Asynchronous Secure Multiparty Computation in Constant Time

Ran Cohen*

December 30, 2015

Abstract

In the setting of secure multiparty computation, a set of mutually distrusting parties wish to securely compute a joint function. It is well known that if the communication model is asynchronous, meaning that messages can be arbitrarily delayed by an unbounded (yet finite) amount of time, secure computation is feasible if and only if at least two-thirds of the parties are honest, as was shown by Ben-Or, Canetti, and Goldreich [STOC'93] and by Ben-Or, Kelmer, and Rabin [PODC'94]. The running-time of all currently known protocols depends on the function to evaluate. In this work we present the first asynchronous MPC protocol that runs in constant time.

Our starting point is the asynchronous MPC protocol of Hirt, Nielsen, and Przydatek [Eurocrypt'05, ICALP'08]. We integrate *threshold fully homomorphic encryption* in order to reduce the interactions between the parties, thus completely removing the need for the expensive *king-slaves* approach taken by Hirt et al.. Initially, assuming an honest majority, we construct a constant-time protocol in the asynchronous Byzantine agreement (ABA) hybrid model. Using a concurrent ABA protocol that runs in constant expected time, we obtain a constant expected time asynchronous MPC protocol, secure facing static malicious adversaries, assuming $t < n/3$.

Keywords: multiparty computation; asynchronous communication; threshold FHE; constant-time protocols; Byzantine agreement.

*Department of Computer Science, Bar-Ilan University. E-mail: cohenrb@cs.biu.ac.il.

Contents

1 Introduction

1.1 Background

In the setting of secure multiparty computation, a set of mutually distrusting parties wish to jointly and securely compute a function of their inputs. This computation should be such that each party receives its correct output, and none of the parties learn anything beyond their prescribed output. The standard definition today [14, 26] formalizes the above requirements (and others) in the following general way. Consider an ideal world in which an external trusted party is willing to help the parties carry out their computation. An ideal computation takes place in this ideal world by having the parties simply send their inputs to the trusted party, who then computes the desired function and passes each party its prescribed output. The security of a real protocol is established by comparing the outcome of the protocol to the outcome of an ideal computation. Specifically, a real protocol that is run by the parties is secure, if an adversary controlling a coalition of corrupted parties can do no more harm in a real execution than in the ideal execution.

One of the most important parameters for designing a protocol is the communication model. In the *synchronous* communication model, messages that are sent are guaranteed to be delivered within a *known* and finite time frame. As a result, the computation can proceed in *rounds*, such that if a party failed to receive a particular message in some round, within the expected time frame, the receiver knows that the sender did not transmit the message. Impressive feasibility results are known in this model [27, 8, 17, 38], stating that every functionality can be securely computed, assuming that a majority of the parties are honest. Furthermore, under suitable cryptographic assumptions, the computation can be done using constant-round protocols [4, 31, 2, 24, 28, 33].

The *asynchronous* model of communication is arguably more appropriate for modeling the real world. In this model the adversary has a stronger control over the communication channels and can impose an arbitrary unbounded (yet finite) delay on the arrival of each message. In particular, an honest party cannot distinguish between a corrupted party that refuses to send messages and an honest party whose messages are delayed.

This inherent limitation was taken into account by Ben-Or et al. [9] by adjusting the ideal-world computation. Since messages from t parties might never be delivered during the execution of the protocol, the trusted party cannot compute the function on *all* inputs. Therefore, the ideal-world adversary gets to decide on a *core set* of $n-t$ input providers (t of which might be corrupted) and the trusted party computes the function on their inputs (and default values for the rest). Next, the trusted party sends to each party the output of the computation along with the identities of the parties in the core set. It immediately follows that a secure protocol implies agreement in the asynchronous setting, since the core set must be agreed upon as part of the protocol, and therefore is feasible in the standard model if and only if $t < n/3$ [9, 10]. Asynchronous protocols that are secure assuming $t < n/2$ are only known in weaker models that assume either a synchronous broadcast round [6] or some form of non-equivocation [3]. Moreover, the running-time[1] of all currently known asynchronous protocols depends on the function to be computed and no constant-time protocols were known.

In this work we study the following question.

Do there exist asynchronous secure multiparty protocols which run in constant time?

[1] The running time is measured by the elapsed time of the protocol while normalizing the maximal delay imposed on a message to 1.

1.2 Our Result

Our main result is a feasibility result of an asynchronous secure multiparty protocol that runs in constant time in a hybrid model where the parties have access to an ideal *asynchronous Byzantine agreement (ABA)* functionality.

The main tools that we use are *threshold fully homomorphic encryption* (TFHE) and *threshold signatures* (TSIG). A fully homomorphic encryption scheme (FHE) is an encryption scheme that enables an evaluation of a function over a tuple of ciphertexts to obtain an encrypted result. TFHE is essentially a distributed version of FHE, where the decryption key is secret shared amongst the parties. In order to decrypt a ciphertext, each party locally uses its share of the decryption key and computes a share of the plaintext. The plaintext can then be reconstructed given $t+1$ decryption shares. Similarly, in a threshold signature scheme, the signing key is secret shared and $t+1$ shares are required in order to sign a message. We note that both of these computational assumption can be based on the standard *learning with errors* (LWE) problem, see Asharov et al. [2], Bendlin and Damgård [11] and Bendlin et al. [12].

Theorem 1.1 (informal)**.** *Assume that TFHE and TSIG schemes exist, and that the cryptographic keys have been pre-distributed. Then any efficiently computable function f can be securely computed in the asynchronous setting facing static malicious adversaries, assuming an honest majority and given access to an ABA ideal functionality. The time complexity of the protocol is $O(1)$, the communication complexity is independent of the multiplication-depth of the circuit representing f and the number of (concurrent) invocations of the ABA ideal functionality is n.*

Using the concurrent ABA protocol of Ben-Or and El-Yaniv [7], which runs in constant expected time[2] and is resilient for $t < n/3$, we obtain the following corollary.

Corollary 1.2 (informal)**.** *Assume that TFHE and TSIG schemes exist, then any function can be securely computed in the asynchronous setting using a constant expected time protocol, in the presence of static malicious adversaries, for $t < n/3$.*

1.3 Overview of the Protocol

The basis of our technique is the protocol of Cramer et al. [20] (designed for the *synchronous* setting), which is based on *threshold additively homomorphic encryption* (TAHE)[3] and is designed in a hybrid model where the encryption keys are pre-distributed before the protocol begins. Initially, each party encrypts its input and broadcasts the ciphertext. Next, the circuit is homomorphically evaluated, where addition gates are computed locally and multiplication gates are computed interactively. Finally, a threshold decryption protocol is executed, and the parties learn the output.

Hirt et al. [29, 30] adopted the protocol of [20] into the asynchronous setting by introducing the *king-slaves paradigm*. Initially, each party sends its encrypted input to all the parties, and the core set is decided upon using an *agreement on a common subset* (ACS) protocol, which incorporates n instances of ABA. Next, n copies of the circuit are interactively evaluated. In each evaluation one of the parties acts as king while all other parties act as slaves. The role of the slaves is to help the king with the computation of multiplication gates. At the end of each such evaluation, the slaves send their decryption shares to the king which recovers the output. The evaluations of the circuit

[2] Following the lower bound of [22], asynchronous agreement protocols cannot be computed in constant time.

[3] Which essentially means that ciphertexts can be added but not multiplied.

are executed asynchronously, i.e., one king may finish its computation while another king hasn't started yet, therefore each party must hold a state for each evaluation of the circuit.

The time complexity of the protocols of Hirt et al. [29, 30] depends on the depth of the circuit to compute. In this work, we use a TFHE instead of TAHE in order to reduce the running time. This adjustment not only yields better time complexity and better communication complexity, but also enables a design *without* the expensive king-slave paradigm, since each party can locally and non-interactively evaluate the entire circuit. As a consequence, the description of the new protocol is greatly simplified, and also results with a better memory complexity compared to [29, 30], since the parties do not need to store a local state for each of the n evaluations of the circuit.

Our protocol consists of three stages. The *input stage*, in which the core set of input providers is determined, follows in the lines of Hirt et al. [29, 30]. In the *computation and threshold decryption stage*, each party homomorphically evaluates the circuit non-interactively and obtains an encrypted output $\tilde{c}$. Next, the party uses its share of the decryption key to compute a decryption share and send it to all other parties. Once a party receives $t+1$ valid decryption shares it can recover the output. During these stages, the validity of each message sent by some party must be proven. This is done by running a sub-protocol which produces a *certificate* for the message (which is essentially a signature produced by $n-t$ parties). Therefore, a party must remain active and assist in constructions of certificates even *after* it obtained its output. The *termination stage* ensures a safe termination of all the parties and follows Bracha [13]. Once a party obtained its output it sends it to all other parties. When a party receives $t+1$ consisting values it can safely set its output to this value (even if it did not complete the computation and threshold decryption stage) and once receiving outputs from $n-t$ parties, terminate.

1.4 Additional Related Work

Ben-Or et al. [9] were the first to define asynchronous secure multiparty computation. They constructed a BGW-alike [8] asynchronous protocol that is secure in the presence of malicious adversaries when $t < n/4$; the authors showed that this threshold is tight when considering *perfect* correctness. Ben-Or et al. [10] constructed a protocol with *statistical* correctness that is secure in the presence of malicious adversaries, for $t < n/3$. This threshold is also tight following the lower bound of Toueg [41], stating that asynchronous Byzantine agreement is impossible if $t \geq n/3$, even in the PKI model.

Following the feasibility results of [9, 10] great improvements have been made regarding the communication complexity. Two main approaches have been used, the first is in the information-theoretic model and does not rely on cryptographic assumptions [40, 37, 5, 35, 36, 19] while the second is in the computational model and is based on threshold additively homomorphic encryption, these protocols appear in [29, 30, 18] and rely on a preprocessing phase for key distribution.

In order to achieve security for an honest majority, the model must be weakened in some sense. Beerliová-Trubíniová et al. [6] allowed a limited usage of synchronous Byzantine agreement and adjusted the protocol from [30] to the case where $t < n/2$. Backes et al. [3] augmented the model with a non-equivocation oracle, and constructed a protocol that is secure assuming an honest majority.

