}_{\text{HYB}}$, since the full security threshold is $\min(T, n - 2t)$.

Termination. We remark that even if the responsiveness bound t of is violated, all the adversary can do in the real world is to prevent a party to obtain an output from $\Pi_{\texttt{aMPC}}$. Hence, responsiveness is lost and the simulator will block the output from the asynchronous phase.

$\square$

One can prove similarly the theorem with the variant where the hybrid offers unanimous abort.

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

Proof. The proof is exactly the same as in Theorem 1, except that the emulation of the synchronous MPC protocol is according to the messages of the protocol $\Pi^{\texttt{ua}}_{\texttt{sMPC}}$ that gives security with unanimous abort instead of full security. □

C ABA with Increased Consistency

C.1 Ideal Functionality

We introduce the asynchronous functionality for Byzantine Agreement, $\mathcal{F}_{\text{ABA}}$. The asynchronous Byzantine Agreement functionality $\mathcal{F}_{\text{ABA}}$ can be seen as a instantiation of the asynchronous MPC functionality $\mathcal{F}_{\text{ASYNC}}$ introduced in Section 4.5 with a specific function and tamper function. The function $f^{\mathcal{F}_{\text{ABA}}}$ to evaluate is defined as follows: If the honest parties in the core set have preagreement on an input value x, the output value is also x. Otherwise, the output value is the same for every honest party, but is defined by the adversary.

We define the functionality $\mathcal{F}_{\text{ABA}}$ to be an asynchronous MPC functionality $\mathcal{F}_{\text{ASYNC}}$ evaluating the function $f^{\mathcal{F}_{\text{ABA}}}$, and parametrized by the tamper function $\mathsf{Tamper}^{\text{BA}}_{t_v,t_c,t_l}$ defined below.

Definition 7. *We say that a Byzantine Agreement functionality has validity, consistency and termination parameters $T = (t_v, t_c, t_l)$ if it has the following tamper function* $\mathsf{Tamper}^{\text{BA}}_{t_v,t_c,t_l}$*:*

Function $\mathsf{Tamper}^{\text{BA}}_{t_v,t_c,t_l}(x_1, ..., x_n, \mathcal{H})$

$(c, p, d) = \mathsf{Tamper}^{\text{BA}}_{t_v,t_c,t_l}(x_1, ..., x_n, \mathcal{H})$, *where:*

- $c = 0$ *if and only if* $|\mathcal{P} \setminus \mathcal{H}| \leq t_v$ *and there exists* x *such that for all* $P_i \in \mathcal{H}$: $x_i = x$, *or* $|\mathcal{P} \setminus \mathcal{H}| < t_c$.
- $p = 1$.
- $l = 1$ *if and only if* $|\mathcal{P} \setminus \mathcal{H}| \geq t_l$.

C.2 Protocol Description

In this section we show how to increase the consistency of an ABA protocol by sacrificing liveness.

In the following, we describe a protocol which operates with PKI setup $\mathcal{F}_{\text{PKI}}$ and uses a secure ABA protocol $\Pi_{\texttt{aBA}}$ with parameters (t_v, t_c, t_l) as primitive. It then realizes a binary asynchronous Byzantine Agreement functionality with the same validity $t'_v = t_v$ and termination $t'_l = t_l$, but with consistency $t'_c < n - 2t_l$.

The protocol is quite simple. First, each party P_i run with input x_i the protocol $\Pi'_{\mathtt{aBA}}$, and once an output x is obtained, 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'. The idea is that there cannot be two lists of $n - t_l$ signatures on different values if there are up to $t_c < n - 2t_l$ corruptions.

