Robust MPC: Asynchronous Responsiveness yet Synchronous Security

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] julian.loss@rub.de, Ruhr-University Bochum
[3] talm@idc.ac.il, IDC Herzliya
[4] tschudi@cs.au.dk. Aarhus University

Abstract. Two paradigms for secure MPC are synchronous and asynchronous protocols, which differ substantially in terms of the guarantees they provide. 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 are as fast as the actual network allows, i.e., run in time proportional to the actual maximal network delay δ, but unavoidably parties with slow network connections cannot give input.

This paper proposes a new, composable model (of UC functionalities) capturing the best of both worlds. Each party obtains the output as fast as the network allows (a property called responsiveness), and it is guaranteed that all parties obtain the same output. We consider different corruption thresholds: correctness, privacy, and responsiveness are guaranteed for less than T_C, T_P, and T_R corruptions, respectively, while termination is always guaranteed. We achieve a trade-off between correctness, privacy and responsiveness: For any $T_R \leq \frac{1}{3}n$, one can achieve $T_C = T_P = \min\{\frac{1}{2}n, n-2T_R\}$. In particular, setting $T_R = \frac{1}{4}n$ allows us to obtain $T_C = T_P = \frac{1}{2}n$, hence achieving substantial responsiveness, yet correctness and privacy much better than in an asynchronous protocol and as good as for a purely synchronous (slow) protocol.

This result is achieved by a black-box compiler for combining an asynchronous and a synchronous protocol, involving new protocol techniques that may have applications in other contexts, and by devising an asynchronous protocol with $T_C = T_P = n-2T_R$, improving the correctness and privacy of known protocols achieving $T_C = T_P = \frac{1}{3}n$.

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 messages sent by honest parties can be delayed by at most some a priori bounded time. Synchronous protocols typically proceed in rounds of length Δ, ensuring that any message sent at the beginning of a round by an honest party will arrive by the end of that round at its intended recipient. On the upside, such strong timing assumptions allow to obtain protocols with an optimal resilience of $\frac{n}{2}$ corruptions. On the downside, Δ has to be chosen rather pessimistically. This means that even for small *actual* maximal network delay δ, i.e., $\delta \ll \Delta$, a synchronous protocol runs in time that depends on Δ, thus failing 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 a priori bounded network delays to work properly. As such, they function correctly under much more realistic network assumptions in which messages can be arbitrarily delayed and arrive out of order. Moreover, asynchronous protocols have the benefit of running at the *actual speed of the network*, i.e., they run in time that depends only on δ, but *not* on Δ; a notion that we shall refer to as *responsiveness* [34]. This speed and robustness comes at a price, however: it can easily be seen that no asynchronous protocol

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

that implements an arbitrary function can tolerate $\frac{n}{3}$ maliciously corrupted parties. A natural question is whether it is possible to obtain hybrid protocols that combine beneficial properties from both regimes, i.e., can be fast under a well-behaved network, but remain resilient above $\frac{n}{3}$ corrupted parties as long as synchrony is ensured. Indeed, several recent works have addressed this question in the context of byzantine agreement (BA) [33] and state-machine-replication (SMR) [34]. All of the aforementioned works give optimistic guarantees for responsiveness using a fast asynchronous path while offering a slow, but resilient synchronous fallback path in case the fast path fails. The main challenge is to trade off properties in a non-trivial fashion, i.e., without 'overwriting' one of the paths by running the other.

1.1 Contributions

In this work, we extend the scope of hybrid protocols to the realm of MPC. We make the following contributions.

The model. We first introduce a new composable model of hybrid functionalities in the UC framework, 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. Our ideal hybrid functionality admits composition in a black-box fashion without sacrificing the responsiveness guarantees of any of the composed components. This is in stark contrast to the aforementioned works for which the issue of composition is left mostly as an open question and security is not proven in the UC framework.

New protocols. We show a protocol that always terminates, and has a trade-off between correctness, privacy and responsiveness: for any responsiveness threshold $T_R \leq \frac{1}{3}n$, we achieve a correctness and privacy threshold of $T_C = T_P = n - 2T_R$. That is, for $T_R = \frac{1}{4}n$, we achieve a reasonable amount of responsiveness, and keep the same correctness and privacy as synchronous protocols. To this end, we follow two steps, which might be of independent interest:

1. *Black-Box compiler.* We give a generic black-box compiler that combines an asynchronous protocol for SFE with a synchronous protocol for SFE and gives a hybrid protocol that combines beneficial properties from both the synchronous and asynchronous regime, roughly in the following way: Assuming synchronous protocols with correctness and privacy $\frac{1}{2}n$, and an asynchronous protocol with correctness, privacy and responsiveness (T_C^a, T_P^a, T_R^a), the compiler provides correctness, privacy and responsiveness $(\min\{T_C^a, \frac{n}{2}\}, \min\{T_P^a, \frac{n}{2}\}, T_R^a)$. Compared to the above works, the main challenge that we have to overcome in our compiler is to preserve the privacy of both protocol components– an issue that does not show up for BA or SMR.
2. *Improved asynchronous protocol.* Finally, we show how to modify the asynchronous MPC protocol by Cohen [10] to obtain the trade-off mentioned above when used in our aforementioned compiler. That is, we achieve an asynchronous protocol with correctness and privacy $T_C = T_P = n - 2T_R$, improving the correctness and privacy of current asynchronous protocols achieving $T_C = T_P = \frac{1}{3}n$.

1.2 Technical Overview of Our Results

The Model. In the real world, we assume a communication network with *unknown* upper bound δ, corresponding to the maximal network delay. This value is unknown to the honest parties' protocol. The protocols make use of a conservatively assumed worst-cast delay $\Delta \gg \delta$. Within δ, the adversary can schedule the messages arbitrarily.

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. Each party can obtain output in either phase, but not in both. 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 Δ.

The ideal functionality $\mathcal{F}_{\text{HYB}}$ gives different guarantees depending on the number of corruptions that the adversary is allowed to do. If less than T_R parties are corrupted, $\mathcal{F}_{\text{HYB}}$ ensures responsiveness. This

means that every honest party outputs during the asynchronous phase. On the downside, $\mathcal{F}_{\text{HYB}}$ unavoidably may ignore up to T_R inputs from honest parties. In contrast, if no honest party outputs during the asynchronous phase, the outputs obtained during the synchronous phase are guaranteed to take into account all inputs from honest parties (but are received much later). Moreover, regardless of when parties output, as long as less than T_C parties are corrupted, each honest party outputs the correct (and identical) output. And as long as less than T_P parties are corrupted, honest parties' inputs remain private. To allow $\mathcal{F}_{\text{HYB}}$ to give different guarantees depending on the number of corruptions, we consider a tuple of thresholds and model the adversary's capabilities in a fine-grained manner, such as tampering outputs, learning the honest parties inputs' or blocking the outputs based on the thresholds. For example, if the correctness threshold is violated, we allow the adversary to modify the outputs; if the responsiveness threshold is violated, we allow the adversary to block the outputs during the asynchronous phase.

To smoothly model the transition between the two phases, $\mathcal{F}_{\text{HYB}}$ operates with respect to a global clock $\mathcal{G}_{\text{CLK}}$ and a global timeout functionality $\mathcal{G}_{\text{TIMEOUT}}$. The goal of $\mathcal{G}_{\text{TIMEOUT}}$ is to communicate from the environment a point in time at which the functionality switches from one phase to the other. This gives a very general way to model a timeout, since it allows $\mathcal{G}_{\text{TIMEOUT}}$ to send the timeout signal to $\mathcal{F}_{\text{HYB}}$ at a time which is not a-priori fixed or within an interval (for example, the timeout could be sent when an agreement protocol finishes).

Compiler. 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 the timeout event occurs. Upon timing out, the parties switch to a synchronous computation. If the network is well behaved and sufficiently many parties are honest, the honest parties can hope to obtain their output at the actual speed of network. The main challenge is to ensure that if even a single honest party obtains output during the asynchronous phase, the remaining honest parties do not recompute the output 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 recomputed 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 t out of n parties can jointly decrypt the ciphertext by pooling their shares. This way, if we set $t = n - T_R$, where T_R is the responsiveness threshold, we are ensured that sufficiently many parties will pool their shares during the asynchronous phase, given that the network is sufficiently well-behaved and fewer than T_R 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 key 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 key 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 key shares to learn the output from the asynchronous phase.

An Asynchronous MPC Protocol with Parameter Tradeoffs. In the last part of our paper, we show how to modify the protocol of [10] to sacrifice termination for improved correctness and privacy. Concretely, we start from Cohen's protocol which achieves correctness, privacy and termination all simultaneously for the corruption threshold $\frac{1}{3}n$. At a high level, the idea of this protocol is to use a threshold homomorphic encryption scheme with threshold $t = \frac{1}{3}n$ 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 $t = \frac{1}{3}n$, we set $t = \frac{3}{4}n$. Intuitively, assuming a perfect 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 correctness and privacy $\frac{1}{3}n$.

We show how to improve the correctness and privacy thresholds of the protocol by using, as a subcomponent, the recent BA protocols proposed by Loss and Moran [33], which offer to smoothly trade termination for validity and consistency. Our protocol inherits the thresholds of the protocols in [33], giving a correctness and privacy threshold of $T_C = T_P = \frac{1}{2}n > \frac{n}{3}$.

1.3 Comparison with Known Approaches for Synchronous Speedup

The astute reader might wonder whether the approach we have presented above is actually necessary to attain the trade-offs that our work aims for. A well-known approach (see e.g. [32]) to 'speed up' a synchronous protocol, i.e., make it fully responsive, 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 even if there are up to T_R corruptions, where $T_R \leq \frac{1}{3}n$.

On the Concrete Speed of Synchronous Protocols. Given that synchronous protocols run in time that depends on Δ, they may perform poorly compared to asynchronous protocols, which run in time δ. Of course, this depends on how the unknown delay δ and the known upper bound Δ compare with each other. If $\delta \approx \Delta$, there is no benefit in running an asynchronous protocol, since the adversary can delay the messages up to Δ. However, typically we think of $\delta \ll \Delta$, and hence even constant-round synchronous protocols are very slow (as they output at $k\Delta$ clock ticks), compared to the asynchronous counterpart (which outputs at $T(d)$ ticks). We remark that our asynchronous phase is actually extremely fast, as it runs in constant time (i.e., outputs at kd ticks).

1.4 Related Work

Despite being a very natural direction of research, compilers for achieving tradeoffs between asynchronous and synchronous protocol have only begun to be studied in relatively recent works. Pass and Shi study a hybrid type of SMR protocol in [34] 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 after the adversary decides to corrupt a given party 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* [35]. 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 even in the presence of a fully adaptive adversary, but can easily be kept on the slow, synchronous path forever in this case. Subsequently, Loss and Moran [33] showed how to obtain compilers for the simpler case of BA that achieve smooth tradeoffs between responsiveness and safety properties even when confronted a fully adaptive adversary. They also showed that the tradeoffs achieved by their main compiler are optimal. Finally, Guo et al. [25] propose a protocol which follows the fast/slow path approach in an alternative model which weakens classical synchrony.

Further Related Work. Best-of-both worlds compilers for distributed protocols (in particular MPC protocols) come in many flavors and we are only able to list an incomplete summary of related work. Goldreich and Petrank [24] give a blackbox 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 [31] 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 $\frac{n}{3}$ corrupted parties in order to still achieve safety and liveness. A line of works [3],[4],[9],[36] consider the setting where parties have a few rounds of synchronous communication before switching to fully asynchronous computation. In this setting, one can achieve protocols with better security guarantees than purely asynchronous ones.

2 Preliminaries

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.

Threshold Fully Homomorphic Encryption. We assume the existence of a secure public-key encryption scheme, which is fully homomorphic and enables threshold decryption.

Definition 1. *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*.*

The security definition is defined for example in [22].

Definition 2. *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* $\mathtt{dk}$ *is represented via a* t*-out-of-*n *secret sharing* $(\mathtt{dk}_1, \ldots, \mathtt{dk}_n)$*.*
- *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

3.1 Communication Network and Clocks

We borrow ideas from a standard model for UC synchronous communication [29, 30]. There, parties have access to functionalities and global functionalities [7]. More concretely, parties have access to a synchronized global clock functionality $\mathcal{G}_{\text{CLK}}$, and a network functionality $\mathcal{F}_{\text{NET}}$ of pairwise authenticated communication channels with an unknown upper bound on the message delay. In our model, parties also have access to a global timeout functionality $\mathcal{G}_{\text{TIMEOUT}}$, which tells each party when it must switch the execution from asynchronous to synchronous.

At a high level, the model captures the two guarantees that parties have in the synchronous models. 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 and a round-ready flag $d_i = 0$, for each party P_i. Each clock tick, the clock functionality 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.

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 τ.

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 $\mathcal{F}_{\text{NET}}$ 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 the corresponding 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.