In an independent work, Choudhury and Patra [18] suggested using TFHE in order to reduce the time complexity, but did not proceed in this route since they considered concrete efficiency. We note that in this work we focus on feasibility results rather than concrete efficiency of the protocols.

A comparison of the asynchronous MPC protocols appears in Table 1.

Paper	Resilience	Correctness	Time[a]	Communication[b]	Assumptions[c]	Hybrid Model[d]
[9]	$t < n/4$	Perfect	$O(c_M)$	$O(c_M \cdot n^6)$		
[10]	$t < n/3$	Statistical	$O(c_M)$	$\Omega(c_M \cdot n^{11})$		
[40]	$t < n/4$	Perfect	$O(c_M)$	$\Omega(c_M \cdot n^5)$		
[37]	$t < n/4$	Statistical	$O(c_M)$	$O(c_M \cdot n^4 + n^5)$		
[29]	$t < n/3$	Computational	$O(c_M)$	$O(c_M \cdot n^3\kappa)$	TAHE, TSIG	KeyDist
[5]	$t < n/4$	Perfect	$O(c_M)$	$O(c_M \cdot n^3)$		
[30]	$t < n/3$	Computational	$O(c_M)$	$O(c_M \cdot n^2\kappa + n^3\kappa)$	TAHE, TSIG	KeyDist
[35]	$t < n/3$	Statistical	$O(c_M)$	$O(c_M \cdot n^5)$		
[36]	$t < n/4$	Statistical	$O(c_M)$	$O(c_M \cdot n^2 + n^4)$		
[36]	$t < n/4$	Perfect	$O(c_M)$	$O(c_M \cdot n^2 + n^3)$		
[6]	$t < n/2$	Computational	$O(c_M)$	$O(c_M \cdot n^4\kappa)$	TAHE, TSIG	KeyDist, Bcast
[19]	$t < n/4$	Statistical	$O(c_M)$	$O(c_M \cdot n + n^3)$		
[3]	$t < n/2$	Computational	$O(c_M)$	$O(c_M \cdot n^3\kappa)$	AHE, TSIG	KeyDist, NEQ
[3]	$t < n/2$	Computational	$O(c_M)$	$O(c_M \cdot n^2\kappa + n^3\kappa)$	TAHE, TSIG	KeyDist, NEQ
[18]	$t < n/3$	Computational	$O(c_M)$	$O(c_M \cdot n\kappa + n^3\kappa)$	TSHE	KeyDist
This work	$t < n/3$	Computational	$O(1)$	$O(n^3\kappa)$	TFHE, TSIG	KeyDist

[a] Time complexity is measured in the ABA-hybrid model.

[b] c_M denotes the number of multiplication gates. Input, output and addition gates are ignored.

[c] TSIG is a threshold digital signature scheme, AHE is an additively homomorphic encryption scheme, TAHE is a threshold additively homomorphic encryption scheme, TSHE is a threshold somewhat homomorphic encryption scheme, TFHE is a threshold fully homomorphic encryption scheme.

[d] KeyDist stands for key distribution for a threshold cryptosystem, NEQ stands for transferable non-equivocation mechanism, Bcast stands for *synchronous* broadcast.

Table 1: Comparison of asynchronous MPC protocols.

1.5 Open Questions

Our main result establishes a feasibility result of asynchronous secure computation in constant time, however it requires quite strong assumptions, e.g., threshold FHE and a preprocessing phase for distributing the cryptographic keys. An interesting open question is to construct constant-time asynchronous protocols under weaker assumptions.

Paper Organization

The cryptographic primitives are defined in Section 2, followed by an overview of the UC framework in Section 3. Certificates are defined in Section 4 and Section 5 presents our asynchronous MPC protocol. The security proof is given in Section 6 and the conclusions appear in Section 7.

2 Preliminaries

In this section we present the definitions of the cryptographic schemes that are used in our protocol.

2.1 Threshold Fully Homomorphic Encryption

Definition 2.1. *A* homomorphic encryption (HE) scheme *consists of 4 PPT algorithms:*

- **Key generation:** $(dk, ek) \leftarrow \text{Gen}(1^\kappa)$*; outputs a pair of keys: the secret decryption key* dk *and the public encryption (and evaluation) key* ek*.*

- **Encryption:** $c \leftarrow \mathrm{Enc}_{ek}(m)$*; using* ek*, encrypt a plaintext* m *into a ciphertext* c*.*
- **Decryption:** $m = \mathrm{Dec}_{dk}(c)$*; using* dk*, decrypt the ciphertext* c *to into a plaintext* m*.*
- **Homomorphic evaluation:** $c = \mathrm{Eval}_{ek}(C, c_1, \ldots, c_\ell)$*; using* ek*, evaluate a circuit* C *over a tuple of ciphertexts* $(c_1, \ldots, c_\ell)$ *to produce a ciphertext* c*.*

We say that a HE scheme is correct for circuits in a circuit class $\mathcal{C}$ if for every $C \in \mathcal{C}$ and every series of inputs $m_1, \ldots, m_\ell \in \{0,1\}^*$ it holds that

$$\Pr\left[\mathrm{Dec}_{dk}\left(\mathrm{Eval}_{ek}\left(C, \mathrm{Enc}_{ek}(m_1), \ldots, \mathrm{Enc}_{ek}(m_\ell)\right)\right) \neq C\left(m_1, \ldots, m_\ell\right)\right] \leq \mathrm{negl}(\kappa).$$

Semantic security of HE schemes is defined in the standard way, see [25].

Definition 2.2. *A family of HE schemes* $\{\Pi^{(d)} = (\mathrm{Gen}^{(d)}, \mathrm{Enc}^{(d)}, \mathrm{Dec}, \mathrm{Eval}^{(d)}) \mid d \in \mathbb{N}^+\}$ *is* leveled fully homomorphic *if for every* $d \in \mathbb{N}^+$*, the following holds:*

- ***Correctness:*** $\Pi^{(d)}$ *correctly evaluates the set of all boolean circuits of depth at most* d*.*
- ***Compactness:*** *There exists a polynomial* s *such that the common decryption algorithm can be expressed as a circuit of size at most* $s(\kappa)$ *and is independent of* d*.*

In our protocol for computing a function f, the depth d of the circuit C representing f is known in advance. We remove the notation (d) from the schemes throughout the paper for clarity. We also require the FHE scheme to have a threshold decryption, informally this means that Gen generates the public key ek as well as a t_e-secret sharing of the secret key $(dk_1, \ldots, dk_n)$, such that decrypting c using dk_i produces a share m_i of the plaintext m. We will use $t_e = t + 1$.

Definition 2.3. *A* threshold homomorphic encryption scheme *is a homomorphic encryption scheme augmented with the following properties:*

- *The key generation algorithm is parameterized by* (t_e, n) *and outputs* $(dk, ek) \leftarrow \mathrm{Gen}_{(t_e,n)}(1^\kappa)$*, where* dk *is represented using a* (t_e, n)*-threshold secret sharing of the secret key* $(dk_1, \ldots, dk_n)$*.*
- *Given a ciphertext* c *and a share of the secret key* dk_i*, the share-decryption algorithm outputs* $d_i = \mathrm{DecShare}_{dk_i}(c)$ *such that* $(d_1, \ldots, d_n)$ *forms a* (t_e, n)*-threshold secret sharing of the plaintext* $m = \mathrm{Dec}_{dk}(c)$*. We denote the reconstruction algorithm that receives* t_e *decryption shares* $\{d_i\}$ *by* $m = \mathrm{DecRecon}(\{d_i\})$*.*

2.2 Threshold Signatures

A threshold signature scheme is a signature scheme in which the signing key is shared amongst n parties using a t_s-threshold secret-sharing scheme. Using t_s shares of the signing key it is possible to sign on any message, however using less than t_s shares it is infeasible to forge a signature. We will use $t_s = n - t$.

Definition 2.4 (Threshold Signature Scheme)**.** *A* threshold signature scheme *is a signature scheme* $(\mathrm{SigGen}, \mathrm{Sign}, \mathrm{Vrfy})$ *augmented with the following properties*

- *The signature key generation algorithm is parameterized by* (t_s, n) *and outputs* $(sk, vk) \leftarrow \mathrm{SigGen}_{(t_s,n)}(1^\kappa)$*, where* sk *is represented using a* (t_s, n)*-threshold secret sharing of the secret signing key* $(sk_1, \ldots, sk_n)$*.*

- *Given a plaintext m and a share of the secret key sk_i, the share-signing algorithm outputs $\sigma_i \leftarrow \text{SignShare}_{sk_i}(m)$ such that $(\sigma_1, \ldots, \sigma_n)$ forms a (t_s, n)-threshold secret sharing of the signature $\sigma \leftarrow \text{Sign}_{sk}(m)$.*

For a security definition of threshold signatures see, for example, [1].

3 The Security Model

3.1 The UC Framework

In this section we present a high-level description of the security model. We follow the UC framework of Canetti [14], which is based on the *real/ideal paradigm*, i.e., comparing what an adversary can do in the real execution of the protocol to what it can do in an ideal model where an uncorrupted trusted party (an ideal functionality) assists the parties. Informally, a protocol is secure if whatever an adversary can do in the real protocol (where no trusted party exists) can be done in the ideal computation.

The real world. An execution of a protocol π in the real model consists of n *interactive Turing machines* (ITMs) $P_1, \ldots, P_n$ representing the parties, along with two additional ITMs, an *adversary* $\mathcal{A}$, describing the behavior of the corrupted parties and an *environment* $\mathcal{Z}$, representing the external environment in which the protocol operates. The environment gives inputs to the honest parties, receives their outputs, and can communicate with the adversary at any point during the execution. The adversary controls the operations of the corrupted parties and the delivery of messages between the parties.