Protocol $\Pi^{\mathsf{con}}_{\mathtt{aBA}}(P_i)$

Setup:

```
1: Input $(\textsc{GetDSSKeys}, \text{sid})$ to $\mathcal{F}_{\text{PKI}}$. Let the signing key be $\mathtt{sk}$ and the corresponding verification key $\mathtt{vk}$.
```

Asynchronous Phase: Upon every activation, progress with the following list of instructions. If not possible, output $(\textsc{ClockReady})$ to $\mathcal{G}_{\text{CLK}}$.

```
1: On input $x_i$, execute $\Pi_{\mathtt{aBA}}$ on input $x_i$. Let $x$ denote the output.
2: Compute the signature $\sigma = \mathsf{Sign}(x, \mathtt{sk})$.
3: Input $(\textsc{Send}, i, j, (x, \sigma))$, for each party $P_j$, to $\mathcal{F}_{\text{NET}}$.
4: Upon receiving $\ell \geq n - t$ valid messages of the form $(x', \sigma)$ from $\mathcal{F}_{\text{NET}}$, let $L = (x', \sigma_1, \ldots, \sigma_l)$ be the list containing these $\ell$ signatures on $x'$. Input $(\textsc{Send}, i, j, L)$, for each party $P_j$, to $\mathcal{F}_{\text{NET}}$, and terminate with output $x'$.
```

Let $\tau_{\mathsf{aba}}(\delta)$ denote the running time of $\Pi'_{\mathtt{aBA}}$ that has validity, consistency and termination parameters (t_v, t_c, t_l), $t_l \leq \frac{n}{3}$.

Lemma 3. *The protocol $\Pi^{\mathsf{con}}_{\mathtt{aBA}}$ operates with PKI setup $\mathcal{F}_{\text{PKI}}$, and is a secure ABA protocol with validity, consistency and termination parameters (t_v, t'_c, t_l), for any $t'_c < n - 2t_l$. The maximum delay for the output is $\tau_{\mathsf{con}} = \tau_{\mathsf{aba}}(\delta) + \delta$.*

Proof. **Completeness.** We first argue that if the adversary does not corrupt any party, the real world and the ideal world are indistinguishable. The output is the same in both worlds. If every party has the same input b, in the real world, $\Pi'_{\mathtt{aBA}}$ outputs b, and then each party signs b and collects $n - t_l$ signatures on b. This implies that the parties terminate with output b, which is the value that is output in the ideal world as well. The same happens if the parties do not hold the same input. In this case, in the real world, each party obtains the input x_1, signs this value, collects $n - t_l$ signatures and terminates with output x_1. This is also the output of the ideal world.

Soundness. We start describing the simulator. The job of the simulator $\mathcal{S}_{con}$ is to simulate the view of the adversary during the protocol execution. For readability, let us denote the ideal world Byzantine agreement functionality with improved consistency $\mathcal{F}_{\text{ABA}}$.

On a very high level, the simulator simulates internally the messages that the real world functionalities $\mathcal{F}_{\text{NET}}$, $\mathcal{F}_{\text{SETUP}}$ and the protocol $\Pi'_{\mathtt{aBA}}$ output to the adversary. In order to simulate the messages that the adversary obtains from the asynchronous network $\mathcal{F}_{\text{NET}}$, the simulator simply keeps the variable `buffer` as in

$\mathcal{F}_{\text{NET}}$, which records the messages sent via $\mathcal{F}_{\text{NET}}$ in the real world, with the delays of the messages. It also records the delays that the adversary inputs, and only delivers the messages when the corresponding party fetches the messages and the delay of the message is 0. To simulate the messages from $\mathcal{F}_{\text{SETUP}}$, the simulator executes the DSS key generation algorithm at the onset of the execution, and outputs the signing keys of the corrupted parties and all the verification keys to the adversary. Finally, to simulate the messages from $\Pi'_{\mathtt{aBA}}$, the simulator waits for the adversary to define a *core set* $\mathcal{I}$ (which by default is the set of honest parties), and after all parties in $\mathcal{I}$ provide his input bit, the simulator computes the output as in $\Pi'_{\mathtt{aBA}}$: if there is preagreement on a value x, that is the output, and otherwise, the output corresponds to the input of the corrupted party with lowest index.