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

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

Message transmission:

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

The parties also have access to a global timeout functionality $\mathcal{G}_{\text{TIMEOUT}}$. As mentioned above, the goal of the timeout is to communicate from the environment the point in time at which it must switch the execution from asynchronous to synchronous. Observe that this way to model the timeout is more general than fixing a point in time in which the parties switch the execution, as it allows to capture situations in which the timeout is given at a point in time which is not a priori fixed (e.g. when an agreement

protocol finishes, or a certain event happens). Moreover, we choose to model the timeout as an explicit global functionality that guarantees that all parties are notified at the same time, in contrast to making a restriction to the environment. This allows to explicitly construct the timeout functionality, and not make any assumptions on the environment.

The functionality stores a flag, which is initially set to false. At soon as the environment inputs the message (TIMEOUT, sid), the timeout is set to true in the next clock tick. This ensures that the timeout is received by the parties at the beginning of the same clock tick.

One can generalize this functionality to allow the timeout to differ among parties, as long as they occur within some known interval of time. Then, when executing the synchronous phase, one could set a round length large enough such that it accommodates for all timeouts and the known upper bound on the network delay Δ. But in this paper, we will describe our protocols relative to the simpler global functionality.

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

The timeout functionality is connected to a clock functionality $\mathcal{G}_{\text{CLK}}$. It stores a flag `timeout` which initially is set to false. It stores the current time τ and a value t which is set to $t = \perp$.
Each time the functionality is activated it first queries $\mathcal{G}_{\text{CLK}}$ for the current time and updates τ accordingly. If $t > 0$ and $\tau > t$ it sets `timeout` to true.

Timeout Input:

1: On input of (TIMEOUT, sid) from the environment, if $t = \perp$ set $t = \tau$.
2: On input of (CHECKTIMEOUT, sid) from a party, a functionality, or the adversary return (CHECKTIMEOUT, `timeout`, sid).

3.2 Ideal World

It is known that asynchronous protocols can only tolerate up to $t < \frac{n}{3}$ active corruptions and can ignore up to t inputs from honest parties. However, the benefit is that the time at which the parties obtain output only depends on the actual delay δ of the network, and not on a known upper bound Δ. On the other hand, synchronous protocols can tolerate up to $t < \frac{n}{2}$ corruptions, but the output is obtained at a time that depends on Δ.

Hybrid SFE: We introduce an ideal functionality $\mathcal{F}_{\text{HYB}}$ which allows to capture the guarantees that asynchronous and synchronous protocols for secure function evaluation offer in a fine-grained manner.

In a nutshell, it gives different guarantees depending on when the timeout occurs. If the timeout occurs after $O(\delta)$ clock ticks, it allows honest parties to evaluate and obtain the fast output as in an asynchronous protocol. On the other hand, if the timeout occurs earlier, the parties are guaranteed to obtain a slow output, at a time which depends on Δ. Moreover, we will see that one can realize the functionality $\mathcal{F}_{\text{HYB}}$ with correctness and privacy parameters beyond $t < \frac{n}{3}$, which is the optimal threshold for purely asynchronous protocols.

More concretely, the functionality $\mathcal{F}_{\text{HYB}}$ has access to the two global functionalities $\mathcal{G}_{\text{CLK}}$ and $\mathcal{G}_{\text{TIMEOUT}}$, and allows parties to evaluate a function f. It has two phases, separated by a timeout event: an asynchronous phase and a synchronous phase. In the asynchronous phase, the adversary can adaptively delay the outputs up to a time which depends solely on the actual network delay δ. During this phase, the function to be evaluated is allowed to ignore up to T_R inputs from honest parties, which the adversary can explicitly choose. Observe that for $T_R < \frac{n}{3}$, we take into account more inputs from honest parties than current asynchronous protocols, which can ignore up to $\frac{n}{3}$ of the inputs from honest parties.

Once the time out happened, the parties switch to the synchronous phase. In this phase, it is guaranteed that the honest parties obtain output at a time which depends on the known upper bound Δ. Moreover, if an honest party got an output during the asynchronous phase (where up to T_R honest inputs can be ignored), it is guaranteed that every other honest party also gets the same output. If no party obtained an output during the first phase, the parties evaluate the function f taking into account all inputs from honest parties.

The functionality $\mathcal{F}_{\text{HYB}}$ is a functionality which provides different guarantees depending on the set of parties the adversary corrupts during the protocol. To model that, we introduce a tamper function Tamper_T. The tamper function, parameterized by a tuple of thresholds T, allows to model the capabilities of the adversary, depending on the set of corrupted parties and the parties' inputs. This generalization allows to naturally capture SFE functionalities which have different guarantees with corruption thresholds for correctness, privacy and termination. Moreover, we also allow the tamper function to depend on the actual inputs of the parties. This allows to capture typical conditions of byzantine agreement or broadcast protocols, such as validity or consistency.

Tamper Function for $\mathcal{F}_{\text{HYB}}$. In the case of $\mathcal{F}_{\text{HYB}}$, the capabilities we consider are correctness, privacy, responsiveness and output delivery.

Definition 3. *We define a hybrid functionality with correctness, privacy, responsiveness and termination parameters $T = (T_C, T_P, T_R, T_L)$ if it has the following tamper function* $\mathsf{Tamper}^{\text{HYB}_T}$*:*

Function $\mathsf{Tamper}^{\text{HYB}_T}$

$(c, p, r, d) = \mathsf{Tamper}^{\text{HYB}_T}(x_1, ..., x_n, \mathcal{H})$*, where:*
- $c = 1$ *if and only if* $|\mathcal{P} \setminus \mathcal{H}| \geq T_C$.
- $p = 1$ *if and only if* $|\mathcal{P} \setminus \mathcal{H}| \geq T_P$.
- $r = 1$ *if and only if* $|\mathcal{P} \setminus \mathcal{H}| \geq T_R$
- $d = 1$ *if and only if* $|\mathcal{P} \setminus \mathcal{H}| \geq T_L$.

We introduce the formal description of $\mathcal{F}_{\text{HYB}}$ for a generic tamper function. In Section 4, we show how to realize $\mathcal{F}_{\text{HYB}}$ with the tamper function presented in Definition 3. In short, if the correctness threshold is satisfied, the honest parties have the guarantee that the adversary could not tamper with the outputs for honest parties. When privacy is satisfied, the honest parties have the guarantee that the adversary does not learn its inputs apart from the output. Responsiveness is modeled as a guarantee for the honest parties to obtain an output at a latest time which depends solely on δ, if there are less than T_R corruptions and the timeout did not trigger. Finally, if the termination threshold is satisfied, it guarantees that every honest party gets an output.

We describe $\mathcal{F}_{\text{HYB}}$ in a generic fashion with 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 in the synchronous phase in the case an output was delivered (or not) during the asynchronous phase. One can think of $\tau_{\texttt{asynch}} = O(\delta)$, and $\tau_{\texttt{OD}} = \tau_{\texttt{OND}} = O(\Delta)$.

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

$\mathcal{F}_{\text{HYB}}$ is connected to a global clock $\mathcal{G}_{\text{CLK}}$ and timeout functionality $\mathcal{G}_{\text{TIMEOUT}}$.
The functionality is parametrized by δ, $\tau_{\texttt{asynch}}$, $\tau_{\texttt{OD}}$, $\tau_{\texttt{OND}}$, Tamper_T, and the function to evaluate f.
The functionality stores variable `OutputDelivered`, τ, a variable τ_i for each P_i, $\tau_{\texttt{tout}}$, `sync`, x_i, y_i, v_i. These variables are initialized as `OutputDelivered` = `false`, $\tau_{\texttt{tout}} = -1$, $\tau = 0$, `sync` = `false`, $x_i = \perp$, and $y_i = w_i = \perp$.
It keeps a set $\mathcal{C} = \varnothing$.

Timeout/Clock :

Each time the functionality is activated query $\mathcal{G}_{\text{CLK}}$ for the current time and updates τ accordingly.
Then, send (CheckTimeOut, sid) to $\mathcal{G}_{\text{TIMEOUT}}$. If the response is (CheckTimeOut, `true`, sid), set `sync` = `true` and $\tau_{\texttt{tout}} = \tau$. If `OutputDelivered` = `false`, compute $y_1 = \cdots = y_n = f(x_1, \ldots, x_n)$.

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

- At the onset of the execution, output δ and $\tau_{\texttt{asynch}}$ to the adversary.
- On input (Input, v_i, sid) from party P_i:
  - If some party has received output, ignore this message. Otherwise, set $x_i = v_i$.
  - If $x_i \neq \perp$ 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 (Input, P_i, sid) to the adversary.
- On input (GetOutput, 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 $(\text{OUTPUT}, y_i, \text{sid})$ to P_i and set $\texttt{OutputDelivered} = \texttt{true}$.
- Otherwise, output $(\text{OUTPUT}, i, \text{sid})$ to the adversary.

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

– On input $(\text{GETOUTPUT}, \text{sid})$ from party P_i
- If $\texttt{OutputDelivered} = \texttt{true}$ and $\tau \geq \tau_{\texttt{tout}} + \tau_{\texttt{OD}}$ and $w_i \neq \texttt{blocked}$, it outputs $(\text{OUTPUT}, y_i, \text{sid})$ to P_i.
- If $\texttt{OutputDelivered} = \texttt{false}$ and $\tau \geq \tau_{\texttt{tout}} + \tau_{\texttt{OND}}$ and $w_i \neq \texttt{blocked}$, it outputs $(\text{OUTPUT}, y_i, \text{sid})$ to P_i.

Adversary

Upon each party corruption, update $(c, p, r, d) = \mathsf{Tamper}_T((x_1, \ldots, x_n), \mathcal{H})$.

// Core Set and Delivery of Outputs

1: Upon receiving a message $(\text{NO-INPUT}, \mathcal{P}', \text{sid})$ from the adversary, if $\texttt{sync} = \texttt{false}$, $\mathcal{P}'$ is a subset of $\mathcal{P}$ of size $|\mathcal{P}'| < T_R$ and $y_1 = \cdots = y_n = \perp$, set $\mathcal{I} = \mathcal{H} \setminus \mathcal{P}'$.
2: On input $(\text{DELIVEROUTPUT}, i, \text{sid})$ from the adversary, if $y_i \neq \perp$, $\tau < \tau_{\texttt{asynch}}$ and $\texttt{sync} = \texttt{false}$, output $(\text{OUTPUT}, y_i, \text{sid})$ to P_i and set $\texttt{OutputDelivered} = \texttt{true}$.

// Adversary's capabilities

3: On input $(\text{TAMPEROUTPUT}, P_i, y_i', \text{sid})$ 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 $(\text{BLOCKASYNCHOUTPUT}, P_i, \text{sid})$ from the adversary, if $r = 1$ and $\texttt{sync} = \texttt{false}$, set $w_i = \texttt{aBlocked}$.
6: On input $(\text{BLOCKOUTPUT}, P_i, \text{sid})$ from the adversary, if $d = 1$, set $w_i = \texttt{blocked}$.

4 Compiler

In this section, we present a protocol which realizes the ideal functionality $\mathcal{F}_{\text{HYB}}$ in the $(\mathcal{G}_{\text{CLK}}, \mathcal{G}_{\text{TIMEOUT}}, \mathcal{F}_{\text{NET}}^{\delta}, \mathcal{F}_{\text{SETUP}}, \mathcal{F}_{\text{ASYNC}}, \mathcal{F}_{\text{SYNC}}, \mathcal{F}_{\text{sBC}})$-hybrid world. The clock $\mathcal{G}_{\text{CLK}}$, the timeout functionality $\mathcal{G}_{\text{TIMEOUT}}$ and the network $\mathcal{F}_{\text{NET}}^{\delta}$ with unknown upper bound δ are described in Section 3.1. $\mathcal{F}_{\text{SETUP}}$ is a functionality which distributes to the parties keys for a threshold encryption scheme and a digital signature scheme. The functionality $\mathcal{F}_{\text{SYNC}}$ models a secure function evaluation synchronous functionality. A concrete functionality is $\mathcal{F}_{\text{sBC}}$, which corresponds to a byzantine broadcast functionality. Finally, $\mathcal{F}_{\text{ASYNC}}$ models a secure function evaluation asynchronous functionality. In the following subsections we describe all of these functionalities in detail.

Tamper Function for SFE. We first introduce a tamper function that models the capabilities of the adversary for a SFE functionality with correctness, privacy and termination parameters $T = (T_C, T_P, T_L)$. In this tamper function, the adversary can tamper with the output value, learn the inputs from honest parties or prevent the honest parties to obtain output, if and only if the number of corruptions is larger than T_C, T_P or T_L, respectively.

Definition 4. *We say that an SFE functionality has correctness, privacy and termination parameters $T = (T_C, T_P, T_L)$ if it has the following tamper function* $\mathsf{Tamper}_T^{\text{SFE}}$*:*

Function $\mathsf{Tamper}_T^{\text{SFE}}(x_1, ..., x_n, \mathcal{H})$

$(c, p, d) = \mathsf{Tamper}_T^{\text{SFE}}(x_1, ..., x_n, \mathcal{H})$*, where:*

- $c = 1$ *if and only if* $|\mathcal{P} \setminus \mathcal{H}| \geq T_C$.
- $p = 1$ *if and only if* $|\mathcal{P} \setminus \mathcal{H}| \geq T_P$.
- $d = 1$ *if and only if* $|\mathcal{P} \setminus \mathcal{H}| \geq T_L$.