In more details, each ITM is initialized with the security parameter κ and random coins, where the environment may receive an additional auxiliary input. We consider *static* corruptions, meaning that the set of corrupted parties is fixed before the protocol begins and is known to $\mathcal{A}$ and $\mathcal{Z}$. The protocol proceeds by a sequence of *activations*, where the environment is activated first and at each point a single ITM is active. The environment can either activate one of the parties with input or activate the adversary by sending it a message. Once a party is activated it can perform a local computation, write on its output tape or send messages to other parties. After the party completes its operations the control is returned to the environment. Once the adversary is activated it can send messages on behalf of the corrupted parties or send a message to the environment. In addition, $\mathcal{A}$ controls the communication between the parties, and so it can read the content of the messages sent between the parties and is responsible for delivering each message to its recipient. Once $\mathcal{A}$ delivers a message to some party, this party is activated. We assume that the adversary cannot omit, change or inject messages, however it can decide *which* message will be delivered and *when*.[4] The protocol completes once $\mathcal{Z}$ stops activating other parties and outputs a single bit.

If the adversary is fail-stop, it always instructs the corrupted parties to follow the protocol, with the exception that they can halt prematurely and stop sending messages. If the adversary is malicious, it may instruct the corrupted parties to deviate from the protocol arbitrarily.

Let $\text{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(\kappa, z, \vec{r})$ denote $\mathcal{Z}$'s output on input z and security parameter κ, after interacting with adversary $\mathcal{A}$ and parties $P_1, \ldots, P_n$ running protocol π with random tapes

[4] This behaviour is formally modeled using the *eventual-delivery secure message transmission* ideal functionality in [32].

$\vec{r} = (r_1, \ldots, r_n, r_{\mathcal{A}}, r_{\mathcal{Z}})$ as described above. Let $\text{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(\kappa, z)$ denote the random variable $\text{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(\kappa, z, \vec{r})$, when the vector $\vec{r}$ is uniformly chosen.

The ideal model. A computation in the ideal model consists of n *dummy* parties $P_1, \ldots, P_n$, an *ideal adversary* (simulator) $\mathcal{S}$, an *environment* $\mathcal{Z}$, and an *ideal functionality* $\mathcal{F}$. The environment gives inputs to the honest (dummy) parties, receives their outputs, and can communicate with the ideal adversary at any point during the execution. The dummy parties act as channels between the environment and the ideal functionality, meaning that they send the inputs received from $\mathcal{Z}$ to $\mathcal{F}$, and transfer the output they receive from $\mathcal{F}$ to $\mathcal{Z}$. We consider static corruptions, and so the set of corrupted parties is fixed before the computations, and is known to $\mathcal{Z}$, $\mathcal{S}$ and $\mathcal{F}$. As before, the computation completes once $\mathcal{Z}$ stops activating other parties and outputs a single bit.

The ideal functionality defines the desired behaviour of the computation. $\mathcal{F}$ receives the inputs from the dummy parties, executes the desired computation and sends the output to the parties. The ideal adversary does not see and cannot delay the communication between the parties and the ideal functionality, however, $\mathcal{S}$ can communicate with $\mathcal{F}$. As we consider asynchronous protocols in the real model, ideal functionalities must consider some inherent limitations, for instance, the ability of the adversary to decide when each honest party learns the output. Since the UC framework has no notion of time, we follow [34, 32] and model time by number of activations. Once $\mathcal{F}$ prepares an output for some party it does not ask permission from the adversary to deliver it to the party, instead the party must request the functionality for the output, and this can only be done when the party is active. Furthermore, the adversary can instruct $\mathcal{F}$ to delay the output for each party by ignoring the requests for a polynomial number of activations. If the environment activates the party sufficiently many times, the party will eventually receive the output from the ideal functionality. It follows that the ideal computation will terminate, i.e., all honest parties will obtain their output, in case the environment will allocate enough resources to the parties. We use the term $\mathcal{F}$ *sends a request-based delayed output to* P_i to describe the above interaction between the $\mathcal{F}$, $\mathcal{S}$ and P_i.

Let $\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\kappa, z, \vec{r})$ denote $\mathcal{Z}$'s output on input z and security parameter κ, after interacting with ideal adversary $\mathcal{S}$ and dummy parties $P_1, \ldots, P_n$ which interact with ideal functionality $\mathcal{F}$ with random tapes $\vec{r} = (r_{\mathcal{S}}, r_{\mathcal{Z}})$ as described above. Let $\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\kappa, z)$ denote the random variable $\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\kappa, z, \vec{r})$, when the vector $\vec{r}$ is uniformly chosen.

Definition 3.1. *We say that a protocol π t-securely UC realizes an ideal functionality $\mathcal{F}$ in the presence of static malicious (resp., fail-stop) adversaries, if for any PPT malicious (resp., fail-stop) real model adversary $\mathcal{A}$, controlling a subset of up to t parties, and any PPT environment $\mathcal{Z}$, there exists a PPT ideal model adversary $\mathcal{S}$ such that following two distribution ensembles are computationally indistinguishable*

$$\left\{\text{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(\kappa, z)\right\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*} \overset{\text{c}}{\equiv} \left\{\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\kappa, z)\right\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}.$$

The hybrid model. In a $\mathcal{G}$-hybrid model, the execution of the protocol proceeds as in the real model, however, the parties have access to an ideal functionality $\mathcal{G}$ for some specific operations. The communication of the parties with the ideal functionality $\mathcal{G}$ is performed as in the ideal model. An important property of the UC framework is that an ideal functionality in a hybrid model can be replaced with a protocol that securely UC realizes $\mathcal{G}$. We informally state the composition theorem from Canetti [14].

Theorem 3.2 ([14]). *Let π be a protocol that t-securely UC realizes $\mathcal{F}$ in the $\mathcal{G}$-hybrid model and let ρ be a protocol that t-securely UC realizes $\mathcal{G}$. Then the protocol π^ρ that is obtained from π by replacing every ideal call to $\mathcal{G}$ with the protocol ρ, t-securely UC realizes $\mathcal{F}$ in the model without ideal functionality $\mathcal{G}$.*

3.2 Some Ideal Functionalities

We now present the asynchronous SFE and asynchronous BA functionalities.

3.2.1 Asynchronous Secure Function Evaluation

Secure function evaluation (SFE) is a multiparty primitive where a set of n parties wish to compute a (possibly randomized) function $f\colon (\{0,1\}^*)^n \times \{0,1\}^* \to (\{0,1\}^*)^n$, where $f = (f_1, \ldots, f_n)$. That is, for a vector of inputs $\boldsymbol{x} = (x_1, \ldots, x_n) \in (\{0,1\}^*)^n$ and random coins $r \in_R \{0,1\}^*$, the output vector is $(f_1(\boldsymbol{x}; r), \ldots, f_n(\boldsymbol{x}; r))$. The output for the i'th party (with input x_i) is defined to be $f_i(\boldsymbol{x}; r)$. The function f has public output, if all parties output the same value, i.e., $f_1 = \ldots = f_n$, otherwise f has private output.

In an asynchronous protocol for computing secure function evaluation, the adversary can always delay messages from t parties, and so t input values might not take part in the computation. Therefore, in the definition of the ideal functionality for asynchronous SFE, the ideal-model adversary is given the power to determine a *core set* of $n - t$ input providers (t of which might be corrupted) that will contribute input values for the computation. The asynchronous secure function evaluation functionality, $\mathcal{F}^f_{\mathsf{ASFE}}$, is presented in Figure 1.

Functionality $\mathcal{F}^f_{\mathsf{ASFE}}$

$\mathcal{F}^f_{\mathsf{ASFE}}$ proceeds as follows, running with parties $P_1, \ldots, P_n$ and an adversary $\mathcal{S}$, and parameterized by an n-party function $f\colon (\{0,1\}^*)^n \times \{0,1\}^* \to (\{0,1\}^*)^n$. For each party P_i initialize an input value $x_i = \perp$ an output value $y_i = \perp$.

- Upon receiving a message $(\mathsf{input}, \mathsf{sid}, v)$ from some party P_i, if CoreSet has not been recorded yet or if $P_i \in \mathsf{CoreSet}$, set $x_i = v$. Next, send a message $(\mathsf{input}, \mathsf{sid}, P_i)$ to $\mathcal{S}$.
- Upon receiving a message $(\mathsf{coreset}, \mathsf{sid}, \mathsf{CoreSet})$ from $\mathcal{S}$, verify that CoreSet is a subset of $\{P_1, \ldots, P_n\}$ of size $n - t$; else ignore the message. If CoreSet has not been recorded yet, record CoreSet and for every P_i not in CoreSet, set x_i to some default input value $x_i = \tilde{x}_i$.
- Upon receiving a message $(\mathsf{output}, \mathsf{sid})$ from some party P_i, do:
  1. If CoreSet has not been recorded yet or if x_j has not been recorded for some $P_j \in \mathsf{CoreSet}$, ignore the message.
  2. Otherwise, if $y_1, \ldots, y_n$ have not been set yet, then choose $r \in_R \{0,1\}^*$ and compute $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n; r)$.
  3. Generate a request-based delayed output $(\mathsf{output}, \mathsf{sid}, (\mathsf{CoreSet}, y_i))$ to P_i and send $(\mathsf{output}, \mathsf{sid}, P_i)$ to $\mathcal{S}$.

Figure 1: The asynchronous secure function evaluation functionality

3.2.2 Asynchronous Byzantine Agreement

In a *synchronous* Byzantine agreement, each party has an input bit and outputs a bit. Three properties are required: *agreement*, meaning that all honest parties agree on the same bit, *validity*, meaning that if all honest parties have the same input bit then this will be the common output and *termination*, meaning that the protocol eventually terminates. When considering *asynchronous* Byzantine agreement (ABA), the definition must be weakened, since t input values may be delayed and not effect the result. We adopt the ABA functionality as defined in [34]. The asynchronous Byzantine agreement functionality, $\mathcal{F}_{\mathsf{ABA}}$, is presented in Figure 2.