Algorithm $\mathcal{S}_{con}$

Network Messages:

The simulator prepares a set $\mathtt{buffer} = \varnothing$ to simulate the messages that are sent to corrupted parties throughout the simulation (recall the variable $\mathtt{buffer}$ in $\mathcal{F}_{\text{NET}}$). More concretely, it does the following:

```
1: On input $(\text{FETCHMESSAGES}, i)$ from $P_i$, for each message tuple $(0, P_k, P_i, m, \mathtt{id}_m)$ in $\mathtt{buffer}$, output $(k, m)$ to $P_i$.
2: On input $(\text{DELAY } \mathcal{F}_{\text{NET}}, T, \mathtt{id})$ from the adversary, if there exists a tuple $(D, P_i, P_j, m, \mathtt{id})$ in $\mathtt{buffer}$ then set $D = D + T$ and return $(\text{DELAY-OK})$ to the adversary. Otherwise, ignore the message.
```

Setup:

```
1: The simulator generates the keys at the beginning of the execution. That is, it computes $(\mathtt{vk}_j, \mathtt{sk}_j) \leftarrow \mathsf{SigGen}(1^\kappa)$ for each party $P_j$. Then, it records the tuple $(\text{sid}, \mathtt{vk}, \mathtt{sk})$, where $\mathtt{vk} = (\mathtt{vk}_1, \ldots, \mathtt{vk}_n)$ and $\mathtt{sk} = (\mathtt{sk}_1, \ldots, \mathtt{sk}_n)$.
2: On input $(\text{GETKEYS}, \text{sid})$ from a corrupted party $P_i$, send $(\text{sid}, \mathtt{vk}, \mathtt{sk}_i)$ to $P_i$.
```

Main:

```
1: On input $(\text{NO-INPUT}, \mathcal{P}', \text{sid})$ from the adversary, set a variable $\mathcal{I} = \mathcal{H} \setminus \mathcal{P}'$, and forward $(\text{NO-INPUT}, \mathcal{P}', \text{sid})$ to $\mathcal{F}_{\text{ABA}}$.
2: Upon receiving the input $b_i$ from honest party $P_i$ or the adversary on behalf of a party, set $x_i^{BA} = b_i$. Moreover, if it is from the adversary, forward $x_i^{BA}$ to $\mathcal{F}_{\text{ABA}}$.
3: On input $(\text{OUTPUT}, x, \text{sid})$ from $\mathcal{F}_{\text{ABA}}$, output $(\text{OUTPUT}, x, \text{sid})$ to the adversary.
4: Emulate the messages of the sub-protocol $\Pi_{\mathtt{aBA}}$ by keeping the delays of each honest party.
5: As soon as $P_i$ obtains output from $\Pi_{\mathtt{aBA}}$, input to $\mathtt{buffer}$, on behalf of $P_i$, the tuple $(\tau, \tau+1, i, j, (x, \mathsf{Sign}(x, \mathtt{sk}_i)), \mathtt{id})$ for each corrupted party $P_j$ and freshly generated $\mathtt{id}$. Output $(\text{SENT}, i, j, (x, \mathsf{Sign}(x, \mathtt{sk}_i)), \mathtt{id})$ to the adversary.
6: Once there are $n - t_l$ tuples of the form $(\tau_1, \tau_2, j, i, (x', \mathsf{Sign}(x', \mathtt{sk}_j)))$ have been delivered from $\mathtt{buffer}$ to a fixed honest party $P_i$, input, for each $j$, to $\mathtt{buffer}$ the tuple $(\tau, \tau + 1, i, j, L, \mathtt{id})$, where $L$ contains the list of signatures on the value $x'$. Output $(\text{SENT}, i, j, L, \mathtt{id})$ to the adversary.
```

```
7: Keep track of the delays so that the parties receive the output at the same
   time as in the real world.
```

Tamper Function:

```
1: On input (TamperOutput, $P_i, y'_i$, sid), where $P_i$ is honest, from the adver-
   sary, forward the input to $\mathcal{F}_{\text{ABA}}$.
2: On input $(x_1, \ldots, x_n)$ from $\mathcal{F}_{\text{ABA}}$, output it to the adversary.
3: On input (BlockOutput, $P_i$, sid), where $P_i$ is honest, from the adversary,
   forward the input to $\mathcal{F}_{\text{ABA}}$.
```

In order to prove that the real world and the ideal world are indistinguishable, we divide cases depending on the adversary's capabilities.