At a very high level, the compiler assumes a $\mathcal{F}_{\text{ASYNC}}$ (resp. $\mathcal{F}_{\text{SYNC}}$) functionality with correctness, privacy and termination parameters (T_C^a, T_P^a, T_L) (resp. (T_C^s, T_P^s, n)). It then realizes the ideal functionality $\mathcal{F}_{\text{HYB}}$

with correctness, privacy, responsiveness and termination parameters $(\min\{T_C^a, T_C^s\}, \min\{T_P^a, T_P^s\}, T_L, n)$. Observe that the compiled functionality $\mathcal{F}_{\text{HYB}}$ terminates independently of the number of corruptions.

In Section 5, we give an asynchronous protocol that realizes $\mathcal{F}_{\text{ASYNC}}$ with correctness, privacy and termination parameters $(n-2T_L, n-2T_L, T_L)$, for any $T_L < \frac{n}{3}$. We can then combine this protocol with any synchronous protocol realizing $\mathcal{F}_{\text{SYNC}}$ with parameters $(\frac{n}{2}, \frac{n}{2}, n)$, and obtain a hybrid protocol with parameters $(\min\{n-2T_L, \frac{n}{2}\}, \min\{n-2T_L, \frac{n}{2}\}, T_L, n)$. Observe that in order to obtain a synchronous protocol realizing $\mathcal{F}_{\text{SYNC}}$ with parameters $(\frac{n}{2}, \frac{n}{2}, n)$, one can take any protocol secure against up to $\frac{n}{2}$ corruptions, unconditional [6, 8, 38, 1, 15, 21, 14, 26] or computational [23, 2, 16, 18, 17], and add a time out to ensure that parties terminate.

As an example, if one sets $T_L = \frac{1}{4}n$, the resulting protocol has the same parameters as a synchronous protocol for correctness, privacy and termination, $(\frac{n}{2}, \frac{n}{2}, n)$, and also has the benefit that the parties obtain the output at a latest time which only depends on δ, given that the adversary corrupts less than $\frac{n}{4}$ parties. We remark that synchronous protocols have to guarantee that messages sent in a certain round must arrive before the next round. Hence, they set the round length to a value $\Delta \gg \delta$ and terminate in a time which depends on Δ.

4.1 Key-Distribution Setup

Our 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 circuit 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 public key $\mathtt{ek}$ and a t-out-of-n share of the corresponding secret key $\mathtt{dk}_i$. Moreover, it gives to each party a pair of signing and verification key $(\mathtt{sk}, \mathtt{vk})$. Due to modularity and clarity in following sections, we describe the two setups, digital signature setup $\mathcal{F}_{\text{DSSKEYS}}$ and threshold encryption setup $\mathcal{F}_{\text{TEKEYS}}$ independently. The setup of the protocol consists of the combined functionality $\mathcal{F}_{\text{SETUP}} = [\mathcal{F}_{\text{DSSKEYS}}, \mathcal{F}_{\text{TEKEYS}}]$, which includes both functionalities.

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

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

$\mathcal{F}_{\text{DSSKEYS}}$ is parameterized by a digital signature scheme.

Key Distribution:

1: At the beginning of the execution, compute $(\mathtt{vk}_j, \mathtt{sk}_j) \leftarrow \mathsf{SigGen}(1^\kappa)$ for each party P_j. Then, record $(\text{sid}, \mathtt{vk}, \mathtt{sk})$, 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 P_i, send $(\text{sid}, P_i, \mathtt{vk})$ to the adversary, and $(\text{sid}, \mathtt{vk}, \mathtt{sk}_i)$ to P_i.

Threshold Encryption Setup. The protocol assumes also a threshold encryption setup, which allows each party to access a public key $\mathtt{ek}$ and a t-out-of-n share of the corresponding secret key $\mathtt{dk}_i$.

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

$\mathcal{F}_{\text{TEKEYS}}$ is parameterized by a threshold encryption scheme.

Key Distribution:

1: At the beginning of the execution, compute $(\mathtt{ek}, \mathtt{dk}) \leftarrow \mathsf{Gen}_{(t,n)}(1^\kappa)$, where $\mathtt{dk} = (\mathtt{dk}_1, \ldots, \mathtt{dk}_n)$. Then, record $(\text{sid}, \mathtt{ek}, \mathtt{dk})$.
2: On input $(\text{GETKEYS}, \text{sid})$ from party P_i, send $(\text{sid}, P_i, \mathtt{ek})$ to the adversary, and output $(\text{sid}, \mathtt{ek}, \mathtt{dk}_i)$ to P_i.

4.2 Synchronous SFE

The synchronous SFE functionality $\mathcal{F}_{\text{SYNC}}$ allows a set of n parties to evaluate a specific function f. It is connected to a global clock $\mathcal{G}_{\text{CLK}}$ and is parametrized by a time at which the honest parties start the execution, and a time at which the honest parties obtain the output. We model the synchronous functionality with deterministic termination, but one can extend this functionality to also model probabilistic termination using the frameworks presented in [12, 11].

Similar to $\mathcal{F}_{\text{HYB}}$, $\mathcal{F}_{\text{SYNC}}$ has different guarantees depending on the set of parties the adversary corrupts. This is modelled by a tamper function Tamper_T.

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

$\mathcal{F}_{\text{SYNC}}$ is connected to a global clock $\mathcal{F}_{\text{CLOCK}}$. $\mathcal{F}_{\text{SYNC}}$ is parameterized by a set $\mathcal{P}$ of n parties, a function f and a tamper function Tamper_T, and a delay time at which the parties obtain output $\tau_{\mathtt{sync}}$. Additionally, it initializes $\tau = 0$ and, for each party P_i, $x_i = y_i = \perp$. 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 $(\text{INPUT}, v_i, \text{sid})$ from each party P_i at a fixed time τ':
   - If $x_i = \perp$, it sets $x_i = v_i$.
   - If for each party $P_i \in \mathcal{P}$ $x_i \neq \perp$ and $y_i = \perp$, set $y_1 = \cdots = y_n = f(x_1, \ldots, x_n)$.
   - Set $\tau_{\mathtt{out}} = \tau' + \tau_{\mathtt{sync}}$.

2: On input $(\text{GETOUTPUT}, \text{sid})$ from honest party P_i or the adversary (for corrupted P_i), if $\tau \geq \tau_{\mathtt{out}}$ and $y_i \neq \top$, it outputs $(\text{OUTPUT}, y_i, \text{sid})$ to P_i.

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

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 $(x_1, \ldots, x_n)$ to the adversary.
3: On input $(\text{BLOCKOUTPUT}, P_i, \text{sid})$ from the adversary, if $d = 1$, set $y_i = \top$.

We denote a synchronous SFE functionality with correctness, privacy and termination parameters (T_C, T_P, T_L) the $\mathcal{F}_{\text{SYNC}}$ functionality with the tamper function presented in Definition 4.

4.3 Synchronous Byzantine Broadcast

We introduce the synchronous functionality for Broadcast, $\mathcal{F}_{\text{sBC}}$. The adversary's capabilities are modelled via the tamper function $\mathsf{Tamper}_T^{\text{BA}}$, which depends on the inputs of honest parties and the thresholds $T = (T_V, T_C, T_L)$, for validity, consistency and termination respectively. The tamper function guarantees that the adversary is not allowed to tamper the output value of the honest parties in the case the sender is honest, or the consistency threshold is satisfied. It is also explicitly stated that in any case the adversary obtains the sender's input. Finally, the adversary can make the honest parties not obtain output if the termination threshold is not satisfied.

Definition 5. *We say that a Byzantine Broadcast functionality has validity, consistency and termination parameters $T = (T_V, T_C, T_L)$ if it has the following tamper function $\mathsf{Tamper}_T^{\text{BC}}$:*

Function $\mathsf{Tamper}_T^{\text{BC}}(x_1, ..., x_n, \mathcal{H})$

$(c, p, d) = \mathsf{Tamper}_T^{\text{BC}}(x_1, ..., x_n, \mathcal{H})$, *where:*
- $c = 0$ *if and only if* $|\mathcal{P} \setminus \mathcal{H}| < T_V$ *and* $P_s \notin \mathcal{P} \setminus \mathcal{H}$, *or* $|\mathcal{P} \setminus \mathcal{H}| < T_C$.
- $p = 1$.
- $d = 1$ *if and only if* $|\mathcal{P} \setminus \mathcal{H}| \geq T_L$.

We define the synchronous Byzantine Broadcast functionality $\mathcal{F}_{\text{sBC}}$. It is defined to be a synchronous SFE functionality $\mathcal{F}_{\text{SYNC}}$, where the tamper function is according to Definition 5, and the function to evaluate $f^{\mathcal{F}_{\text{sBC}}}$ is defined as follows: The output value is the sender's input value x_s.

4.4 Asynchronous SFE

We borrow ideas from [29, 13] to model an asynchronous SFE functionality. In traditional asynchronous protocols, the parties are guaranteed to *eventually* receive output. That is, the adversary can delay the output of honest parties in an arbitrary but finite manner. This comes from the fact that the network that is assumed in these protocols, only guarantees eventual delivery. In our setting, the guarantee that is obtained from eventual delivery is not so useful, and hence we make the simple observation that if the network has an unknown upper bound δ, then asynchronous protocols actually achieve better guarantees. That is, that parties obtain output at a time which depends only on δ. We model this generically, and allow the adversary to delay the outputs of honest parties up to time $\tau_{\mathtt{asynch}}$, which typically is a function of δ. The guarantee obtained in an asynchronous SFE with eventual delivery (e.g. as in [13]) is a special case of our functionality, namely when $\tau_{\mathtt{asynch}} = \infty$. We describe it for the case where $\tau_{\mathtt{asynch}}$ is a fixed time, but one can model $\tau_{\mathtt{asynch}}$ to be probabilistic as well.

Similar to $\mathcal{F}_{\text{HYB}}$ and $\mathcal{F}_{\text{SYNC}}$, we also want to capture different guarantees depending on the actual set of corrupted parties. For this, we use the same idea and parameterize the functionality by a tamper function Tamper_T.

As known from asynchronous protocols, it is impossible to 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_L parties are corrupted, the parties can only wait for $n - T_L$ 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 $\mathcal{P}' \subseteq \mathcal{P}$, $|\mathcal{P}'| < T_L$, whose input is not considered. Therefore, the ideal functionality also allows the simulator to choose this set.

As in [13], and similar to the network functionality $\mathcal{F}_{\text{NET}}^{\delta}$, 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 Tamper_T, a delay δ, and a maximum delay $\tau_{\mathtt{asynch}}$. It initializes the variables $x_i = y_i = \bot$, $\tau_{\mathtt{in}} = \bot$ and $\tau_i = 0$ for each party $P_i \in \mathcal{P}$ and the variable $\mathcal{I} = \mathcal{H}$, where $\mathcal{H}$ is the set of honest parties.

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 $y_j = f(x'_1, \ldots, x'_n)$, where $x'_i = x_i$ for each $P_i \in \mathcal{I} \cup (\mathcal{P} \setminus \mathcal{H})$ and $x'_i = \bot$ otherwise. Set $\tau_{\mathtt{in}} = \tau$.
   - Output $(\text{INPUT}, P_i, \text{sid})$ 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 $|\mathcal{P}'| \leq T_L$ and $y_1 = \cdots = y_n = \bot$, set $\mathcal{I} = \mathcal{H} \setminus \mathcal{P}'$.

2: On input $(\text{SETOUTPUTTIME}, P_i, \tau', \text{sid})$ from the adversary, if $\tau_{\mathtt{in}} \neq \bot$ and $\tau' < \tau_{\mathtt{in}} + \tau_{\mathtt{asynch}}$, set $\tau_i = \tau'$.

Upon each party corruption, update $(c, p, d) = \mathsf{Tamper}_T((x_1, \ldots, x_n), \mathcal{H})$.

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 $(x_1, \ldots, x_n)$ to the adversary.

3: On input $(\text{BLOCKOUTPUT}, P_i, \text{sid})$ from the adversary, if $d = 1$, set $y_i = \top$.

An asynchronous SFE functionality with correctness, privacy and termination parameters (T_C, T_P, T_L) is the $\mathcal{F}_{\text{ASYNC}}$ functionality with the tamper function presented in Definition 4.

4.5 Protocol Compiler

The protocol operates in the $(\mathcal{G}_{\text{CLK}}, \mathcal{G}_{\text{TIMEOUT}}, \mathcal{F}_{\text{NET}}^{\delta}, \mathcal{F}_{\text{SETUP}}, \mathcal{F}_{\text{SYNC}}, \mathcal{F}_{\text{SBC}}, \mathcal{F}_{\text{ASYNC}})$-hybrid model. Parties know a delay upper bound $\Delta \gg \delta$ of $\mathcal{F}_{\text{NET}}^{\delta}$.