Functionality $\mathcal{F}_{\mathsf{ABA}}$

$\mathcal{F}_{\mathsf{ABA}}$ proceeds as follows, running with parties $P_1, \ldots, P_n$ and an adversary $\mathcal{S}$:

- Upon receiving a message $(\mathsf{vote}, \mathsf{sid}, b)$, where $b \in \{0, 1\}$ from party P_i, send a message $(\mathsf{vote}, \mathsf{sid}, P_i, b)$ to the adversary. The adversary is also allowed to vote.
- The result is computed using one of the following rules:
  - If $n - t$ parties voted, and $t + 1$ voted b and $\mathcal{S}$ voted b, then set the result to be b.
  - If $n - t$ parties voted b, then set the result to be b.
  - If $n - t$ parties voted, but do not agree, then the result is set by the vote of $\mathcal{S}$.

  When the result of voting sid has been decided to be v, the functionality sends $(\mathsf{decide}, \mathsf{sid}, v)$ as a request-based delayed output to all parties.

Figure 2: The asynchronous Byzantine agreement functionality

4 Zero-Knowledge Proofs and Certificates

In order to ensure security against malicious behaviour, the parties must prove their actions using zero-knowledge proofs during the protocol. The zero-knowledge functionality $\mathcal{F}_{\mathsf{ZK}}$ and its one-to-many extension $\mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}$ are defined in Section 4.1 and the notion of certificates in Section 4.2.

4.1 Zero-Knowledge Proofs

In the *zero-knowledge functionality*, parameterized by a relation R, the prover sends the functionality a statement x to be proven along with a witness w. In response, the functionality forwards the statement x to the verifier if and only if $R(x, w) = 1$ (i.e., if and only if x a correct statement and w is a witness for x). Thus, in actuality, this is a proof of knowledge in that the verifier is assured that the prover actually knows w (and has explicitly sent w to the functionality), rather than just being assured that such a w exists. The zero-knowledge functionality, $\mathcal{F}_{\mathsf{ZK}}$, is presented in Figure 3.[5]

[5]For simplicity, we concentrate on the single-session version of $\mathcal{F}_{\mathsf{ZK}}$, which requires a separate common reference string for each protocol that realizes $\mathcal{F}_{\mathsf{ZK}}$. The protocols realizing $\mathcal{F}_{\mathsf{ZK}}$ will later be composed, using the universal composition with joint state of Canetti and Rabin [16], to obtain protocols that use only a single copy of the common reference string when realizing all the copies of $\mathcal{F}_{\mathsf{ZK}}$.

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

$\mathcal{F}_{\mathsf{ZK}}$ proceeds as follows, running with prover P, a verifier V and an adversary $\mathcal{S}$, and parameterized with a relation R:

- Upon receiving $(\mathsf{ZK\text{-}prover}, \mathsf{sid}, x, w)$ from P, do: if $R(x, w) = 1$, then send $(\mathsf{ZK\text{-}proof}, \mathsf{sid}, x)$ to $\mathcal{S}$, send a request-based delayed output $(\mathsf{ZK\text{-}proof}, \mathsf{sid}, x)$ to V and halt. Otherwise, halt.

Figure 3: The zero-knowledge functionality

The zero-knowledge functionality, as defined in Figure 3, is parameterized by a single relation R (and thus a different copy of $\mathcal{F}_{\mathsf{ZK}}$ is needed for every different relation required). In this work we require zero-knowledge proofs for several relations, therefore, we use standard techniques by considering the relation R index several predetermined relations. This can be implemented by separating the statement x into two parts: x_1 that indexes the relation to be used and x_2 that is the actual statement. Then, define $R((x_1, x_2), w)$ as $R_{x_1}(x_2, w)$.

We now define the *one-to-many extension* of the zero-knowledge functionality, where one party proves a statement to some subset of parties. The definition of the one-to-many zero-knowledge functionality, denoted $\mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}$, is presented in Figure 4.

Functionality $\mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}$

$\mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}$ proceeds as follows, running with parties $P_1, \ldots, P_n$ and an adversary $\mathcal{S}$, and parameterized with a relation R:

- Upon receiving $(\mathsf{ZK\text{-}prover}, \mathsf{sid}, \mathcal{P}, x, w)$ from party P_i, where $\mathcal{P} \subseteq \{P_1, \ldots, P_n\}$ do: if $R(x, w) = 1$, then send $(\mathsf{ZK\text{-}proof}, \mathsf{sid}, P_i, \mathcal{P}, x)$ to $\mathcal{S}$, a request-based delayed output $(\mathsf{ZK\text{-}proof}, \mathsf{sid}, P_i, \mathcal{P}, x)$ to all parties in $\mathcal{P}$ and halt. Otherwise, halt.

Figure 4: The one-to-many zero-knowledge functionality

4.2 Certificates

As we consider static corruptions, there exists efficient constant-round zero-knowledge protocols in the $\mathcal{F}_{\mathsf{CRS}}$-hybrid model, e.g., omega protocols [23], and even non-interactive zero-knowledge proofs [21]. These protocols would suffice for realizing $\mathcal{F}_{\mathsf{ZK}}$ as it is a two-party functionality. However, when considering the multiparty functionality $\mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}$, some problems may arise. The reason is that the statement that needs to be proven is not public, and a malicious prover may prove different statements to different parties.

This problem is resolved using *certificates*, introduced by Hirt et al. [30]. Certificates are generated by an interactive protocols among the parties such that at the end of the execution, one party can non-interactively prove correctness of some statement to each other party, without revealing additional information. The protocol for issuing a certificate is based on threshold signatures and involves two stages. First, a signature proving the statement is computed interactively with all the parties – it is essential that all the parties are active during this stage, otherwise the prover might not receive enough shares to reconstruct the signature. Next, the prover can send the signature as a non-interactive proof of the statement and every other party can validate it.

During out main protocol, in Section 5, we consider three relations:

- **Proof of Plaintext Knowledge.** The relations is parameterized by a TFHE scheme. The statement consists of a public encryption key ek and a ciphertext c and the witness consists of the plaintext x and random coins r, explaining c as an encryption of x under ek. That is

$$R_{\mathsf{PoPK}} = \{((ek, c), (x, r)) \mid c = \mathrm{Enc}_{ek}(x; r)\}\,.$$

- **Proof of Correct Decryption.** The relations is parameterized by a TFHE scheme. The statement consists of a public encryption key ek, a ciphertext c and a decryption share d and the witness consists of the decryption key dk. That is

$$R_{\mathsf{PoCD}} = \{((ek, c, d), dk) \mid d = \mathrm{DecShare}_{dk}(c)\}\,.$$

- **Proof of Correct Signature.** The relations is parameterized by a TSIG scheme. The statement consists of a public verification key vk, a message msg and a signature share σ and the witness consists of the signing key sk. That is

$$R_{\mathsf{PoCS}} = \{((vk, \mathsf{msg}, \sigma), sk) \mid \sigma = \mathrm{SignShare}_{sk}(\mathsf{msg})\}$$

Lemma 4.1. *Let $n > 2t + 1$ and let R_{x_1} be a binary relation. Assuming the existence of threshold signature schemes, $\mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}$ can be UC realized in the $\mathcal{F}_{\mathsf{ZK}}$-hybrid model in the presence of static malicious adversaries.*

Proof. Consider a party P_i, holding a witness w, that wishes to prove a statement x to all other parties. The high-level idea is for P_i to prove x to each other P_j using a two-party zero-knowledge proof. If all parties are active and P_i is honest, it is guaranteed that eventually at least $n - t$ proofs will successfully terminate. Once a verifier P_j accepts the proof, it produces a share σ_j of a signature approving x, sends the share back to P_i and proves the validity of σ_j to P_i using another two-party zero-knowledge proof. After P_i obtains $n - t$ valid signature shares, it can reconstruct the signature σ which serves as its certificate.

Assuming that $n > 2t + 1$, it holds that $(n - t) - t \geq 1$, and so it is guaranteed that at least one honest party accepted the proof of the statement x; it follows that the corrupted parties cannot falsely certify invalid statements. Furthermore, assuming the two-parties zero-knowledge proofs are constant round, certifying a statement takes constant time.

Protocol 4.2 shows how to compute $\mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}$ in the $\mathcal{F}_{\mathsf{ZK}}$-hybrid model. During the protocol, two instances of $\mathcal{F}_{\mathsf{ZK}}$ are used; the first is for proving statements for the relation R_{x_1} and the second for the relation R_{PoCS}. We use the notation sid_j^k for the string $\mathsf{sid} \circ k \circ j$.

Let $\mathcal{A}$ be an adversary attacking Protocol 4.2 and let $\mathcal{Z}$ be an environment. We construct a simulator $\mathcal{S}$ as follows. $\mathcal{S}$ runs the adversary $\mathcal{A}$ and simulates the environment, the honest parties and the ideal functionality $\mathcal{F}_{\mathsf{ZK}}$ towards $\mathcal{A}$. In order to simulate $\mathcal{Z}$, $\mathcal{S}$ forwards every message it receives from $\mathcal{Z}$ to $\mathcal{A}$ and vice-versa. $\mathcal{S}$ simulates the honest parties towards $\mathcal{A}$. In case P_i is corrupted, $\mathcal{S}$ receives $((x_1, x_2), w)$ by simulating $\mathcal{F}_{\mathsf{ZK}}$ and in addition receives $\mathcal{P}$ from $\mathcal{A}$. Next, $\mathcal{S}$ sends $(\mathsf{ZK\text{-}prover}, \mathsf{sid}, \mathcal{P}, (x_1, x_2), w)$ to $\mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}$ and continues simulating the honest parties and $\mathcal{F}_{\mathsf{ZK}}$ to $\mathcal{A}$. In case P_i is not corrupted, it first receives $(\mathsf{ZK\text{-}proof}, \mathsf{sid}, P_i, \mathcal{P}, (x_1, x_2))$ from $\mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}$. Next, whenever $\mathcal{A}$ requests output from $\mathcal{F}_{\mathsf{ZK}}$ with sid_j^1 for $j \in \mathcal{I}$, $\mathcal{S}$ replies with $(\mathsf{ZK\text{-}proof}, \mathsf{sid}, (x_1, x_2))$. The rest of the simulation follows the protocol. It is straight-forward to see that the view of $\mathcal{A}$ is indistinguishable when interacting with $\mathcal{S}$ and when attacking the execution of Protocol 4.2, and the proof follows. □

Protocol 4.2 ($\mathcal{F}_{\mathsf{ZK}}^{1:M}$ protocol, in the $\mathcal{F}_{\mathsf{ZK}}$-hybrid model)**.**

Offline setup:

For every $j \in [n]$, party P_j is initialized with keys for a threshold signature scheme (vk, sk_j), where $(sk, vk) \leftarrow \text{SigGen}_{(n-t,n)}(1^\kappa)$, and $sk = (sk_1, \ldots, sk_n)$.