If the validity threshold is satisfied, i.e. $|\mathcal{P} \setminus \mathcal{H}| \leq t_v$ and the parties in the core-set have the same input, or the consistency threshold is satisfied, i.e. $|\mathcal{P} \setminus \mathcal{H}| \leq t_c$, then $\mathit{\Pi}_{\mathtt{aBA}}$ ensures that the output at Step 1 is consistent among the honest parties. Let us denote this value x. In this case, if $|\mathcal{P} \setminus \mathcal{H}| \leq t_l$, then every honest party eventually receives a list of $n - T_L$ signatures on x. In the ideal world, the output is x as well. Otherwise, if $|\mathcal{P} \setminus \mathcal{H}| > t_l$, some honest parties may not receive a list of $n - t_l$ signatures on x, and hence they do not receive any output. For these honest parties, the simulator blocks the output value of these parties.

On the other hand, if it is not the case that $|\mathcal{P} \setminus \mathcal{H}| \leq t_v$ where the parties in the core-set have the same input, nor the consistency threshold is satisfied, i.e. $|\mathcal{P} \setminus \mathcal{H}| > t_c$, then it is not guaranteed that the output after Step 1. (from $\mathcal{F}_{\text{ABA}}$) is consistent. However, we still need that if $|\mathcal{P} \setminus \mathcal{H}| \leq t'_c < n - 2t_l$, all final outputs are consistent. That is the case, because there cannot be two lists of signatures of size at least $n - t_l$ on different values. Assume towards contradiction, that there are such two lists. Observe that any two lists of size $n - t_l$, intersect in at least $n - 2t_l$ parties. Since $|\mathcal{P} \setminus \mathcal{H}| \leq t'_c < n - 2t_l$, there must be at least one honest party in this intersection. But honest parties do not send signatures on different values.

Moreover, let us remark that in the real world, the parties only send messages in Step 2 via the network, and in Step 1 via the protocol $\mathit{\Pi}'_{\mathtt{aBA}}$. This means, since the adversary can only delay each network message by up to δ clock ticks, and the output from $\mathcal{F}_{\text{ABA}}$ up to $\tau_{\mathtt{aba}}(\delta)$ clock ticks, then the maximum delay for the output is $\tau_{\mathtt{con}} = \tau_{\mathtt{aba}}(\delta) + \delta$. Hence, it is enough that the simulator has the power to delay the output up to $\tau_{\mathtt{con}}$ clock ticks.

□

If we assume an asynchronous Byzantine Agreement $\mathit{\Pi}'_{\mathtt{aBA}}$ which runs concurrently in expected constant time as in [5], with validity, consistency and termination for any $t < \frac{n}{3}$ corruptions, we obtain the following corollary:

Lemma 1. *There exists a protocol which realizes $\mathcal{F}_{\text{ABA}}$ with validity, consistency and termination parameters (t_v, t_c, t_l), for any $t_l < \frac{n}{3}$, $t_l \leq t_v < \frac{n}{3}$ and $t_c < n - 2t_l$, in the $(\mathcal{G}_{\text{CLK}}, \mathcal{F}^{\delta}_{\text{NET}}, \mathcal{F}_{\text{PKI}})$-hybrid world. The expected maximum delay for the output is $\tau_{aba} = O(\delta)$.*

D Proof of Theorem 3

In this section, we prove the theorem stated in Section 5.2.

Theorem 3. *The protocol $\Pi_{\mathtt{aMPC}}$ uses $\mathcal{F}_{\text{SETUP}}^{\mathtt{FHE}}$ 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 $\tau_{\mathtt{asynch}} = \tau_{\mathtt{aba}}(\delta) + 2\tau_{\mathtt{zk}}(\delta) + 9\delta$.*

Proof. **Completeness.** We first show that the protocol is complete. It is easy to see that, if there are no corruptions, no environment can distinguish the real world from the ideal world. First, observe that the output that is evaluated in both worlds is the same, since the simulator sets the core set containing the same parties as in the real world. Moreover, the simulator delivers the outputs of honest parties at the time at which the honest parties obtain the output and terminate in the real execution.