The protocol works with a key setup $\mathcal{F}_{\text{SETUP}}$, which distributes to the parties at a public key $\mathtt{ek}$, a $(n - T_L)$-out-of-n share of the decryption key $\mathtt{dk}$, a signing key $\mathtt{sk}_i$ and all the verification keys $\mathtt{vk}_j$ of the parties.

We assume an asynchronous SFE functionality $\mathcal{F}_{\text{ASYNC}}$ which evaluates the function $f' = \mathtt{Enc}_{\mathtt{ek}}(f)$. This corresponds to the function f, where the output is encrypted under the setup public key $\mathtt{ek}$. The $\mathcal{F}_{\text{ASYNC}}$ functionality has the parameters (T_C^a, T_P^a, T_L). We also need to assume a synchronous SFE functionality $\mathcal{F}_{\text{SYNC}}$ which evaluates the function f with parameters $(\frac{n}{2}, \frac{n}{2}, n)$. Moreover, we assume that every party has access to a broadcast functionality $\mathcal{F}_{\text{sBC}}$ with parameters $(\frac{n}{2}, \frac{n}{2}, n)$.

The main idea of the protocol is to allow the parties to first optimistically evaluate $\mathcal{F}_{\text{ASYNC}}$, and use the synchronous functionality $\mathcal{F}_{\text{SYNC}}$ as a robust fall-back. A challenge is to ensure that if an honest party obtains an output $y_{\mathtt{asynch}}$ during the asynchronous phase, then every other party obtains this output as well. We remark that even if the function to evaluate is the same, the output obtained by $\mathcal{F}_{\text{SYNC}}$ is not necessarily $y_{\mathtt{asynch}}$. This is because in an asynchronous functionality $\mathcal{F}_{\text{ASYNC}}$, up to T_L inputs from honest parties can be ignored. To solve this, we assume an asynchronous functionality $\mathcal{F}_{\text{ASYNC}}$ which evaluates the function $f' = \mathtt{Enc}_{\mathtt{ek}}(f)$. This corresponds to the function f, where the output is encrypted under the setup public key $\mathtt{ek}$. Then, we require that when a party obtains an encrypted output $[y]$, it signs this value and must collect $n - T_L$ signatures on this value.

Once a list of $n - T_L$ signatures is collected on a value $[y]$, the party forwards this list along with $[y]$, and also sends its decryption key share to every other party. Then, once a party collects $n - T_L$ decryption shares, it can obtain the output y by decrypting $[y]$.

Once the timeout from $\mathcal{G}_{\text{TIMEOUT}}$ is received, parties are instructed to broadcast all pairs (v, L) of value and list of at least $n - T_L$ signatures if such a pair was collected before during the asynchronous phase. If a party receives via broadcast any valid pair (v, L), then it sends its decryption share to every other party. Otherwise, it gives its input to $\mathcal{F}_{\text{SYNC}}$.

The main observation here, is that if an honest party collected a list L of $n - T_L$ signatures on a ciphertext $[y_{\mathtt{asynch}}]$, it must broadcast the pair $([y_{\mathtt{asynch}}], L)$, where L contains at least $n - T_L$ signatures during the broadcast round of the synchronous phase. Then, every honest party obtains at least a valid pair $([y_{\mathtt{asynch}}], L')$ after the broadcast round is finished. Since there cannot be two signature lists of size $n - T_L$ on different values, this value must be the ciphertext encrypting the output $[y_{\mathtt{asynch}}]$, and all parties are instructed to send their decryption shares. Once the parties send their decryption shares, every party obtains as output $y_{\mathtt{asynch}}$. On the other hand, if no honest party obtained such a pair during the asynchronous phase, it is guaranteed that the adversary did not learn $y_{\mathtt{asynch}}$, but it might be that the adversary collected a valid $([y_{\mathtt{asynch}}], L')$. The adversary can then decide whether to broadcast a valid pair. If it does, every party will hold this pair and everyone will output $y_{\mathtt{asynch}}$ as before. On the contrary, if it does not, no honest party holds a valid pair after the broadcast round, and every party can safely give their input to $\mathcal{F}_{\text{SYNC}}$.

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

The party stores the current time τ, a flag $\mathtt{sync} = \mathtt{false}$ and a variable $\tau_{\mathtt{sync}} = \perp$.

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

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

Setup:

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

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

– On input $(\text{INPUT}, x_i, \text{sid})$ (and following activations) do
  1: Send $(\text{INPUT}, x_i, \text{sid})$ to $\mathcal{F}_{\text{ASYNC}}$.
  2: Send $(\text{GETOUTPUT}, \text{sid})$ to $\mathcal{F}_{\text{ASYNC}}$ until you get an output $[y]$.
  3: Send $([y], \mathsf{Sign}([y], \mathtt{sk}))$ to every other party using $\mathcal{F}_{\text{NET}}$.
  4: Receive signatures and values via $\mathcal{F}_{\text{NET}}$ until you received $n - T_L$ signatures $L = (\sigma_1, \ldots, \sigma_l)$ on a value v.
  5: Send (v, L) to every party using $\mathcal{F}_{\text{NET}}$.

6: Receive message lists (v, L'). For each such list send (v, L') to every party using $\mathcal{F}_{\text{NET}}$.
7: Once done with the above, send the secret key share $\mathtt{dk}_i$ to each party using $\mathcal{F}_{\text{NET}}$.
8: Once $n - T_L$ key shares are received via $\mathcal{F}_{\text{NET}}$, reconstruct the value y from $[y]$ and output y.

– 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 $\mathtt{sync} = \mathtt{true}$ and $\tau \geq \tau_{\mathtt{sync}}$, stop all previous steps and do the following commands.

– On input (ClockReady) do:
1: Send (ClockReady) to $\mathcal{G}_{\text{CLK}}$.
2: **if** $\tau \geq \tau_{\mathtt{sync}}$ **then**
3: Input (v, L) to the synchronous BC functionality $\mathcal{F}_{\text{sBC}}$, for each pair (v, L) received during the Asynchronous Phase.
4: Wait until $\mathcal{F}_{\text{sBC}}$ terminated. If a pair (v, L) was received from $\mathcal{F}_{\text{sBC}}$, input the message $(\text{Send}, i, j, \mathtt{dk}_i)$, for each P_j, to $\mathcal{F}_{\text{NET}}$. Otherwise, input x_i to $\mathcal{F}_{\text{SYNC}}$.
5: **end if**
– On input $(\text{GetOutput}, \text{sid})$ send $(\text{GetOutput}, \text{sid})$ to $\mathcal{F}_{\text{sBC}}$. If there was an output $([y], L')$ from $\mathcal{F}_{\text{sBC}}$, wait for Δ clock ticks. After Δ clock ticks, $n - T_L$ key shares are received via $\mathcal{F}_{\text{NET}}$. Compute and reconstruct the value y from $[y]$. Output y. Otherwise, input $(\text{GetOutput}, \text{sid})$ to $\mathcal{F}_{\text{SYNC}}$ and output answer from $\mathcal{F}_{\text{SYNC}}$.

The following theorem is formally proven in Section A.

Theorem 1. *The protocol $\Pi^{\Delta}_{\mathtt{hyb}}$ operates in the $(\mathcal{G}_{\text{CLK}}, \mathcal{G}_{\text{TIMEOUT}}, \mathcal{F}^{\delta}_{\text{NET}}, \mathcal{F}_{\text{SETUP}}, \mathcal{F}_{\text{SYNC}}, \mathcal{F}_{\text{sBC}}, \mathcal{F}_{\text{ASYNC}})$-hybrid, with the following parameters:*

– *$\mathcal{F}^{\delta}_{\text{NET}}$ has unknown delay δ.*
– *$\mathcal{F}_{\text{SYNC}}$ evaluates function f and gives output after $T_{\mathtt{sync}}(\Delta)$ clock ticks. Moreover, it has parameters $(\frac{n}{2}, \frac{n}{2}, n)$ for correctness, privacy and termination.*
– *$\mathcal{F}_{\text{sBC}}$ gives output after $T_{\mathtt{BC}}(\Delta)$ clock ticks. Moreover, it has parameters $(\frac{n}{2}, \frac{n}{2}, n)$ for validity, consistency and termination.*
– *$\mathcal{F}_{\text{ASYNC}}$ evaluates function $f' = \mathtt{Enc}_{\mathtt{ek}}(f)$ and gives output after $T_{\mathtt{asynch}}(\delta)$ clock ticks. Moreover, it has parameters (T^a_C, T^a_P, T_L) for correctness, privacy and termination.*

For any $\Delta \geq \delta$, it realizes $\mathcal{F}_{\text{HYB}}$ with correctness, privacy, responsiveness and termination parameters $(\min(T^a_C, \frac{n}{2}), \min(T^a_P, \frac{n}{2}), T_L, n)$. The maximum delay of the asynchronous phase is $\tau_{\mathtt{asynch}} = T_{\mathtt{asynch}}(\delta) + 3\delta$, and of the synchronous phase is $\tau_{\mathtt{OD}} = T_{\mathtt{BC}}(\Delta) + \Delta$ if an output was delivered in the asynchronous phase, and otherwise is $\tau_{\mathtt{OND}} = T_{\mathtt{BC}}(\Delta) + T_{\mathtt{sync}}(\Delta)$.

5 Asynchronous Protocols

In this section, we show how to realize $\mathcal{F}_{\text{ASYNC}}$ with correctness and privacy parameters beyond $\frac{n}{3}$. More concretely, we realize $\mathcal{F}_{\text{ASYNC}}$ with correctness, privacy $T_C = T_P = n - 2T_L$, for any termination $T_L \leq \frac{1}{3}n$. We remark that currently known asynchronous protocols only tolerate up to $T_C = T_P = T_L = \frac{1}{3}n$.

The first step is to obtain an asynchronous Byzantine Agreement protocol with higher validity and consistency thresholds. For that, we borrow ideas from [33] and prove UC security of their protocols. The resulting BA protocol is then used to obtain an SFE with higher correctness and privacy thresholds.

As argued in Section 4.4, current asynchronous SFE functionalities only provide an eventual delivery guarantee on the output [13], where the adversary can delay the output of honest parties in an arbitrary but finite manner. The core reason for this is that the network functionality that is assumed in these protocols, only guarantees eventual delivery.

Asynchronous protocols do not rely on knowing any upper bound Δ on the network delay, as they work *no matter* what the upper bound is. However, they do provide some guarantees on the output delivery time if there exists an upper bound δ on the message delivery. That is, they the output is obtained at a time which depends only on the adversary's strategy, but there is a latest time at which the parties obtain output, which depends on δ. With the SFE functionality $\mathcal{F}_{\text{ASYNC}}$ introduced in Section 4.4, we make explicit this guarantee. As a special case, when $\delta = \infty$, one obtains the usual guarantees that an asynchronous SFE with eventual delivery provides (e.g. as in [13]).

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* receive 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 ABA with increased Validity and Consistency

In this section, we explain how to improve the validity and consistency parameters of a given ABA protocol. First, we define the functionality $\mathcal{F}_{\text{ABA}}$ to be an instantiation of $\mathcal{F}_{\text{ASYNC}}$ with a specific function to evaluate and a specific tamper function. Then, we show a protocol to increase the validity. Finally, we show how to increase the consistency.

Asynchronous Byzantine Agreement. We introduce the asynchronous functionality for Byzantine Agreement, $\mathcal{F}_{\text{ABA}}$. We define the asynchronous Byzantine Agreement functionality $\mathcal{F}_{\text{ABA}}$ to be a special case of the asynchronous SFE functionality $\mathcal{F}_{\text{ASYNC}}$ introduced in Section 4.4. Here, the function $f^{\mathcal{F}_{\text{ABA}}}$ to evaluate is defined as follows: If the 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. Moreover, the tamper function $\mathsf{Tamper}_T^{\text{BA}}$ is according to Definition 6, introduced below.

In this case, the adversary's capabilities depends on the inputs of honest parties and the thresholds $T = (T_V, T_C, T_L)$, for validity, consistency and termination respectively. The tamper function is very similar to the Broadcast Tamper function $\mathsf{Tamper}_T^{\text{BC}}$. It guarantees that the adversary is not allowed to tamper the output value of the honest parties in the case he satisfies the validity threshold and the parties have preagreement on the inputs, or the consistency threshold is satisfied. It is also explicitly stated that in any case the adversary obtains the input values from the honest parties, since there is no privacy in Byzantine Agreement. Finally, the adversary can make the honest parties not obtain output if the termination threshold is not satisfied.