Code for sender P_i:

- Upon receiving $(\mathsf{ZK\text{-}prover}, \mathsf{sid}, \mathcal{P}, (x_1, x_2), w)$ from the environment, party P_i sends $(\mathsf{ZK\text{-}prover}, \mathsf{sid}_j^1, (x_1, x_2), w)$ to $\mathcal{F}_{\mathsf{ZK}}$ where P_i acts as the prover and P_j acts as the verifier (for every $j \in [n] \setminus \{i\}$). In addition, send $(\mathsf{sid}, \mathcal{P})$ to every party.
- Request output from $\mathcal{F}_{\mathsf{ZK}}$ until receiving $(\mathsf{ZK\text{-}proof}, \mathsf{sid}_j^2, (\mathsf{PoCS}, vk, \mathsf{msg}, \sigma))$, with $\mathsf{msg} = \langle (x_1, x_2) \text{ is a valid statement, for } (\mathsf{sid}, \mathcal{P}) \rangle$ (for every $j \in [n] \setminus \{i\}$), until receiving $n - t$ signature shares $\{\sigma_j\}$.
- Compute $\mathsf{cert} = \text{SignRecon}(\{\sigma_j\})$, send $(\mathsf{sid}, (x_1, x_2), \mathsf{cert})$ to every party in $\mathcal{P}$ and halt.

Code for receiver P_j (for $j \neq i$):

- Requests output from $\mathcal{F}_{\mathsf{ZK}}$ until receiving $(\mathsf{ZK\text{-}proof}, \mathsf{sid}_j^1, (x_1, x_2))$. Next, upon receiving the message $(\mathsf{sid}, \mathcal{P})$ from P_i, set $\mathsf{msg} = \langle (x_1, x_2) \text{ is a valid statement, for } (\mathsf{sid}, \mathcal{P}) \rangle$, compute $\sigma_j = \text{SignShare}_{sk_j}(\mathsf{msg})$ and send $(\mathsf{ZK\text{-}prover}, \mathsf{sid}_j^2, (\mathsf{PoCS}, vk, \mathsf{msg}, \sigma_j), sk_j)$ to $\mathcal{F}_{\mathsf{ZK}}$ where P_j acts as the prover and P_i acts as the verifier.
- Upon receiving the first message $(\mathsf{sid}, (x_1, x_2), \mathsf{cert})$ from P_i set $\mathsf{msg} = \langle (x_1, x_2) \text{ is a valid statement, for } (\mathsf{sid}, \mathcal{P}) \rangle$ and verify that $\text{Vrfy}_{vk}(\mathsf{msg}, \mathsf{cert}) = 1$. If so output $(\mathsf{ZK\text{-}proof}, \mathsf{sid}, P_i, \mathcal{P}, (x_1, x_2))$ and halt.

The one-to-many zero-knowledge protocol

5 Asynchronous MPC Protocol

Following the spirit of [29, 30], the protocol consists of an offline key-distribution stage (preprocessing) followed three online stages: the input stage, the computation and threshold-decryption stage and the termination stage. We present the protocol for *public-output* functionalities, and a variant for *private-output* functionalities can be obtained using the technique of [29].

5.1 Key-Distribution Stage

The *key-distribution stage* can be computed once for multiple instances of the protocol and essentially distributes the keys for threshold schemes amongst the parties. We will describe the protocol in a hybrid model where the key-distribution is done by an ideal functionality $\mathcal{F}_{\mathsf{KeyDist}}$. This ideal functionality can be realized using any asynchronous MPC protocol that does not require preprocessing, e.g., [35]. We emphasize that the time complexity of the protocol realizing the key-distribution stage is *independent* of the function to compute.

$\mathcal{F}_{\mathsf{KeyDist}}$ generates the public and secret keys for the TFHE and the TSIG schemes and sends to each party its corresponding keys. The key-distribution functionality is described in Figure 5.

5.2 Input Stage

In the input stage, as described in Protocol 5.1, each party encrypts its input and sends it to all the other parties along with certificates proving that the party knows the plaintext (and so independence of inputs is retained) and that $n - t$ parties have obtained it. Next, the parties

Functionality $\mathcal{F}_{\mathsf{KeyDist}}$

$\mathcal{F}_{\mathsf{KeyDist}}$ proceeds as follows, interacting with parties $P_1, \ldots, P_n$ and an adversary $\mathcal{S}$, and parameterized by TFHE and TSIG schemes.

- Upon receiving a message $(\mathsf{keydist}, \mathsf{sid})$ from party P_i, do:
  1. If there is no value $(\mathsf{sid}, dk, ek, sk, vk)$ recorded, compute $(dk, ek) \leftarrow \mathrm{Gen}_{(t,n)}(1^\kappa)$, where $dk = (dk_1, \ldots, dk_n)$, and $(sk, vk) \leftarrow \mathrm{SigGen}_{(n-t,n)}(1^\kappa)$, where $sk = (sk_1, \ldots, sk_n)$ and record $(\mathsf{sid}, dk, ek, sk, vk)$.
  2. Send $(\mathsf{sid}, P_i, ek, vk)$ to $\mathcal{S}$ and a request-based delayed output[a] $(\mathsf{sid}, dk_i, ek, sk_i, vk)$ to P_i.

[a] This is the standard formalization of the asynchronous setting in the UC framework, see Section 3; P_i must request the output from $\mathcal{F}_{\mathsf{KeyDist}}$, and $\mathcal{S}$ can continuously instruct $\mathcal{F}_{\mathsf{KeyDist}}$ to arbitrarily delay the answer.

Figure 5: The key-distribution functionality

jointly agree on a common subset of input providers, $\mathsf{CoreSet}$, which consists of $n-t$ parties whose encrypted input has been obtained by all the parties. This stage proceeds in a similar manner to [29] with the difference that the plaintexts are encrypted using TFHE rather than TAHE.

In more details, each party P_i starts by encrypting its input $c_i \leftarrow \mathrm{Enc}_{ek}(x_i)$, and proving to each other party knowledge of the plaintext. Once a party P_j accepts the proof, it sends P_i a signature share for the statement $\mathsf{msg} = \langle n-t \text{ parties hold the input } c_i \text{ of } P_i \rangle$. After P_i obtains $n-t$ signature shares, it can reconstruct and distribute the certificate $\mathsf{cert}_i^{\mathsf{input}}$, which is essentially a signature on msg.

When a party collects $n-t$ certificates it knows that at least $n-t$ parties have their certified inputs distributed to at least $n-t$ parties. Since $n \geq 2t+1$, by assumption, this means that at least $(n-t)-t \geq 1$ *honest* parties obtained certified inputs from at least $n-t$ parties. Hence, if the honest parties echo the certified inputs they receive and collect $n-t$ echoes, then all honest parties will end up holding the certified inputs of the $n-t$ parties which had their certified inputs distributed to at least one honest party. These $n-t$ parties will eventually be the input providers. To determine who they are, the asynchronous Byzantine agreements functionality $\mathcal{F}_{\mathsf{ABA}}$ is invoked (concurrently) n times. During the protocol description we use the notation sid_j^k for the string $\mathsf{sid} \circ k \circ j$.

5.3 Computation and Threshold Decryption Stage

In the computation and threshold-decryption stage, as described in Protocol 5.2, each party locally prepares the circuit $\mathsf{Circ}(\mathsf{CoreSet})$ (with hard-wired default input values for parties outside $\mathsf{CoreSet}$) and evaluates it over the encrypted input ciphertexts that were agreed upon in the input stage. Since the encryption scheme is fully homomorphic, this part is done without interaction between the parties. Once the encrypted output $\tilde{c}_i$ is obtained, P_i computes a decryption share d_i and interactively certifies it. Next, P_i sends the certified decryption share to all other parties and waits until it receives $t+1$ certified decryption shares, from which it can reconstruct the output y_i.

Once P_i obtains the output, it should send it to all other parties in order to trigger the termination stage. This is done by first computing a signature share $\sigma_i^{\mathsf{output}}$ for the statement that y_i

Protocol 5.1 (The input stage, in the $(\mathcal{F}_{\mathsf{KeyDist}}, \mathcal{F}_{\mathsf{ZK}}, \mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}, \mathcal{F}_{\mathsf{ABA}})$-hybrid)**.**

Setup: Upon receiving input $(\mathsf{input}, \mathsf{sid}, x_i)$ from the environment, proceed as follows:

1. Send $(\mathsf{keydist}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{KeyDist}}$.
2. Request the output from $\mathcal{F}_{\mathsf{KeyDist}}$ until receiving $(\mathsf{sid}, dk_i, ek, sk_i, vk)$.
3. Initialize the following sets to $\emptyset$: $\mathsf{VerProv}_i$ (verified input providers), $\mathsf{VerDistProv}_i$ (verified distributed input providers), $\mathsf{GlobalProv}_i$ (globally verified distributed input providers), $\mathsf{CertInputs}_i$ (certified inputs) and $\mathsf{GlobalInputs}_i$ (globally certified inputs).