One can readily verify, that in the protocol, the parties send messages in 9 steps, performs calls to $\Pi_{\mathtt{ZK}}$ in two steps, and executes in parallel n BAs during the input provider selection. Hence, the protocol takes at most $\tau_{\mathtt{aba}}(\delta)+2\tau_{\mathtt{zk}}(\delta)+9\delta$ clock ticks to execute.

Soundness. At a very high level, the consistency property of $\Pi_{\mathtt{aBA}}$ can affect both correctness and privacy of the overall SFE. Moreover, it is important that the validity of $\Pi_{\mathtt{aBA}}$ is higher than the termination threshold t_l. Otherwise, when parties wait for the input ciphertexts from each $j \in \mathtt{CoreSet}$, it might be that no party has this input ciphertext and the protocol does not terminate. Given that $t_v \geq t_l$, then in the region of thresholds where there are up to t_l corruptions, validity is guaranteed to hold and hence in the input phase parties are guaranteed to collect all tuples $(j, (c_j, \pi_j))$ such that $j \in \mathtt{CoreSet}$. Let us now describe the simulator.

Algorithm $\mathcal{S}_{\mathtt{MPC}}$

Network Messages:

The simulator prepares a set $\mathtt{buffer} = \varnothing$ to simulate the messages that are sent to corrupted parties throughout the simulation (recall the variable $\mathtt{buffer}$ in $\mathcal{F}_{\text{NET}}$). More concretely, it does the following:

```
1: Let $\delta$ be the network delay, received from $\mathcal{F}_{\text{ASYNC}}$
2: On input (FETCHMESSAGES, $i$) from $P_i$, for each message tuple
   $(0, P_k, P_i, m, \mathtt{id}_m)$ in $\mathtt{buffer}$, output $(k, m)$ to $P_i$.
3: On input (DELAY $\mathcal{F}_{\text{NET}}, T, \mathtt{id}$) from the adversary, if there exists a tuple
   $(D, P_i, P_j, m, \mathtt{id})$ in $\mathtt{buffer}$ and $T \leq \delta$, then set $D = D + T$ and return
   (DELAY-OK) to the adversary. Otherwise, ignore the message.
```

Setup:

```
1: The simulator generates the keys at the beginning of the execution. That is, it
   computes and records $(\mathtt{ek}, \mathtt{dk}) \leftarrow \mathsf{Gen}_{(n-t_l,n)}(1^\kappa)$, where $\mathtt{dk} = (\mathtt{dk}_1, \ldots, \mathtt{dk}_n)$.
```

2: On input (GetKeys, sid) from a corrupted party P_i, output $(\text{sid}, \texttt{ek}, \texttt{dk}_i)$ to P_i.

Input Stage:

// Plaintext Knowledge and Distribution.

1: Set $c_i = \texttt{Enc}_{\texttt{ek}}(0)$, for each honest party P_i.

2: The simulator keeps track of the delays the adversary sets for the outputs from $\Pi_{\texttt{ZK}}$. Then, when the adversary requests the output of P_i from $\Pi_{\texttt{ZK}}$ at the corresponding time, the simulator responds with a confirmation of the validity of the ciphertext c_i.

3: On input $\sigma_j^{\texttt{popk}}$ from corrupted party P_j to P_i, input the tuple $(\tau, \tau + 1, P_j, P_i, \sigma_j^{\texttt{popk}}, \texttt{id})$ to $\texttt{buffer}$.

4: When a corrupted party P_i inputs $((\texttt{ek}, c_i), (x_i, r_i))$ to prove plaintext knowledge of c_i to a party P_j, the simulator checks that $c_i = \texttt{Enc}_{\texttt{ek}}(x_i, r_i)$. If so, it inputs $(\tau, \tau + 1, P_j, P_i, \sigma_j^{\texttt{popk}}, \texttt{id})$ to $\texttt{buffer}$.

5: As soon as there are $n - t$ tuples $(\tau_1, \tau_2, P_j, P_i, \sigma_j^{\texttt{popk}}, \texttt{id})$ for different P_j, such that $\tau \geq \tau_2$ in $\texttt{buffer}$, then compute $\pi_i = \{\sigma_j^{\texttt{popk}}\}$ and input $(\tau, \tau + 1, i, j, (c_i, \pi_i), \texttt{id})$ for each P_j.