Definition 6. *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 $\mathsf{Tamper}_T^{\text{BA}}$:*

Function $\mathsf{Tamper}_T^{\text{BA}}(x_1, ..., x_n, \mathcal{H})$

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

- $c = 0$ *if and only if* $|\mathcal{P} \setminus \mathcal{H}| < T_V$ *and there exists* x *such that for all* $P_i \in \mathcal{H} : x_i = x$*, or* $|\mathcal{P} \setminus \mathcal{H}| < T_C$*.*
- $p = 1$*.*
- $d = 1$ *if and only if* $|\mathcal{P} \setminus \mathcal{H}| \geq T_L$*.*

We define the functionality $\mathcal{F}_{\text{ABA}}$ to be an asynchronous SFE functionality $\mathcal{F}_{\text{ASYNC}}$, parameterized by the tamper function $\mathsf{Tamper}_T^{\text{BA}}$ and evaluating the function $f^{\mathcal{F}_{\text{ABA}}}$, defined as follows: $f^{\mathcal{F}_{\text{ABA}}}(x'_1, ..., x'_n) = x$ if there exists x s.t. for every honest party $P_i \in \mathcal{I}$, $x'_i = x$. Otherwise, $f^{\mathcal{F}_{\text{ABA}}}(x'_1, ..., x'_n) = x'_j$ where j is the lowest index corresponding to a corrupted party.

Protocol to Increase Validity. We describe the protocol presented in [33] in our model. It constructs a binary asynchronous Byzantine Agreement functionality with validity, consistency and termination $(\frac{1}{2}(n - T_L), T_C, T_L)$, which we denote $\mathcal{F}_{\text{BABA-VAL}}$. It operates in the $(\mathcal{G}_{\text{CLK}}, \mathcal{F}_{\text{NET}}^{\delta}, \mathcal{F}_{\text{DSSKEYS}}, \mathcal{F}_{\text{ABA}})$-hybrid world, where $\mathcal{F}_{\text{ABA}}$ has parameters (T_V, T_C, T_L).

At a very high level, the protocol instructs each party P_i to sign its input and send the signature and the input to every party. Once P_i collects a list L of $n - T_L$ correct pairs of signature-input, it computes the majority bit b and sends L to every other party. Then, each party inputs the majority bit b to $\mathcal{F}_{\text{ABA}}$, and obtains as output a bit b^*. Each party terminates when it receives a list of $n - T_L$ pairs, where the majority of the signatures are on b^*. Let us argue why validity holds up to $\frac{1}{2}(n - T_L)$ corruptions. The

idea is that if less than $\frac{1}{2}(n - T_L)$ parties are corrupted and obtains a list L of $n - T_L$ correct pairs in Step 3, then at least $\frac{1}{2}(n - T_L)$ pairs are from honest parties. In the case that all honest parties have the same input b, it follows that the majority bit in L must be b (in fact, any list of size at least $n - T_L$, will contain b as the majority bit). Then, in Step 5 of the protocol, every party either terminates with output b upon receiving a valid list L on b, or does not terminate (in case it received $1 - b$ as output from $\mathcal{F}_{\text{aBA}}$ in the previous step).

Protocol $\Pi^{\texttt{val}}_{\texttt{aBA}}(P_i)$

Setup:

1: Input $(\text{GetKeys}, \text{sid})$ to $\mathcal{F}_{\text{DSSKeys}}$. Let the signing key be $\texttt{sk}$ and the corresponding verification key $\texttt{vk}$.

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

1: On input x_i, compute the signature $\sigma \leftarrow \mathsf{Sign}(x_i, \texttt{sk})$.
2: Send (x_i, σ) to every party. That is, input the message $(\text{Send}, i, j, (x_i, \sigma))$, for each P_j, to $\mathcal{F}_{\text{NET}}$.
3: Upon receiving $\ell \geq n - T_L$ valid messages of the form (x, σ) from $\mathcal{F}_{\text{NET}}$, compute the majority bit b among them. Let $L = (x_1, \sigma_1, \ldots, x_\ell, \sigma_l)$ be the list containing $\ell/2$ signatures on the majority bit b. For each j, input the message (Send, i, j, L) to $\mathcal{F}_{\text{NET}}$.
4: Input b to $\mathcal{F}_{\text{aBA}}$. Let b^* denote the output of $\mathcal{F}_{\text{aBA}}$.
5: Upon receiving a valid message L' with majority bit b^* from $\mathcal{F}_{\text{NET}}$, terminate with output b^*.

The following theorem is proven in Section B.

Theorem 2. *The protocol $\Pi^{\texttt{val}}_{\texttt{aBA}}$ operates in the $(\mathcal{G}_{\text{CLK}}, \mathcal{F}^{\delta}_{\text{NET}}, \mathcal{F}_{\text{DSSKeys}}, \mathcal{F}_{\text{aBA}})$-hybrid, where $\mathcal{F}_{\text{aBA}}$ gives output at most after $\tau_{\texttt{aba}}(\delta)$ clock ticks, and has validity, consistency and termination parameters (T_V, T_C, T_L), $T_L < \frac{n}{3}$. It realizes $\mathcal{F}_{\text{BABA-Val}}$ with validity, consistency and termination parameters $(\frac{1}{2}(1 - T_L), T_C, T_L)$. The maximum delay for the output is $\tau_{\texttt{val}} = \tau_{\texttt{aba}}(\delta) + 2\delta$.*

Protocol to Increase Consistency. In the following, we describe a protocol which operates in the $(\mathcal{G}_{\text{CLK}}, \mathcal{F}^{\delta}_{\text{NET}}, \mathcal{F}_{\text{DSSKeys}}, \mathcal{F}_{\text{aBA}})$, where $\mathcal{F}_{\text{aBA}}$ has parameters (T_V, T_C, T_L). It then realizes a binary asynchronous Byzantine Agreement functionality with parameters $(T_V, (n - 2T_L), T_L)$. In order to distinguish the assumed BA from the ideal BA, we denote the ideal BA functionality $\mathcal{F}_{\text{BABA-Con}}$.

The protocol is quite simple. First, each party P_i inputs x_i to $\mathcal{F}_{\text{aBA}}$, and once an output x is obtained from $\mathcal{F}_{\text{aBA}}$, it computes a signature $\sigma = \mathsf{Sign}(x, \texttt{sk})$ 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 the number of corruptions is smaller than $n - 2T_L$.

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

Setup:

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

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

1: On input x_i, input x_i to $\mathcal{F}_{\text{aBA}}$. Let x denote the output of $\mathcal{F}_{\text{aBA}}$.
2: Compute the signature $\sigma = \mathsf{Sign}(x, \texttt{sk})$.
3: Input $(\text{Send}, i, j, (x, \sigma))$, for each party P_j, to $\mathcal{F}_{\text{NET}}$, .
4: Upon receiving $\ell \geq n - T_L$ valid messages of the form (x', σ) from $\mathcal{F}_{\text{NET}}$, let $L = (x', \sigma_1, \ldots, \sigma_l)$ be the list containing these ℓ signatures on x'. Input (Send, i, j, L), for each party P_j, to $\mathcal{F}_{\text{NET}}$, and terminate with output x'.

The following theorem is proven in Section C.

Theorem 3. *The protocol $\Pi^{\texttt{con}}_{\texttt{aBA}}$ operates in the $(\mathcal{G}_{\text{CLK}}, \mathcal{F}^{\delta}_{\text{NET}}, \mathcal{F}_{\text{DSSKeys}}, \mathcal{F}_{\text{aBA}})$-hybrid, where $\mathcal{F}_{\text{aBA}}$ gives output at most after $\tau_{\texttt{aba}}(\delta)$ clock ticks, and it has validity, consistency and termination parameters*

(T_V, T_C, T_L), $T_L < \frac{n}{3}$. *It realizes* $\mathcal{F}_{\text{BABA-CON}}$ *with validity, consistency and termination parameters* $(T_V, n - 2T_L, T_L)$. *The maximum delay for the output is* $\tau_{\mathtt{con}} = \tau_{\mathtt{aba}}(\delta) + \delta$.

If we assume an asynchronous Byzantine Agreement $\mathcal{F}_{\text{ASYNC}}$ which runs concurrently in expected constant time as in [5], with Theorem 2 and Theorem 3, we obtain the following corollary:

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

5.2 SFE with increased Correctness and Privacy

In this section, we show how to realize $\mathcal{F}_{\text{ASYNC}}$ with correctness, privacy and termination parameters $(n - 2T_L, n - 2T_L, T_L)$, for any $T_L < \frac{n}{3}$. For that, we follow the ideas from [10, 27, 28], and replace the asynchronous Byzantine Agreement functionality with parameters $(\frac{n}{3}, \frac{n}{3}, \frac{n}{3})$, for the one that we obtained in Section 5.1, with parameters $(\frac{1}{2}(n - T_L), (n - 2T_L), T_L)$.

Let us first compare the guarantees that our protocol achieves with currently known protocols: at the cost of having termination when $t < T_L$, we obtain higher parameters for correctness and privacy than currently known protocols. As an example, if we set $T_L = \frac{1}{4}n$, one obtains correctness and privacy up to $T_C = T_P = \frac{1}{2}n$. Currently known protocols can only guarantee correctness and privacy up to $\frac{1}{3}n$ corruptions.

Moreover, the quality of the output is also improved: if the parties manage to agree on a common set of parties for which the input will be taken into account, the set has size at least $n - T_L$. In the range where it is guaranteed that the protocol terminates, $t < T_L$, we know that it also terminates correctly, privacy is preserved, and it takes into account at least $n - 2T_L$ inputs from honest parties. For $T_L = \frac{1}{4}n$, it therefore takes into account at least $\frac{n}{2}$ inputs from honest parties. By comparison, in current protocols, even if the adversary effectively corrupts less than $\frac{1}{4}n$ parties, the number of inputs from honest parties taken into account is only guaranteed to be $\frac{1}{3}n$.

Asynchronous SFE Protocol. The protocol operates in the hybrid model containing the functionalities $(\mathcal{G}_{\text{CLK}}, \mathcal{F}_{\text{NET}}^{\delta}, \mathcal{F}_{\text{SETUP}}^{\mathtt{FHE}}, \mathcal{F}_{\text{ABA}}, \mathcal{F}_{\text{ZK}})$. Here, $\mathcal{F}_{\text{SETUP}}^{\mathtt{FHE}}$ is the same functionality as the functionality $\mathcal{F}_{\text{SETUP}}$ presented in Section 4.1, where the threshold encryption scheme is fully-homomorphic. The functionality $\mathcal{F}_{\text{ZK}}$ is a zero-knowledge functionality, which allows a specific party P to prove to a party V knowledge of a witness w corresponding to a statement x. It is formally defined as follows:

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 $\tau_{\mathtt{zk}}$. It also stores the current time τ and keeps a buffer `buffer` of messages containing the proofs, 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 $R(x, w) = 1$, it creates a new identifier `id` and records the tuple $(\tau, \tau + 1, (x, w), \mathtt{id})$. It then sends $(x, \mathtt{id})$ to the adversary.
2: On input (GETPROOF, sid) from V, for each tuple $(T_{\mathtt{init}}, T_{\mathtt{end}}, (x, w), \mathtt{id})$ such that $T_{\mathtt{end}} \leq \tau$, remove it from `buffer` and output (x, sid) to V.
3: On input (DELAY, T, `id`) from the adversary, if there is a tuple $(T_{\mathtt{init}}, T_{\mathtt{end}}, (x, w), \mathtt{id})$ in `buffer` and $T_{\mathtt{end}} + T \leq T_{\mathtt{init}} + \tau_{\mathtt{zk}}$, then set $T_{\mathtt{end}} = T_{\mathtt{end}} + T$ and return (DELAY-OK) to the adversary. Otherwise, ignore the message.

Throughout the protocol, we consider two relations:

1. *Proof of Plaintext Knowledge:* The relation is parameterized by a threshold FHE scheme. The statement is an encryption key `ek` and a ciphertext c, and the witness is a plaintext m and randomness r such that $c = \mathtt{Enc}_{\mathtt{ek}}(m; r)$.

2. *Proof of Correct Decryption:* The relation is parameterized by a threshold FHE scheme. The statement consists of an encryption key $\mathtt{ek}$, a ciphertext c, a decryption share d, and the witness consists of a decryption key share $\mathtt{dk}_i$, such that $d = \mathtt{Dec}_{\mathtt{dk}_i}(c)$.

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

Input Stage. At a very high level, 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_L corruptions, we can take into account $n - T_L$ input providers.

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

After P_i collects $n - T_L$ certificates of distribution, it knows that at least $n - T_L$ parties have proved knowledge of the plaintext of their input ciphertext and also have distributed the ciphertext correctly to $n - T_L$ parties. Since $n - T_L > T_L$ for any $T_L < \frac{n}{2}$, this means that each of the $n - T_L$ 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 $n - T_L$ 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 $n - T_L$, 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 $n - T_L$ ones from the BAs before inputting any 0.