Distribution of Encrypted Input:

1. Compute $c_i = \mathrm{Enc}_{ek}(x_i; r_i)$ (for uniformly distributed r_i).
2. Send $(\mathsf{ZK\text{-}prover}, \mathsf{sid}_i^1, \{P_1, \ldots, P_n\} \setminus \{P_i\}, (\mathsf{PoPK}, ek, c_i), (x_i, r_i))$ to $\mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}$.
3. Request output from $\mathcal{F}_{\mathsf{ZK}}$ (with $\mathsf{sid}_{i,j}^2$ for every $j \in [n] \setminus \{i\}$) until receiving $(\mathsf{ZK\text{-}proof}, \mathsf{sid}_{i,j}^2, (\mathsf{PoCS}, vk, \mathsf{msg}, \sigma_j^{\mathsf{input}_i}))$, where P_i acts as the verifier and P_j acts as the prover, with $\mathsf{msg} = \langle n - t$ parties hold the input c_i of $P_i\rangle$, until receiving $n - t$ signature shares $\{\sigma_j^{\mathsf{input}_i}\}$.
4. Compute the certificate $\mathsf{cert}_i^{\mathsf{input}} = \mathrm{SignRecon}(\{\sigma_j^{\mathsf{input}_i}\})$ (which equals $\mathrm{Sign}_{sk}(\mathsf{msg})$). Send $(\mathsf{sid}, \mathsf{msg}, c_i, \mathsf{cert}_i^{\mathsf{input}})$ to all the parties.

Grant Certificate:

Request the output from $\mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}$ (with sid_j^1 for every $j \in [n] \setminus \{i\}$). Upon receiving $(\mathsf{ZK\text{-}proof}, \mathsf{sid}_j^1, P_j, \{P_1, \ldots, P_n\} \setminus \{P_j\}, (\mathsf{PoPK}, ek, c_j))$, add j to $\mathsf{VerProv}_i$. Next, set the message $\mathsf{msg} = \langle n - t$ parties hold the input c_j of $P_j\rangle$, compute $\sigma_i^{\mathsf{input}_j} = \mathrm{SignShare}_{sk_i}(\mathsf{msg})$, and send $(\mathsf{ZK\text{-}prover}, \mathsf{sid}_{j,i}^2, (\mathsf{PoCS}, vk, \mathsf{msg}, \sigma_i^{\mathsf{input}_j}), sk_i)$ to $\mathcal{F}_{\mathsf{ZK}}$, where P_i acts as the prover and P_j as the verifier.

Echo Certificate:

Upon receving $(\mathsf{sid}, \mathsf{msg}, c_j, \mathsf{cert}_j^{\mathsf{input}})$ with the message $\mathsf{msg} = \langle n - t$ parties hold the input c_j of $P_j\rangle$ and $\mathrm{Vrfy}_{vk}(\mathsf{msg}, \mathsf{cert}_j^{\mathsf{input}}) = 1$, check if $j \notin \mathsf{VerDistProv}_i$. If so, add j to $\mathsf{VerDistProv}_i$, add $(c_j, \mathsf{cert}_j^{\mathsf{input}})$ to $\mathsf{CertInputs}_i$ and forward $(\mathsf{sid}, \mathsf{msg}, c_j, \mathsf{cert}_j^{\mathsf{input}})$ to all the parties.

Select Input Providers:

When $|\mathsf{VerDistProv}_i| \geq n - t$, stop executing the above rules and proceed as follows:

1. Send $(\mathsf{sid}, \mathsf{VerProv}_i, \mathsf{CertInputs}_i)$ to all the parties.
2. Collect a set of $\{(\mathsf{VerProv}_j, \mathsf{CertInputs}_j)\}_{j \in J}$ of $n - t$ pairs.
3. Let $\mathsf{GlobalProv}_i = \cup_{j \in J} \mathsf{VerProv}_j$ and $\mathsf{GlobalInputs}_i = \cup_{j \in J} \mathsf{CertInputs}_j$.
4. For $j \in [n]$, send $(\mathsf{vote}, \mathsf{sid}_j^3, v_j)$ to $\mathcal{F}_{\mathsf{ABA}}$, where $v_j = 1$ iff $j \in \mathsf{GlobalProv}_i$.
5. Request the outputs from $\mathcal{F}_{\mathsf{ABA}}$ until receiving $(\mathsf{decide}, \mathsf{sid}_j^3, w_j)$ for every $j \in [n]$.
6. Denote $\mathsf{CoreSet} = \{j \in [n] \mid w_j = 1\}$.
7. For each $j \in \mathsf{GlobalProv}_i \cap \mathsf{CoreSet}$, send $(\mathsf{sid}, c_j, \mathsf{cert}_j^{\mathsf{input}})$ to all the parties (note that $(c_j, \mathsf{cert}_j^{\mathsf{input}}) \in \mathsf{GlobalInputs}_i$).
8. Wait until receiving $(c_j, \mathsf{cert}_j^{\mathsf{input}})$ for every $j \in \mathsf{CoreSet}$.

The input stage code for P_i

is the output value, interactively certify $\sigma_i^{\mathsf{output}}$ and send it to all parties. Once P_i receives $n-t$ signature shares it can reconstruct a certificate proving that y_i is indeed the output value. Finally P_i sends y_i along with the certificate to all the parties.

Protocol 5.2 (The computation and threshold-decryption stage)**.**
Wait until input stage is completed, resulting with a core set $\mathsf{CoreSet}$ and input ciphertexts $\{c_j \mid j \in \mathsf{CoreSet}\}$.

Circuit Evaluation:

1. For each $j \notin \mathsf{CoreSet}$, hard-wire the default value $\tilde{x}_j$ for P_j into the circuit Circ, denote the new circuit by $\mathsf{Circ}(\mathsf{CoreSet})$.
2. Locally compute the homomorphic evaluation of the circuit
$$\tilde{c}_i = \mathrm{Eval}_{ek}\left(\mathsf{Circ}(\mathsf{CoreSet}), c_{j_1}, \ldots, c_{j_{|\mathsf{CoreSet}|}}\right).$$

Threshold Decryption:

1. Compute the decryption share $d_i = \mathrm{DecShare}_{dk_i}(\tilde{c}_i)$.
2. Send $(\mathsf{ZK\text{-}prover}, \mathsf{sid}_i^4, \{P_1, \ldots, P_n\} \setminus \{P_i\}, ((\mathsf{PoCD}, ek, \tilde{c}_i, d_i), dk_i)$ to $\mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}$.
3. Request the output from $\mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}$ (for every $j \in [n] \setminus \{i\}$). Upon receiving $(\mathsf{ZK\text{-}proof}, \mathsf{sid}_j^4, P_j, \{P_1, \ldots, P_n\} \setminus \{P_j\}, (\mathsf{PoCD}, ek, \tilde{c}_j, d_j))$, accept the proof if $\tilde{c}_i = \tilde{c}_j$.
4. Once $t+1$ decryption shares with accepted proofs $\{(ek, \tilde{c}_i, d_j)\}$ have arrived, reconstruct the output $y_i = \mathrm{DecRecon}(\{d_j\})$.
5. Set $\mathsf{msg} = \langle y_i \text{ is the output value} \rangle$ and compute $\sigma_i^{\mathsf{output}} = \mathrm{SignShare}_{sk_i}(\mathsf{msg})$.
6. Send $(\mathsf{ZK\text{-}prover}, \mathsf{sid}_i^5, \{P_1, \ldots, P_n\} \setminus \{P_i\}, (\mathsf{PoCS}, vk, \mathsf{msg}, \sigma_i^{\mathsf{output}}), sk_i)$ to $\mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}$.
7. Request output from $\mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}$ (for $j \in [n] \setminus \{i\}$) until receiving $(\mathsf{ZK\text{-}proof}, \mathsf{sid}_j^5, P_j, \{P_1, \ldots, P_n\} \setminus \{P_j\}, (\mathsf{PoCS}, vk, \mathsf{msg}, \sigma_j^{\mathsf{output}}))$, with $\mathsf{msg} = \langle y_i \text{ is the output value} \rangle$.
8. Compute the certificate $\mathsf{cert}_i^{\mathsf{output\text{-}verified}} = \mathrm{SignRecon}(\{\sigma_j^{\mathsf{output}_i}\})$ (which equals $\mathrm{Sign}_{sk}(\mathsf{msg})$ with $\mathsf{msg} = \langle y_i \text{ is the output value} \rangle$).
Send $(\mathsf{sid}, \mathsf{msg}, \mathsf{cert}_i^{\mathsf{output\text{-}verified}})$ to all the parties.

The computation and threshold-decryption stage code for P_i

5.4 Termination Stage

The termination stage, as described in Protocol 5.3, ensures that all honest parties will eventually terminate the protocol, and will do so with the same output. Recall that the computation and threshold-decryption stage is concluded when a party sends a certified output value to all the parties. The party cannot terminate at this point since it might be required to assist in certifying statements for other parties. Therefore, during the entire course of the protocol the termination code is run concurrently. The termination stage follows the technique of Bracha [13]. In this stage, each party continuously collects certified outputs sent by other parties. Once it receives $t+1$ certified outputs of the same value it knows that this is the correct output value for the computation (since at least one honest party sent it). The party then adopts this certified output as its own output (in case it did not obtain the output value earlier) and echoes it to all other parties. Once the party receives $n-t$ certified outputs of the same value, it can terminate.

Protocol 5.3 (The termination stage)**.**
During the protocol, concurrently executes the following rule:

Collecting Output Values:
When receiving for the first time from party P_j the value $(\mathsf{sid}, \mathsf{msg}, \mathsf{cert}_j^{\text{output-verified}})$, with $\mathsf{msg} = \langle y_j \text{ is the output value} \rangle$ and $\text{Vrfy}_{vk}(\mathsf{msg}, \mathsf{cert}_j^{\text{output-verified}}) = 1$.

1. If the value y_j has arrived from $t+1$ parties and the output of P_i is not set to be y_j, then set the output y_i to be y_j and echo $(\mathsf{sid}, \mathsf{msg}, \mathsf{cert}_j^{\text{output-verified}})$ to all the parties.
2. If the value y_j has arrived from $n-t$ parties, then terminate with output $(\mathsf{output}, \mathsf{sid}, (\mathsf{CoreSet}, y_i))$.