6: On input (c_i, π_i) from a corrupted party P_i to P_j, the simulator inputs $(\tau, \tau + 1, P_i, P_j, (c_i, \pi_i), \texttt{id})$ to $\texttt{buffer}$.

7: As soon as there is a tuple $(\tau_1, \tau_2, P_j, P_i, (c_j, \pi_j), \texttt{id})$, such that $\tau \geq \tau_2$ in $\texttt{buffer}$, input a signature to $\texttt{buffer}$. That is, input $(\tau, \tau + 1, i, j, \sigma_i^{\texttt{dist}}, \texttt{id})$ to $\texttt{buffer}$.

8: As soon as there are $n - t$ tuples $(\tau_1, \tau_2, P_j, P_i, \sigma_j^{\texttt{dist}}, \texttt{id})$ for different P_j, such that $\tau \geq \tau_2$ in $\texttt{buffer}$, then start simulating the input provider selection.

// Input Providers.

9: For each party P_i, keep track of the parties which successfully proved plaintext knowledge to P_i. We denote that set S_i.

10: The simulator inputs to $\texttt{buffer}$ each set S_i towards every party. That is, input $(\tau, \tau + 1, i, j, S_i, \texttt{id})$ to $\texttt{buffer}$, for each P_j.

11: Once an emulated honest party P_i received $n - t$ such sets, emulate for that party the execution of the BAs. That is, input a 1 to P_j's BA, if P_j is in one of the received sets. Take into account all the commands tampering the outputs or blocking the outputs of the BAs that come from the adversary, and change the output accordingly.

12: Wait until there are $n - t$ ones as outputs from the BAs. Then, input 0 to the remaining BAs.

13: Define $\texttt{CoreSet}^i$ as the set of parties such that the emulated BA for that party outputted 1. Observe that if the adversary corrupted more than $n - 2t$, the consistency of the BAs is not satisfied, since $t_c < n - 2t$, and hence the core sets can be different.

14: The simulator emulates each party P_i, by inputting the pairs (c_j, π_j) that it collected in the $n - t$ sets S_j, to $\texttt{buffer}$.

Computation and Threshold Stage:

// Setting the Core Set.

1: Once the simulator computes $\texttt{CoreSet}^i$ from the previous Stage, do the following: if the core sets are consistent, it sends to $\mathcal{F}_{\text{ASYNC}}$ the input values x_i

```
   from each corrupted party, and also inputs (No-Input, $\mathcal{P} \setminus \texttt{CoreSet}, \texttt{id}$) to
   $\mathcal{F}_{\text{ASYNC}}$. It obtains the output $y$. Otherwise, input any of the core sets $\texttt{CoreSet}^i$
   to $\mathcal{F}_{\text{ASYNC}}$. Then, obtain the inputs from honest parties (if the core set are not
   consistent, $f \geq n - 2t$, the simulator is allowed to obtain the inputs since
   privacy is not satisfied).
   // Computation.
2: For each honest party $P_i$, the simulator internally computes the evaluated
   ciphertext $c^i = f_{\texttt{ek}}(c_1, \ldots, c_{|\texttt{CoreSet}^i|})$, based on the ciphertext from the input
   providers.
   // Threshold Decryption.
3: The simulator computes the decryption share $d_i = \texttt{DecShare}_{\texttt{dk}_i}(c^i)$ for each
   corrupted party $P_i$, and sets the decryption shares from honest parties such
   that $(d_1, \ldots, d_n)$ forms a secret sharing of the output value $y$, if the core sets
   are consistent. Otherwise, for each honest $P_i$ it can evaluate the function on
   the inputs in $\texttt{CoreSet}^{\texttt{i}}$ to obtain $y_i$, encrypt it, and set the decryption share
   exactly as in the real world. In this case, the simulator also fixes the output
   of $P_i$ to $y_i$.
4: Each time the adversary requests validity of the decryption share $d_i$ from an
   honest party $P_i$, the simulator responds with a confirmation of the validity
   of $d_i$.
5: As soon as the adversary inputs a decryption share $d_i$ for ciphertext $c'$, the
   simulator checks the validity of the decryption share, and if it is valid, inputs
   to buffer a signature on $(d_i, c')$.
6: Once an emulated honest party $P_i$ received $n - t$ signatures on the same
   pair $(d_i, c')$, it computes a proof that the decryption share $d_i$ for $c'$ is correct
   $\texttt{ProofShare}_i = \{\sigma_j^{\texttt{pocs}}\}$. It inputs to buffer the tuple $((d_i, c'), \texttt{ProofShare}_i)$
   to every party.
7: When an honest party receives $n - t$ tuples of the form $((d_i, c'), \texttt{ProofShare}_i)$
   with the same $c'$, it sets his output bit to $y$.
Termination Stage:
1: The simulator keeps track of the votes that each party performs. That is, if
   an emulated honest party $P_i$ received an output $y$ in the previous stage, it
   inputs $y$ to buffer, towards every other party.
2: As soon as an emulated honest party receives $n - 2t$ votes on $y$, if the party
   $P_i$ did not vote yet, it sets its output to $y$, and inputs $y$ to buffer, towards
   every other party.
3: As soon as an emulated honest party receives $n - t$ votes on $y$, the simulator
   delivers the party's output in the ideal world.
```