Protocol $\Pi^{\mathtt{input}}_{\mathtt{SFE}}(P_i)$

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

Setup:

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

Plaintext Knowledge and Distribution:

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

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

1: Send S_i to every party.
2: Once $n - T_L$ sets $\{S_j\}$ are collected, let $R = \bigcup_j S_j$ and enter n asynchronous BAs with inputs $v_1, \ldots, v_n \in \{0, 1\}$, 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 $n - T_L$ outputs which are one. Then, input 0 for the BAs which do not have input yet.
4: Let $w_1, \ldots, w_n$ be the outputs of the BAs.
5: Let $\mathtt{CoreSet} := \{j | w_j = 1\}$.
6: For each $j \in \mathtt{CoreSet}$ with $(j, (c_j, \pi_j)) \in R$, send $(j, (c_j, \pi_j))$ to all parties.

Computation and Threshold-Decryption Stage. At the end of the input stage, the parties have agreed on a common subset $\mathtt{CoreSet}$ of parties of size at least $n - T_L$, and each party holds the $n - T_L$

ciphertexts corresponding to the encryption of the input from each party in $\texttt{CoreSet}$. In the computation stage, the parties locally compute the homomorphic evaluation of the circuit, resulting on the ciphertext c encrypting the output.

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

Protocol $\Pi^{\texttt{comp}}_{\texttt{SFE}}(P_i)$

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

Function Evaluation:

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

Threshold Decryption:

1: Compute a decryption share $d_i = \texttt{Dec}_{\texttt{dk}_i}(c)$.
2: Prove, using $\mathcal{F}_{\text{ZK}}$, to each P_j that d_i is a correct decryption share of c.
3: Upon receiving a correct proof of decryption share for a ciphertext c' and decryption share d_j from P_j, send $\sigma_i^{\texttt{pocs}} = \mathsf{Sign}_{\texttt{sk}_i}((d_j, c'))$ to P_j.
4: Upon receiving $n - T_L$ signatures $\{\sigma_j^{\texttt{pocs}}\}$ on the same pair (d_i, c'), compute $\texttt{ProofShare}_i = \{\sigma_j^{\texttt{pocs}}\}$ and send $((d_i, c'), \texttt{ProofShare}_i)$ to all parties.
5: Upon receiving $n - T_L$ valid pairs $((d_j, c'), \texttt{ProofShare}_j)$ for the same c', compute the output $y_i = \texttt{Rec}(\{d_j\})$.

Termination Stage. The termination stage ensures that all honest parties terminate with the same output. This stage is essentially a Bracha broadcast 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 $n - 2T_L$ 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 correctness threshold $T_C = n - 2T_L$ 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 correctness threshold is not satisfied, the adversary can tamper the outputs, but so can the simulator.

Once $n - T_L$ votes on the same value y are collected, terminate with output y. The observation is that if a party receives $n - T_L$ votes on y, and termination should be guaranteed ($t < T_L$), there are $n - 2T_L$ honest parties that voted for y, and hence every honest party which did not have output will at some point collects $n - 2T_L$ 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 $n - T_L$ votes for y.

Protocol $\Pi^{\texttt{term}}_{\texttt{SFE}}(P_i)$

During the overall protocol, execute this protocol concurrently.

Waiting for Output:

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

Adopt Output:

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

Termination:

1: Wait until receiving $n - T_L$ votes for the same value y.
2: Terminate.

Let us denote $\Pi_{\texttt{SFE}}$ the protocol that executes concurrently the protocols $\Pi^{\texttt{input}}_{\texttt{SFE}}$, $\Pi^{\texttt{comp}}_{\texttt{SFE}}$ and $\Pi^{\texttt{term}}_{\texttt{SFE}}$. 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. The following theorem is proven in Section D.

Theorem 4. *The protocol $\Pi_{\mathtt{SFE}}$ operates in the $(\mathcal{G}_{\mathrm{CLK}}, \mathcal{F}_{\mathrm{NET}}^{\delta}, \mathcal{F}_{\mathrm{SETUP}}^{\mathtt{FHE}}, \mathcal{F}_{\mathrm{ABA}}, \mathcal{F}_{\mathrm{ZK}})$-hybrid, with the following parameters:*

- *$\mathcal{F}_{\mathrm{NET}}^{\delta}$ has unknown delay δ.*
- *$\mathcal{F}_{\mathrm{ABA}}$ gives output at most after $\tau_{\mathtt{aba}}(\delta)$ clock ticks. Moreover, it has validity, consistency and termination parameters $(\frac{1}{2}(n-T_L), n-2T_L, T_L)$, where the parameter $T_L \leq \frac{n}{3}$.*
- *$\mathcal{F}_{\mathrm{ZK}}$ gives output at most after $\tau_{\mathtt{zk}}(\delta)$ clock ticks.*

It realizes $\mathcal{F}_{\mathrm{ASYNC}}$ with validity, consistency and termination parameters $(n-2T_L, n-2T_L, T_L)$. The total maximum delay for the honest parties to obtain output is $\tau_{\mathtt{asynch}} = \tau_{\mathtt{aba}}(\delta) + 2\tau_{\mathtt{zk}}(\delta) + 9\delta$.

Given that $\mathcal{F}_{\mathrm{ZK}}$ can be UC realized in the $\mathcal{F}_{\mathrm{CRS}}$-hybrid model non-interactively [19], using Corollary 1 and Theorem 4 we can state the following theorem:

Theorem 5. *There exists a protocol which realizes $\mathcal{F}_{\mathrm{ASYNC}}$ with correctness, privacy and termination parameters $(n-2T_L, n-2T_L, T_L)$, for any $T_L < \frac{1}{3}$ in the $(\mathcal{G}_{\mathrm{CLK}}, \mathcal{F}_{\mathrm{NET}}^{\delta}, \mathcal{F}_{\mathrm{SETUP}}^{\mathtt{FHE}}, \mathcal{F}_{\mathrm{CRS}})$-hybrid world. The expected maximum delay for the output is $\tau_{\mathtt{asynch}} = O(\delta)$.*

6 Conclusions

In this section, we combine the main theorems of Section 4 and Section 5. We can instantiate $\mathcal{F}_{\mathrm{SYNC}}$ and $\mathcal{F}_{\mathrm{sBC}}$ with parameters $(\frac{n}{2}, \frac{n}{2}, n)$ using an honest majority MPC protocol such as [6, 23], and an honest majority broadcast protocol such as [37, 20]. We denote by $T_{\mathtt{sync}}(\Delta)$ and $T_{\mathtt{BC}}(\Delta)$ the delays for the outputs in the $\mathcal{F}_{\mathrm{SYNC}}$ and $\mathcal{F}_{\mathrm{sBC}}$ functionalities, respectively. With the above remarks and Theorem 1 and 5, we state formally the final theorem:

Theorem 6. *There exists a protocol parameterized by $\Delta \geq \delta$, which realizes $\mathcal{F}_{\mathrm{HYB}}$ on a function f, with correctness and privacy $T_C = T_P = \min\{\frac{n}{2}, n-2T_R\}$, for any $T_R \leq \frac{1}{3}n$, in the $(\mathcal{G}_{\mathrm{CLK}}, \mathcal{G}_{\mathrm{TIMEOUT}}, \mathcal{F}_{\mathrm{NET}}^{\delta}, \mathcal{F}_{\mathrm{SETUP}}^{\mathtt{FHE}}, \mathcal{F}_{\mathrm{CRS}})$-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 $\tau_{\mathtt{OD}} = T_{\mathtt{BC}}(\Delta) + \Delta$ if an output was delivered in the asynchronous phase, and otherwise is $\tau_{\mathtt{OND}} = T_{\mathtt{BC}}(\Delta) + T_{\mathtt{sync}}(\Delta)$.*

In particular, for $T_R = \frac{1}{4}n$, we obtain a protocol which realizes $\mathcal{F}_{\mathrm{HYB}}$ with correctness, privacy, responsiveness and termination parameters $(\frac{1}{2}n, \frac{1}{2}n, \frac{1}{4}n, n)$.

Acknowledgements. The authors would like to thank the anonymous reviewers for their comments which helped to improve the quality of this paper.

References

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

[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] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

[16] 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.

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

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

[19] 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.

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

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

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

[23] 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.

[24] 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.

[25] Yue Guo, Rafael Pass, and Elaine Shi. Synchronous, with a chance of partition tolerance. Cryptology ePrint Archive, Report 2019/179, 2019. `https://eprint.iacr.org/2019/179`.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[33] 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`.

[34] 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.

[35] 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.

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

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

[38] 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 Proof of the Protocol Compiler

In this section, we show the proof of the Theorem 1 of $\Pi^{\Delta}_{\mathtt{hyb}}$ from Section 4.

Theorem 1. *The protocol $\Pi^{\Delta}_{\mathtt{hyb}}$ operates in the $(\mathcal{G}_{\text{CLK}}, \mathcal{G}_{\text{TIMEOUT}}, \mathcal{F}^{\delta}_{\text{NET}}, \mathcal{F}_{\text{SETUP}}, \mathcal{F}_{\text{SYNC}}, \mathcal{F}_{\text{sBC}}, \mathcal{F}_{\text{ASYNC}})$-hybrid, with the following parameters:*

- *$\mathcal{F}^{\delta}_{\text{NET}}$ has unknown delay δ.*
- *$\mathcal{F}_{\text{SYNC}}$ evaluates function f and gives output after $T_{\mathtt{sync}}(\Delta)$ clock ticks. Moreover, it has parameters $(\frac{n}{2}, \frac{n}{2}, n)$ for correctness, privacy and termination.*
- *$\mathcal{F}_{\text{sBC}}$ gives output after $T_{\mathtt{BC}}(\Delta)$ clock ticks. Moreover, it has parameters $(\frac{n}{2}, \frac{n}{2}, n)$ for validity, consistency and termination.*
- *$\mathcal{F}_{\text{ASYNC}}$ evaluates function $f' = \mathtt{Enc}_{\mathtt{ek}}(f)$ and gives output after $T_{\mathtt{asynch}}(\delta)$ clock ticks. Moreover, it has parameters (T^a_C, T^a_P, T_L) for correctness, privacy and termination.*

For any $\Delta \geq \delta$, it realizes $\mathcal{F}_{\text{HYB}}$ with correctness, privacy, responsiveness and termination parameters $(\min(T^a_C, \frac{n}{2}), \min(T^a_P, \frac{n}{2}), T_L, n)$. The maximum delay of the asynchronous phase is $\tau_{\mathtt{asynch}} = T_{\mathtt{asynch}}(\delta) + 3\delta$, and of the synchronous phase is $\tau_{\mathtt{OD}} = T_{\mathtt{BC}}(\Delta) + \Delta$ if an output was delivered in the asynchronous phase, and otherwise is $\tau_{\mathtt{OND}} = T_{\mathtt{BC}}(\Delta) + T_{\mathtt{sync}}(\Delta)$.

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. We divide two cases, which depends on how the adversary delays the messages and at which time the timeout triggers:

1. An honest party obtains an output $[y_{\mathtt{asynch}}]$ from $\mathcal{F}_{\text{ASYNC}}$ and managed to collect a list L of $n - T_L$ signatures on this ciphertext during the asynchronous phase. In this case, we argue that every honest party outputs $y_{\mathtt{asynch}}$. Observe that there cannot be two lists L of size $n - T_L$ for different outputs, since $T_L < \frac{n}{3}$. Then, every honest party obtains the pair $([y_{\mathtt{asynch}}], L)$ as output of the synchronous BC in the synchronous phase. Hence, all parties send the decryption shares and every honest party who did not obtain the output in the asynchronous phase, will decrypt $[y_{\mathtt{asynch}}]$ and obtain the output $y_{\mathtt{asynch}}$.
2. No honest party obtains an output $[y_{\mathtt{asynch}}]$ from $\mathcal{F}_{\text{ASYNC}}$ and managed to collect a list L of $n - T_L$ signatures on this ciphertext during the asynchronous phase. Observe that in this case the adversary could have gathered a valid pair $([y_{\mathtt{asynch}}], L)$, but it was not able to learn anything about the output $y_{\mathtt{asynch}}$. This follows from the security of the threshold encryption scheme and the fact that no honest party sent any decryption key share.
   If the adversary broadcasts a valid pair, every honest party will decrypt $[y_{\mathtt{asynch}}]$ and obtain $y_{\mathtt{asynch}}$ as output. Otherwise, every honest party gives its input to $\mathcal{F}_{\text{SYNC}}$ and hence, every party obtains the same output $y_{\mathtt{sync}}$.

Moreover, one can easily see that if the timeout occurs after $\tau_{\mathtt{asynch}} = T_{\mathtt{asynch}}(\delta) + 3\delta$ clock ticks, then every honest party obtains output during the asynchronous phase. Let us analyze what happens if the timeout occurs before $\tau_{\mathtt{asynch}}$. In this case, we divide two cases: if an honest party obtained output, as argued above, every party will execute a broadcast round, and will send decryption shares. Hence, the honest parties obtain output after $\tau_{\mathtt{OD}} = T_{\mathtt{BC}}(\Delta) + \Delta$ clock ticks. Otherwise, the honest parties obtain output either with delay $\tau_{\mathtt{OD}}$ in case the adversary broadcasts a valid pair, or with delay $\tau_{\mathtt{OND}} = T_{\mathtt{BC}}(\Delta) + T_{\mathtt{sync}}(\Delta)$ if the adversary did not broadcast a valid pair.