The termination stage code for P_i

6 Proof of Security

Lemma 6.1. *Let f be an n-party functionality and assume the existence of TFHE and TSIG schemes. Then the protocol π described in Protocol 5.1, Protocol 5.2 and Protocol 5.3 UC realizes $\mathcal{F}_{\mathsf{ASFE}}^{f}$ in the $(\mathcal{F}_{\mathsf{KeyDist}}, \mathcal{F}_{\mathsf{ZK}}, \mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}, \mathcal{F}_{\mathsf{ABA}})$-hybrid model, in constant time, in the presence of static malicious adversaries corrupting at most t parties, for $t < n/2$.*

Proof. Let $\mathcal{A}$ be a static malicious adversary against the execution of π and let $\mathcal{Z}$ be an environment. Denote by $\mathcal{I}$ the set of indices of the corrupted parties. We construct an ideal-process adversary $\mathcal{S}$, interacting with the environment $\mathcal{Z}$ and with the ideal functionality $\mathcal{F}_{\mathsf{ASFE}}^{f}$. $\mathcal{S}$ constructs virtual real-model honest parties and runs the real-model adversary $\mathcal{A}$. $\mathcal{S}$ must simulate the view of $\mathcal{A}$, i.e., its communication with $\mathcal{Z}$, the messages sent by the uncorrupted parties, and the interactions with the functionalities $(\mathcal{F}_{\mathsf{KeyDist}}, \mathcal{F}_{\mathsf{ZK}}, \mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}, \mathcal{F}_{\mathsf{ABA}})$.

In order to simulate the communication with $\mathcal{Z}$, every message that $\mathcal{S}$ receives from $\mathcal{Z}$ is sent to $\mathcal{A}$, and likewise, every message sent from $\mathcal{A}$ sends to $\mathcal{Z}$ is forwarded by $\mathcal{S}$.

Simulating the input stage. $\mathcal{S}$ starts by simulating $\mathcal{F}_{\mathsf{KeyDist}}$ and generates the cryptographic keys by computing $(dk, ek) \leftarrow \text{Gen}_{(t,n)}(1^{\kappa})$, where $dk = (dk_1, \ldots, dk_n)$, and $(sk, vk) \leftarrow \text{SigGen}_{(n-t,n)}(1^{\kappa})$, where $sk = (sk_1, \ldots, sk_n)$, and recording (dk, ek, sk, vk). Upon request from $\mathcal{A}$, $\mathcal{S}$ sends the corresponding keys (dk_i, ek, sk_i, vk) for each corrupted party P_i $(i \in \mathcal{I})$.

Next, $\mathcal{S}$ simulates the operations of all honest parties in the input stage (Protocol 5.1). During the *Distribution of Encrypted Input* phase, $\mathcal{S}$ sets every ciphertext of an honest party to be an encryption of zero, that is for every $j \notin \mathcal{I}$, compute $c_j \leftarrow \text{Enc}_{ek}(0)$. When the adversary send a request to $\mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}$ with sid_j^1 (for $j \notin \mathcal{I}$) on behalf of a corrupted party, $\mathcal{S}$ responds with a confirmation of the validity of the ciphertext c_j, i.e., with $(\mathsf{ZK\text{-}proof}, \mathsf{sid}_j^1, P_j, \{P_1, \ldots, P_n\} \setminus \{P_j\}, (\mathsf{PoPK}, ek, c_j))$. When a corrupted party P_i $(i \in \mathcal{I})$ sends $(\mathsf{ZK\text{-}prover}, \mathsf{sid}_i^1, \{P_1, \ldots, P_n\} \setminus \{P_i\}, (\mathsf{PoPK}, ek, c_i), (x_i, r_i))$ to $\mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}$, $\mathcal{S}$ confirms that indeed $c_i = \text{Enc}_{ek}(x_i; r_i)$ and if so records the input x_i. $\mathcal{S}$ continues to simulate the honest parties by following the protocol; in all other calls to $\mathcal{F}_{\mathsf{ZK}}$, $\mathcal{S}$ responds according to the ideal functionality. When the simulation reaches the *Select Input Providers* phase, $\mathcal{S}$ simulates the interface to $\mathcal{F}_{\mathsf{ABA}}$ to $\mathcal{A}$. When the first honest party completes the simulated input stage, $\mathcal{S}$ learns the set $\mathsf{CoreSet}$.

Note that $\mathcal{S}$ learned the input values that were used by the adversary $\mathcal{A}$ on behalf of the corrupted parties that were selected to be input providers. This follows since for every $i \in \mathcal{I} \cap$

$\mathsf{CoreSet}$, there exists an honest party that confirmed the ciphertext c_i and sent a signature share to P_i (except for the negligible probability that $\mathcal{A}$ managed to forge a signature). It follows that the corrupted party must have sent its input to $\mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}$ during the *Distribution of Encrypted Input* phase, and so its input value x_i was recorded by $\mathcal{S}$.

Interacting with $\mathcal{F}_{\mathsf{ASFE}}^{f}$. Once $\mathcal{S}$ learns $\mathsf{CoreSet}$, it sends to $\mathcal{F}_{\mathsf{ASFE}}^{f}$ the input value x_i that was recorded for each $i \in \mathcal{I} \cap \mathsf{CoreSet}$, the input value $x_i = 0$ for each $i \in \mathcal{I} \setminus \mathsf{CoreSet}$ and the set $\mathsf{CoreSet}$ as the set of input providers. Once $\mathcal{S}$ receives back the output value y, it starts the simulation of the computation and threshold-decryption stage.

Simulating the computation and threshold-decryption stage. In order to simulate the honest parties in this stage (Protocol 5.2), $\mathcal{S}$ proceeds as follows. Initially, $\mathcal{S}$ computes the evaluated ciphertext $\tilde{c}$ based on the input ciphertexts of the input providers, i.e., $\tilde{c} = \mathrm{Eval}_{ek}(\mathsf{Circ}(\mathsf{CoreSet}), c_{j_1}, \ldots, c_{j_{|\mathsf{CoreSet}|}})$. Next, for every $i \in \mathcal{I}$, use the share of the decryption key dk_i to compute the decryption share $d_i = \mathrm{DecShare}_{dk_i}(\tilde{c})$. $\mathcal{S}$ then sets the decryption share d_j, for every $j \notin \mathcal{I}$, such that $(d_1, \ldots, d_n)$ form a secret sharing of the output value y. When the adversary sends a request to $\mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}$ with sid_j^4 (for $j \notin \mathcal{I}$) on behalf of a corrupted party, $\mathcal{S}$ responds with a confirmation of the validity of the decryption share d_j, i.e., with $(\mathsf{ZK\text{-}proof}, \mathsf{sid}_j^4, P_j, \{P_1, \ldots, P_n\} \setminus \{P_j\}, (\mathsf{PoCD}, ek, \tilde{c}, d_j))$. $\mathcal{S}$ continues to simulate the honest parties by following the protocol; in all other calls to $\mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}$, $\mathcal{S}$ responds according to the ideal functionality.

Simulating the termination stage. $\mathcal{S}$ simulates the honest parties in the termination stage (Protocol 5.3) by following the protocol;

We now define a series of hybrid games that will be used to prove the indistinguishability of the real and ideal worlds. The output of each game is the output of the environment.

The game $\mathrm{REAL}_{\pi,\mathcal{A},\mathcal{Z}}$. This is exactly the execution of the protocol π in the real-model with environment $\mathcal{Z}$ and adversary $\mathcal{A}$ (and ideal functionalities $(\mathcal{F}_{\mathsf{KeyDist}}, \mathcal{F}_{\mathsf{ZK}}, \mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}, \mathcal{F}_{\mathsf{ABA}})$).

The game $\mathrm{HYB}^{1}_{\pi,\mathcal{A},\mathcal{Z}}$. In this game, we modify the real-model experiment in the computation stage as follows. Whenever a corrupted party requests output from $\mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}$ with sid_j^4 (for $j \notin \mathcal{I}$), the response from $\mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}$ is $(\mathsf{ZK\text{-}proof}, \mathsf{sid}_j^4, P_j, \{P_1, \ldots, P_n\} \setminus \{P_j\}, (\mathsf{PoCD}, ek, \tilde{c}, d_j))$, without checking if P_j sent a valid witness.

Claim 6.2. $\mathrm{REAL}_{\pi,\mathcal{A},\mathcal{Z}} \equiv \mathrm{HYB}^{1}_{\pi,\mathcal{A},\mathcal{Z}}$.

Proof. This follows since in the execution of π, honest parties always send a valid witness to $\mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}$, and so the response from $\mathcal{F}_{\mathsf{ZK}}^{1:\mathrm{M}}$ is the same in both games. □

The game $\mathrm{HYB}^{2}_{\pi,\mathcal{A},\mathcal{Z}}$. This game is just like an execution of HYB^{1} except for the computation of the decryption shares of honest parties during the computation stage. Let y be the output of f, let $\tilde{c}$ be the evaluated ciphertext, let dk_i (for $i \in \mathcal{I}$) be the shares of the decryption key held by the corrupted parties, and let $d_i = \mathrm{DecShare}_{dk_i}(\tilde{c})$ be the corresponding decryption shares. Then, instead of computing the decryption share of the honest parties as $d_j = \mathrm{DecShare}_{dk_j}(\tilde{c})$ (for $j \notin \mathcal{I}$), the decryption shares are computed such that $(d_1, \ldots, d_n)$ form a secret sharing of the output value y.

Claim 6.3. $\text{HYB}^1_{\pi,\mathcal{A},\mathcal{Z}} \stackrel{c}{\equiv} \text{HYB}^2_{\pi,\mathcal{A},\mathcal{Z}}$.

Proof. The ability to compute the decryption shares of the honest parties follows from the properties of the secret sharing scheme.[6] Computational indistinguishability follows from the semantic security of the TFHE scheme. □

The game $\text{HYB}^3_{\pi,\mathcal{A},\mathcal{Z}}$. This game is just like an execution of HYB^2 except for the following difference. Whenever a corrupted party requests output from $\mathcal{F}_{\mathsf{ZK}}^{1:\text{M}}$ with sid_j^1 (for $j \notin \mathcal{I}$), the response from $\mathcal{F}_{\mathsf{ZK}}^{1:\text{M}}$ is $(\mathsf{ZK\text{-}proof}, \mathsf{sid}_j^1, P_j, \{P_1, \ldots, P_n\} \setminus \{P_j\}, (\mathsf{PoPK}, ek, c_j))$, without checking if P_j sent a valid witness.