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 computation stage. Here, when a corrupted party requests a proof of decryption share from an honest party, the simulator simply gives a valid response without checking the witness from the honest party.

Hybrid 3. This is similar to Hybrid 2, but in the computation of the decryption shares is different. In this case, the simulator obtains the output y from $\mathcal{F}_{\text{ASYNC}}$, 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. That is, here the simulator does not need to know the secret key share of honest parties to compute the decryption shares. If there are more than $n - 2t$ corrupted parties, privacy is broken, so the simulator obtains the inputs from the honest parties and computes the decryption shares as in the previous hybrid.

Hybrid 4. We modify the previous hybrid in the Input Stage. Here, when a corrupted party requests a proof of plaintext knowledge from an honest party, the simulator simply gives a valid response without checking the witness from the honest party.

Hybrid 5. 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 6. 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 have a valid witness in Π_{ZK}. ■

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 $n - t$, any number corrupted decryption shares below $n - t$ 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. Above $n - t$ corruptions, the simulator obtains the inputs from honest parties, and hence both hybrids are trivially indistinguishable. ■

Claim 3. No efficient environment can distinguish between Hybrid 3 and Hybrid 4.
Proof: This follows trivially, since the honest parties always have a valid witness in Π_{ZK}. ■

Claim 4. No efficient environment can distinguish between Hybrid 4 and Hybrid 5.
Proof: This follows from the semantic security of the encryption scheme. ■

Claim 5. No efficient environment can distinguish between Hybrid 5 and Hybrid 6.
Proof: This follows, because the simulator in the ideal world and the simulator in Hybrid 5 emulate internally the joint behavior of the ideal assumed functionalities, exactly the same way. ■

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

□

E Fully Homomorphic Encryption Scheme

In this section, we briefly recall the sintax of a fully homomorphic encryption scheme. One can find the security definition for example in [31, 1].

Definition 8. *A fully homomorphic encryption scheme consists of four algorithms:*

– *Key generation:* $(\mathtt{ek}, \mathtt{dk}) = \mathsf{Gen}(1^\kappa)$*, where* $\mathtt{ek}$ *is the public encryption key and* $\mathtt{dk}$ *is the decryption key.*
– *Encryption:* $c = \mathtt{Enc}_{\mathtt{ek}}(m; r)$ *denotes an encryption with key* $\mathtt{ek}$ *of a plaintext* m *with randomness* r*, to obtain ciphertext* c*.*
– *Decryption:* $m = \mathtt{Dec}_{\mathtt{dk}}(c)$ *denotes a decryption of ciphertext* c *with key* $\mathtt{dk}$ *to obtain plaintext* m*.*
– *Homomorphic evaluation:* $c = f_{\mathtt{ek}}(c_1, \ldots, c_n)$ *denotes the evaluation of a circuit* f *over a tuple of ciphertexts* $(c_1, \ldots, c_n)$ *to obtain* c*.*