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 τ accordingly.
2: Send (CHECKTIMEOUT, sid) to $\mathcal{G}_{\text{TIMEOUT}}$. If the response is (CHECKTIMEOUT, `true`, 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 `buffer` in $\mathcal{F}_{\text{NET}}$). More concretely, it does the following:

1: On input δ from $\mathcal{F}_{\text{HYB}}$, output δ to the adversary.
2: On input (FETCHMESSAGES, i) from P_i, for each message tuple $(T_{\texttt{init}}, T_{\texttt{end}}, P_k, P_i, m, \texttt{id}_m)$ from `buffer` where $T_{\texttt{end}} \leq \tau$, output (k, m) to P_i.
3: On input (DELAY, T, `id`) from the adversary, if there exists a tuple $(T_{\texttt{init}}, T_{\texttt{end}}, P_i, P_j, m, \texttt{id})$ in `buffer` and $T_{\texttt{end}} + T \leq T_{\texttt{init}} + \delta$, then set $T_{\texttt{end}} = T_{\texttt{end}} + T$ 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 $(\texttt{ek}, \texttt{dk}) \leftarrow \mathsf{Gen}_{(n-T_L,n)}(1^\kappa)$, 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 (GETKEYS, 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 $\mathcal{F}_{\text{HYB}}$. It keeps a variable τ_i for each party P_i.

// Internal emulation of $\mathcal{F}_{\text{ASYNC}}$.
1: On input (SETOUTPUTTIME, P_i, τ', sid) from the adversary, if $\tau' < \tau_{\texttt{asynch}}$, set $\tau_i = \tau'$.
2: Upon receiving an output y from $\mathcal{F}_{\text{HYB}}$, compute an encryption $[y]$ under the key `ek`. Output this encryption to each corrupted party P_i as soon as $\tau_i = 0$.
// Internal emulation of $\mathcal{F}_{\text{NET}}$.
3: As soon as $\tau_i = 0$ for an honest party P_i, input to `buffer`, the tuple $(\tau, \tau+1, i, j, [y], \mathsf{Sign}([y], \texttt{sk}_i)), \texttt{id})$, for each party P_j and freshly generated `id`. Output $(\text{SENT}, i, j, ([y], \mathsf{Sign}([y], \texttt{sk}_i))), \texttt{id})$ to the adversary.
4: As soon as the current time τ is such that there are $n - T_L$ tuples $(\tau_1, \tau_2, j, i, ([y], \mathsf{Sign}([y], \texttt{sk}_j)))$ such that $\tau_2 \leq \tau$ for the same i in `buffer`, input to `buffer` the tuple $(\tau, \tau+1, i, j, ([y], L'), \texttt{id})$, for each P_j, where L' contains the list of signatures. Input $(\tau, \tau+1, i, j, \texttt{dk}_i, \texttt{id})$ to `buffer`, for each party P_j. Also, output $(\text{SENT}, i, j, \texttt{dk}_i, \texttt{id})$ to the adversary.
5: Whenever a tuple $(i, (L, [y]))$ is delivered from `buffer` to a P_j, for each party P_i, input $(\tau, \tau+1, P_j, P_i, \texttt{dk}_j, \texttt{id})$ to `buffer` and output $(\text{SENT}, j, i, \texttt{dk}_j, \texttt{id})$ to the adversary.
// Delivery of honest parties' outputs.
6: On input (OUTPUT, i, sid) from $\mathcal{F}_{\text{HYB}}$, where P_i is an honest party, if $\tau \geq \tau_i$, and there are $n - T_L$ tuples $(\tau_1, \tau_2, j, i, \texttt{dk}_j, \texttt{id})$ with different j, input (DELIVEROUTPUT, i, sid) to $\mathcal{F}_{\text{HYB}}$.

Synchronous Phase:

// Internal emulation of $\mathcal{F}_{\text{SBC}}$.
1: 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 $T_{\texttt{BC}}(\Delta)$ clock ticks.
2: On input a valid pair $([y], L)$ from the adversary, input $(\tau, \tau+1, P_j, P_i, \texttt{dk}_j, \texttt{id})$ to `buffer`, for each honest party P_j to each corrupted party P_i to the adversary after $T_{\texttt{BC}}(\Delta)$ clock ticks.
// Internal emulation of $\mathcal{F}_{\text{SYNC}}$.
3: If no valid pair was received from the adversary, and no honest party received a valid pair in the asynchronous phase, do the following:
- On input x_i' from a corrupted party P_i, output x_i' to $\mathcal{F}_{\text{HYB}}$.
- On input the message (GETOUTPUT, `id`) from corrupted party P_i, forward the message to $\mathcal{F}_{\text{HYB}}$.
- On input a message (OUTPUT, y_i, `id`) where P_i is a corrupted party from $\mathcal{F}_{\text{HYB}}$, forward the message to the adversary.

Tamper Function:

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

3: On input $(\text{BLOCKOUTPUT}, P_i, \text{sid}; \mathcal{F}_{\text{ASYNC}})$ from the adversary, forward the input $(\text{BLOCKASYNCHOUTPUT}, P_i, \text{sid})$ to $\mathcal{F}_{\text{HYB}}$.
4: On input $(\text{BLOCKOUTPUT}, P_i, \text{sid}; \mathcal{F}_{\text{SYNC}})$ from the adversary, forward the input $(\text{BLOCKOUTPUT}, P_i, \text{sid})$ to $\mathcal{F}_{\text{HYB}}$.

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, lists of such signatures and decryption shares. All these messages can be simulated, since the simulator obtains the output and then encrypts and signs the output the same way as parties in the real world. 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 exactly the same in both worlds.
Internal emulation of $\mathcal{F}_{\text{ASYNC}}$. 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, as soon as the simulator obtains the output y, it computes $[y]$ and then delivers the ciphertext to P_i when $\tau_i = 0$ as well.
Internal emulation of $\mathcal{F}_{\text{NET}}$. In the real world, the corrupted parties obtain three types of messages after obtaining the ciphertext $[y]$: signatures on $[y]$, lists of signatures and decryption shares. Once an honest party obtains $[y]$ from the asynchronous functionality, it inputs to $\mathcal{F}_{\text{NET}}$ a signature of $[y]$ towards every party. Then, when $n - T_L$ signatures are collected, the honest party inputs the list and the decryption share 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 $[y]$ on behalf of each honest party P_i to `buffer`, towards every party (in particular, towards corrupted parties), and at the corresponding time. Once $n-T_L$ signatures are collected with destination P_i, the simulator emulates internally the protocol of P_i, and inputs to `buffer` the corresponding list, and also the decryption share $\texttt{dk}_i$, towards every party.
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 $n - T_L$ 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 $\mathcal{F}_{\text{sBC}}$. 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 $\mathcal{F}_{\text{sBC}}$ 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 $\mathcal{F}_{\text{SYNC}}$. After the round of synchronous broadcasts terminated, if a valid pair $([y], L)$ was received, then in the real world the honest parties send their decryption shares via $\mathcal{F}_{\text{NET}}$. Equivalently in the ideal world, the simulator inputs the decryption shares of each honest party to each corrupted party to `buffer`. Finally, if no valid pair was received, in the real world the parties execute $\mathcal{F}_{\text{SYNC}}$, whose behavior is directly emulated by the $\mathcal{F}_{\text{HYB}}$ functionality in the ideal world. That is, the simulator forwards the output from $\mathcal{F}_{\text{HYB}}$ 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.

Correctness. In the real world, if the adversary corrupts more than T_C^a parties, it can set the output of the asynchronous functionality $\mathcal{F}_{\text{ASYNC}}$ to any output y. In this case, the simulator will set the output

to an encryption of y. If the adversary corrupts more than $\frac{n}{2}$ parties, we also allow the simulator to set the output correspondingly. That is, the output of the parties will only be affected if there was not an output during the asynchronous phase and the Byzantine agreement protocol output $\perp$. In both cases, the adversary can set the output of $\mathcal{F}_{\text{HYB}}$ in the ideal world, since the correctness bound of $\mathcal{F}_{\text{HYB}}$ is $\min(T_C^a, \frac{n}{2})$.

Privacy. In the real world, if the adversary corrupts more than either T_P^a or $\frac{n}{2}$ parties, it can obtain the inputs from the honest parties. This is also the case in the ideal world, since the privacy bound of $\mathcal{F}_{\text{HYB}}$ is $\min(T_P^a, \frac{n}{2})$.

Termination. We remark that even if the termination bound T_L of $\mathcal{F}_{\text{ASYNC}}$ is violated, all the adversary can do in the real world is to prevent a party to obtain an output from $\mathcal{F}_{\text{ASYNC}}$. Hence, responsiveness is lost and the simulator will block the output from the asynchronous phase.

□

B Proof of the Protocol for ABA with Increased Validity

In this section, we show the proof of the Theorem 2 of $\Pi_{\texttt{aBA}}^{\texttt{val}}$ from Section 5.

Theorem 2. *The protocol $\Pi_{\texttt{aBA}}^{\texttt{val}}$ operates in the $(\mathcal{G}_{\text{CLK}}, \mathcal{F}_{\text{NET}}^{\delta}, \mathcal{F}_{\text{DSSKEYS}}, \mathcal{F}_{\text{ABA}})$-hybrid, with the following parameters:*

- *$\mathcal{F}_{\text{NET}}^{\delta}$ has unknown delay δ.*
- *$\mathcal{F}_{\text{ABA}}$ gives output at most after $\tau_{\texttt{aba}}(\delta)$ clock ticks. Moreover, it has validity, consistency and termination parameters (T_V, T_C, T_L), where $T_L < \frac{n}{3}$.*

It realizes $\mathcal{F}_{\text{BABA-VAL}}$ with validity, consistency and termination parameters $(\frac{1}{2}(1-T_L), T_C, T_L)$. The maximum delay for the output is $\tau_{\texttt{val}} = \tau_{\texttt{aba}}(\delta) + 2\delta$.

Proof. **Completeness.** Let us 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, every party will receive $n - T_L$ signatures on b, and hence b is the majority bit. By validity of the assumed $\mathcal{F}_{\text{ABA}}$, every party will output b. In the case that the parties do not hold the same input, after any party collects a list of $n - T_L$ signatures, it echoes the list to every other party. By the consistency of the assumed $\mathcal{F}_{\text{ABA}}$, every honest party terminates Step 5 with the same bit b'. Also, since the parties do not hold the same value, and we consider bits, at least one honest party P_j has input b'. This means that P_j collected a list of $n - T_L$ signatures with majority bit b'. Every other party will eventually obtain such list, and will also terminate with bit b'. In the ideal world, the simulator can choose this as the output bit.

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 $\mathcal{F}_{\text{BABA-VAL}}$, and $\mathcal{F}_{\text{ABA}}$ the functionality assumed in the real world.

At a very high level, the simulator simulates internally the messages that the real world functionalities $\mathcal{F}_{\text{NET}}$, $\mathcal{F}_{\text{SETUP}}$ and $\mathcal{F}_{\text{ABA}}$ 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. Moreover, it internally emulates the messages that the honest parties obtains.

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 $\mathcal{F}_{\text{ABA}}$, the simulator first computes the input to $\mathcal{F}_{\text{ABA}}$ (recall that it emulated the messages that each honest party obtained), and then internally emulates $\mathcal{F}_{\text{ABA}}$. That is, it 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 $\mathcal{F}_{\text{ABA}}$.

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

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 δ from $\mathcal{F}_{\text{BABA-VAL}}$, output δ 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 $(\text{DELAY}, T, \texttt{id})$ 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 $T_{\texttt{end}} + T \leq T_{\texttt{init}} + \delta$, then set $T_{\texttt{end}} = T_{\texttt{end}} + T$ 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 $(\texttt{vk}_j, \texttt{sk}_j) \leftarrow \mathsf{SigGen}(1^\kappa)$ for each party P_j. Then, it records the tuple $(\text{sid}, \texttt{vk}, \texttt{sk})$, 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, output $(\text{sid}, \texttt{vk}, \texttt{sk}_i)$ to P_i.

Main:

1: Upon receiving the input b_i from honest party P_i, input to $\texttt{buffer}$, on behalf of P_i, the tuple $(\tau, \tau+1, i, j, (b_i, \mathsf{Sign}(b_i, \texttt{sk}_i)), \texttt{id})$ for each corrupted party P_j and freshly generated $\texttt{id}$. Output $(\text{SENT}, i, j, (b_i, \mathsf{Sign}(b_i, \texttt{sk}_i)), \texttt{id})$ to the adversary.
2: Once there are $n - T_L$ tuples of the form $(\tau_1, \tau_2, j, i, (b_j, \mathsf{Sign}(b_j, \texttt{sk}_j)))$ that have been delivered from $\texttt{buffer}$ to a fixed party P_i ($\tau_2 \geq \tau$), compute the majority bit b among the signed bits. Then input, on behalf of honest party P_i, for each j, to $\texttt{buffer}$ the tuple $(\tau, \tau+1, i, j, L, \texttt{id})$, where L contains the list with a majority of signatures on the value b. Output $(\text{SENT}, i, j, L, \texttt{id})$ to the adversary. Set $x_i^{BA} = b$.
3: On input $(\text{NO-INPUT}, \mathcal{P}', \text{sid})$ from the adversary, forward the command to $\mathcal{F}_{\text{BABA-VAL}}$.
4: Once an output $(\text{OUTPUT}, b, \text{sid})$ is received from $\mathcal{F}_{\text{BABA-VAL}}$, output $(\text{OUTPUT}, b, \text{sid})$ to the adversary.
5: Define the output time τ_i of each honest party P_i. Set the output delay of P_i equal to D', where D' is the smallest number such that $(D', j, i, L, \texttt{id}) \in \texttt{buffer}$. Output $(\text{SETOUTPUTTIME}, P_i, \tau_i, \text{sid})$.
6: On input $(\text{SETOUTPUTTIME}, P_i, \tau', \text{sid})$ from the adversary, if $\tau' > \tau_i$, forward the command to $\mathcal{F}_{\text{BABA-VAL}}$.

Tamper Function:

1: On input $(\text{TAMPEROUTPUT}, P_i, y_i', \text{sid})$, where P_i is honest, from the adversary, forward the input to $\mathcal{F}_{\text{BABA-VAL}}$.
2: On input $(x_1, \ldots, x_n)$ from $\mathcal{F}_{\text{BABA-VAL}}$, output it to the adversary.
3: On input $(\text{BLOCKOUTPUT}, P_i, \text{sid})$, where P_i is honest, from the adversary, forward the input to $\mathcal{F}_{\text{BABA-VAL}}$.

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

First, let us remark that the real world is guaranteed to terminate when $|\mathcal{P} \setminus \mathcal{H}| < T_L$. This is because in this case every party will be able to collect a list of $n - T_L$ signatures, and $\mathcal{F}_{\text{ABA}}$ also terminates. In the ideal world, the simulator is allowed to block the outputs for the parties if $|\mathcal{P} \setminus \mathcal{H}| \geq T_L$.

Now, let us argue what happens if the validity threshold is satisfied, i.e. $|\mathcal{P} \setminus \mathcal{H}| < \frac{1}{2}(n - T_V)$, and every honest party holds the same bit b. In the real world any list containing $n - T_L$ signatures will contain this bit b as the majority bit. Hence, every party terminates with output b. In the ideal world, the validity of $\mathcal{F}_{\text{BABA-VAL}}$ guarantees that the output is b. If the consistency threshold holds, $|\mathcal{P} \setminus \mathcal{H}| < T_C$, then at least one honest party P_j has input b', corresponding to the output of the assumed $\mathcal{F}_{\text{ABA}}$. This means that P_j collected a list of $n - T_L$ signatures with majority bit b'. Every other party will also obtain such list, and will also terminate with bit b'. In the ideal world, the simulator can choose this as the output bit.

If none of the above conditions hold, then the output of the real world can be anything, depending on the adversary's strategy. Since the simulator can choose the output in this case, it can just compute the output based on the adversary's strategy.

Moreover, let us remark that in the real world, the parties only send messages in two steps via the network, and one via $\mathcal{F}_{\text{ABA}}$. 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 $\tau_{\texttt{aba}}(\delta)$ clock ticks, then the maximum delay for the

output is $\tau_{\mathtt{val}} = \tau_{\mathtt{aba}}(\delta) + 2\delta$. Hence, it is enough that the simulator has the power to delay the output up to $\tau_{\mathtt{val}}$ clock ticks.

□

C Proof of the Protocol for ABA with Increased Consistency

In this section, we show the proof of the Theorem 3 of $\Pi^{\mathtt{con}}_{\mathtt{aBA}}$ from Section 5.

Theorem 3. *The protocol $\Pi^{\mathtt{con}}_{\mathtt{aBA}}$ operates in the $(\mathcal{G}_{\text{CLK}}, \mathcal{F}^{\delta}_{\text{NET}}, \mathcal{F}_{\text{DSSKEYS}}, \mathcal{F}_{\text{ABA}})$-hybrid, with the following parameters:*

- *$\mathcal{F}^{\delta}_{\text{NET}}$ has unknown delay δ.*
- *$\mathcal{F}_{\text{ABA}}$ gives output at most after $\tau_{\mathtt{aba}}(\delta)$ clock ticks. Moreover, it has validity, consistency and termination parameters (T_V, T_C, T_L), where $T_L < \frac{n}{3}$.*

It realizes $\mathcal{F}_{\text{BABA-CON}}$ with validity, consistency and termination parameters $(T_V, n - 2T_L, T_L)$. The maximum delay for the output is $\tau_{\mathtt{con}} = \tau_{\mathtt{aba}}(\delta) + \delta$.

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, $\mathcal{F}_{\text{ABA}}$ 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 $\mathcal{F}_{\text{BABA-CON}}$, and $\mathcal{F}_{\text{ABA}}$ the functionality assumed in the real world.

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

1: On input (FETCHMESSAGES, i) from P_i, for each message tuple $(0, P_k, P_i, m, \mathtt{id}_m)$ in `buffer`, output (k, m) to P_i.
2: 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 `buffer` 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 $(\mathtt{vk}_j, \mathtt{sk}_j) \leftarrow \mathsf{SigGen}(1^{\kappa})$ 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 (GETKEYS, 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 $\mathcal{F}_{\text{baBA-Con}}$.
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 $\mathcal{F}_{\text{baBA-Con}}$.
3: On input $(\text{Output}, x, \text{sid})$ from $\mathcal{F}_{\text{baBA-Con}}$, output $(\text{Output}, x, \text{sid})$ to the adversary.
4: Keep a variable $D_{\mathcal{F}_{\text{aBA}},i} \coloneqq 0$ which denotes the delay of honest party P_i in $\mathcal{F}_{\text{aBA}}$. Moreover, on input $(\text{SetOutputTime}, P_i, D, \text{sid})$ from the adversary, set $D_{\mathcal{F}_{\text{aBA}},i} = D$, and also forward the input to $\mathcal{F}_{\text{baBA-Con}}$.
5: Once $D_{\mathcal{F}_{\text{aBA}},i} = 0$ and P_i is activated, input to $\texttt{buffer}$, 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 $\texttt{buffer}$ to a fixed honest party P_i, input, for each j, to $\texttt{buffer}$ 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: Input $(\text{Delay } \mathcal{F}_{\text{baBA-Con}}, P_i, \tau_2 + D_{\mathcal{F}_{\text{aBA}},i}, \text{sid})$, where τ_2 is the smallest number such that there is a tuple $(\tau_1, \tau_2, j, i, L, \mathtt{id}) \in \texttt{buffer}$ containing a list of at least $n - T_L$ signatures on a value x'. From now on, keep the output delay of P_i equal to $D_{\mathcal{F}_{\text{aBA}},i}$ plus the smallest number such that there is a tuple $(\tau_1, \tau_2, j, i, L, \mathtt{id}) \in \texttt{buffer}$.

Tamper Function:

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

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. $|\mathcal{P} \setminus \mathcal{H}| < T_V$ and the parties in the core-set have the same input, or the consistency threshold is satisfied, i.e. $|\mathcal{P} \setminus \mathcal{H}| < T_C$, then $\mathcal{F}_{\text{aBA}}$ ensures that the output at Step 1 is consistent among the honest parties. Let us denote this value x. In this case, if $|\mathcal{P} \setminus \mathcal{H}| < T_L$, 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 $|\mathcal{P} \setminus \mathcal{H}| \geq T_L$, 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 $|\mathcal{P} \setminus \mathcal{H}| < T_V$ where the parties in the core-set have the same input, nor the consistency threshold is satisfied, i.e. $|\mathcal{P} \setminus \mathcal{H}| \geq T_C$, then it is not guaranteed that the output after Step 1. (from $\mathcal{F}_{\text{aBA}}$) is consistent. However, we still need that if $|\mathcal{P} \setminus \mathcal{H}| < n - 2T_L$, 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 $|\mathcal{P} \setminus \mathcal{H}| < n - 2T_L$, 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 $\mathcal{F}_{\text{aBA}}$. 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.

□

D Proof of the Protocol for SFE with increased Correctness and Privacy

In this section, we show the proof of the Theorem 4 of $\Pi_{\mathtt{SFE}}$ from Section 5.

Theorem 4. *The protocol $\Pi_{\mathtt{SFE}}$ operates in the $(\mathcal{G}_{\text{CLK}}, \mathcal{F}_{\text{NET}}^{\delta}, \mathcal{F}_{\text{SETUP}}^{\mathtt{FHE}}, \mathcal{F}_{\text{aBA}}, \mathcal{F}_{\text{ZK}})$-hybrid, with the following parameters:*

– $\mathcal{F}_{\text{NET}}^{\delta}$ *has unknown delay* δ.
– $\mathcal{F}_{\text{ABA}}$ *gives output at most after* $\tau_{\texttt{aba}}(\delta)$ *clock ticks. Moreover, it has validity, consistency and termination parameters* $(\frac{1}{2}(n-T_L), n-2T_L, T_L)$*, where the parameter* $T_L \leq \frac{n}{3}$.
– $\mathcal{F}_{\text{ZK}}$ *gives output at most after* $\tau_{\texttt{zk}}(\delta)$ *clock ticks.*

It realizes $\mathcal{F}_{\text{ASYNC}}$ *with validity, consistency and termination parameters* $(n-2T_L, n-2T_L, T_L)$*. The total maximum delay for the honest parties to obtain output is* $\tau_{\texttt{asynch}} = \tau_{\texttt{aba}}(\delta) + 2\tau_{\texttt{zk}}(\delta) + 9\delta$.

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 $\mathcal{F}_{\text{ZK}}$ in two steps, and executes in parallel n BAs during the input provider selection. Hence, the protocol takes at most $\tau_{\texttt{aba}}(\delta) + 2\tau_{\texttt{zk}}(\delta) + 9\delta$ clock ticks to execute.

Soundness. We first describe the simulator.

Algorithm $\mathcal{S}_{\text{SFE}}$

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 `buffer` in $\mathcal{F}_{\text{NET}}$). More concretely, it does the following:

1: Receive δ from $\mathcal{F}_{\text{ASYNC}}$.
2: On input $(\text{FETCHMESSAGES}, i)$ from P_i, for each message tuple $(0, P_k, P_i, m, \texttt{id}_m)$ in `buffer`, output (k, m) to P_i.
3: On input $(\text{DELAY } \mathcal{F}_{\text{NET}}, T, \texttt{id})$ from the adversary, if there exists a tuple $(D, P_i, P_j, m, \texttt{id})$ in `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 $(\texttt{ek}, \texttt{dk}) \leftarrow \texttt{Gen}_{(n-T_L,n)}(1^{\kappa})$, where $\texttt{dk} = (\texttt{dk}_1, \ldots, \texttt{dk}_n)$.
2: On input $(\text{GETKEYS}, \text{sid})$ 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 $\mathcal{F}_{\text{ZK}}$. Then, when the adversary requests the output of P_i from $\mathcal{F}_{\text{ZK}}$ 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 `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 `buffer`.
5: As soon as there are $n - T_L$ 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 `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 `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 `buffer`, input a signature to `buffer`. That is, input $(\tau, \tau+1, i, j, \sigma_i^{\texttt{dist}}, \texttt{id})$ to `buffer`.
8: As soon as there are $n - T_L$ 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 `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 `buffer` each set S_i towards every party. That is, input $(\tau, \tau+1, i, j, S_i, \texttt{id})$ to `buffer`, for each P_j.
11: Once an emulated honest party P_i received $n - T_L$ 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 $n - T_L$ 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 $n - 2T_L$, the consistency of the BAs is not satisfied, 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 $n - T_L$ sets S_j, to `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 $(\text{No-Input}, \mathcal{P} \setminus \texttt{CoreSet}, \texttt{id})$ 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, $t \geq n - 2T_L$, 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 $CoreSet^i$ 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 $n - T_L$ signatures on the same pair (d_i, c'), it computes a proof that the decryption share d_i for c' is correct $\texttt{ProofShare}_i = \{\sigma_j^{\text{pocs}}\}$. It inputs to `buffer` the tuple $((d_i, c'), \texttt{ProofShare}_i)$ to every party.

7: When an honest party receives $n - T_L$ 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 $n - 2T_L$ 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 $n - T_L$ 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 $n - 2T_L$ 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 send a valid witness to $\mathcal{F}_{\text{ZK}}$. ■

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 $n - T_L$, any number corrupted decryption shares below $n - T_L$ 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 $n - T_L$ 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 send a valid witness to $\mathcal{F}_{\text{ZK}}$. ■

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.

□