Claim 6.4. $\text{HYB}^2_{\pi,\mathcal{A},\mathcal{Z}} \equiv \text{HYB}^3_{\pi,\mathcal{A},\mathcal{Z}}$.

Proof. This follows since in the execution of π, honest parties always send a valid witness to $\mathcal{F}_{\mathsf{ZK}}^{1:\text{M}}$, and so the response from $\mathcal{F}_{\mathsf{ZK}}^{1:\text{M}}$ is the same in both games. □

The game $\text{HYB}^{4,\ell}_{\pi,\mathcal{A},\mathcal{Z}}$. This game is just like an execution of HYB^3 with the following difference. In the input stage, in case $i \leq \ell$ honest party P_i encrypts its actual input $c_i \leftarrow \text{Enc}_{ek}(x_i)$, whereas in case $i > \ell$ P_i encrypts zeros $c_i \leftarrow \text{Enc}_{ek}(0)$. (Note that $\text{HYB}^{4,n}$ is exactly HYB^3.)

Claim 6.5. *For every* $\ell \in \{0, \ldots, n-1\}$, $\text{HYB}^{4,\ell}_{\pi,\mathcal{A},\mathcal{Z}} \stackrel{c}{\equiv} \text{HYB}^{4,\ell+1}_{\pi,\mathcal{A},\mathcal{Z}}$.

Proof. This follows from the semantic security of the encryption scheme. □

Claim 6.6. $\text{HYB}^{4,0}_{\pi,\mathcal{A},\mathcal{Z}} \equiv \text{IDEAL}_{f,\mathcal{S},\mathcal{Z}}$.

Proof. This follows since the joint behaviour of ideal functionalities $(\mathcal{F}_{\mathsf{KeyDist}}, \mathcal{F}_{\mathsf{ZK}}, \mathcal{F}_{\mathsf{ABA}})$, the modified behaviour of the ideal functionality $\mathcal{F}_{\mathsf{ZK}}^{1:\text{M}}$ and the behaviour of the honest parties in $\text{HYB}^{4,0}$ is identical to the simulation done by $\mathcal{S}$. □

Combining Claims 7-11, we conclude that $\text{REAL}_{\pi,\mathcal{A},\mathcal{Z}} \stackrel{c}{\equiv} \text{IDEAL}_{f,\mathcal{S},\mathcal{Z}}$. □

7 Conclusions

By Lemma 4.1, $\mathcal{F}_{\mathsf{ZK}}^{1:\text{M}}$ can be realized in the $\mathcal{F}_{\mathsf{ZK}}$-hybrid model (assuming the existence of TSIG and an honest majority). Assuming the existence of enhanced trapdoor permutations, $\mathcal{F}_{\mathsf{ZK}}$ can be UC realized in the $\mathcal{F}_{\mathsf{CRS}}$-hybrid model non-interactively (meaning that the prover sends a single message to the verifier) [21]. Using universal composition with joint state [16], a multi-session version of $\mathcal{F}_{\mathsf{ZK}}$ that requires a single copy of the CRS can be used. We thus obtain the following theorem from Lemma 6.1:

Theorem 7.1 (formal statement of Theorem 1.1)**.** *Let f be an n-party function and assume that enhanced trapdoor permutations, TFHE schemes and TSIG schemes exist. Then $\mathcal{F}_{\mathsf{ASFE}}^f$ can be UC realized in the $(\mathcal{F}_{\mathsf{CRS}}, \mathcal{F}_{\mathsf{KeyDist}}, \mathcal{F}_{\mathsf{ABA}})$-hybrid model, in constant time, in the presence of static malicious adversaries corrupting at most t parties, for $t < n/2$.*

[6] In the scheme of Shamir [39], fix the points corresponding to the shares d_i (for $i \in \mathcal{I}$) and the secret y, create a degree t polynomial interpolating these points, and compute the shares d_j (for $j \notin \mathcal{I}$) accordingly.

During the input stage (Protocol 5.1) the functionality $\mathcal{F}_{\mathsf{ABA}}$ is concurrently invoked n times. If $\mathcal{F}_{\mathsf{ABA}}$ is instantiated using a constant expected round protocol, e.g., the protocol of Canetti and Rabin [15], the time complexity of the concurrent composition will result with expectancy of $\log(n)$. Ben-Or and El-Yaniv [7] constructed a concurrent ABA protocol that runs in constant expected time, assuming that $t < n/3$.[7] We therefore conclude with the following corollary.

Corollary 7.2 (formal statement of Corollary 1.2)**.** *Let f be an n-party function and assume that enhanced trapdoor permutations, TFHE schemes and TSIG schemes exist. Then $\mathcal{F}_{\mathsf{ASFE}}^{f}$ can be UC realized in the $(\mathcal{F}_{\mathsf{CRS}}, \mathcal{F}_{\mathsf{KeyDist}})$-hybrid model, in constant expected time, in the presence of static malicious adversaries corrupting at most t parties, for $t < n/3$.*

Acknowledgements

We would like to thank Yehuda Lindell and Ran Canetti for helpful discussions on modeling asynchronous MPC in the UC framework, and to Juan Garay for pointing us to the paper of Ben-Or and El-Yaniv [7].

References

[1] J. F. Almansa, I. Damgård, and J. B. Nielsen. Simplified Threshold RSA with Adaptive and Proactive Security. In *Advances in Cryptology – EUROCRYPT 2006*, pages 593–611, 2006.

[2] G. Asharov, A. Jain, A. López-Alt, E. Tromer, V. Vaikuntanathan, and D. Wichs. Multiparty Computation with Low Communication, Computation and Interaction via Threshold FHE. In *Advances in Cryptology – EUROCRYPT 2012*, pages 483–501, 2012.

[3] M. Backes, F. Bendun, A. Choudhury, and A. Kate. Asynchronous MPC with a strict honest majority using non-equivocation. In *Proceedings of the 33rd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 10–19, 2014.

[4] D. Beaver, S. Micali, and P. Rogaway. The Round Complexity of Secure Protocols (Extended Abstract). In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing (STOC)*, pages 503–513, 1990.

[5] Z. Beerliová-Trubíniová and M. Hirt. Simple and Efficient Perfectly-Secure Asynchronous MPC. In *Advances in Cryptology – ASIACRYPT 2007*, pages 376–392, 2007.

[6] Z. Beerliová-Trubíniová, M. Hirt, and J. B. Nielsen. On the theoretical gap between synchronous and asynchronous MPC protocols. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 211–218, 2010.

[7] M. Ben-Or and R. El-Yaniv. Resilient-optimal interactive consistency in constant time. *Distributed Computing*, 16(4):249–262, 2003.

[8] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1–10, 1988.

[7] Although the protocol in [7] is proved based on the property-based definition of ABA, a simulation-based proof should follow as we consider static adversaries.

[9] M. Ben-Or, R. Canetti, and O. Goldreich. Asynchronous secure computation. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC)*, pages 52–61, 1993.

[10] M. Ben-Or, B. Kelmer, and T. Rabin. Asynchronous Secure Computations with Optimal Resilience (Extended Abstract). In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 183–192, 1994.

[11] R. Bendlin and I. Damgård. Threshold Decryption and Zero-Knowledge Proofs for Lattice-Based Cryptosystems. In *Theory of Cryptography, 7th Theory of Cryptography Conference, TCC 2010*, pages 201–218, 2010.

[12] R. Bendlin, S. Krehbiel, and C. Peikert. How to Share a Lattice Trapdoor: Threshold Protocols for Signatures and (H)IBE. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security (ACNS)*, pages 218–236, 2013.

[13] G. Bracha. An Asynchronous $[(n-1)/3]$-Resilient Consensus Protocol. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 154–162, 1984.

[14] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *Proceedings of the 42nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 136–145, 2001.

[15] R. Canetti and T. Rabin. Fast asynchronous Byzantine agreement with optimal resilience. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC)*, pages 42–51, 1993.

[16] R. Canetti and T. Rabin. Universal Composition with Joint State. In *Advances in Cryptology – CRYPTO 2003*, pages 265–281, 2003.

[17] D. Chaum, C. Crépeau, and I. Damgård. Multiparty Unconditionally Secure Protocols (Extended Abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC)*, pages 11–19, 1988.

[18] A. Choudhury and A. Patra. Optimally Resilient Asynchronous MPC with Linear Communication Complexity. In *Proceedings of the 16th International Conference on Distributed Computing and Networking (ICDCN)*, page 5, 2015.

[19] A. Choudhury, M. Hirt, and A. Patra. Asynchronous Multiparty Computation with Linear Communication Complexity. In *Proceedings of the 27th International Symposium on Distributed Computing (DISC)*, pages 388–402, 2013.

[20] R. Cramer, I. Damgård, and J. B. Nielsen. Multiparty Computation from Threshold Homomorphic Encryption. In *Advances in Cryptology – EUROCRYPT 2001*, pages 280–299, 2001.

[21] A. De Santis, G. Di Crescenzo, R. Ostrovsky, G. Persiano, and A. Sahai. Robust Non-interactive Zero Knowledge. In *Advances in Cryptology – CRYPTO 2001*, pages 566–598, 2001.

[22] M. J. Fischer, N. A. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, 1985.

[23] J. A. Garay, P. D. MacKenzie, and K. Yang. Strengthening Zero-Knowledge Protocols Using Signatures. *Journal of Cryptology*, 19(2):169–209, 2006.

[24] S. Garg, C. Gentry, S. Halevi, and M. Raykova. Two-Round Secure MPC from Indistinguishability Obfuscation. In *Theory of Cryptography, 10th Theory of Cryptography Conference, TCC 2014*, pages 74–94, 2014.

[25] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis.

[26] O. Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004. ISBN 0-521-83084-2.

[27] O. Goldreich, S. Micali, and A. Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 218–229, 1987.

[28] D. Gordon, F. Liu, and E. Shi. Constant-Round MPC with Fairness and Guarantee of Output Delivery. In *Advances in Cryptology – CRYPTO 2015*, pages 63–82, 2015.

[29] M. Hirt, J. B. Nielsen, and B. Przydatek. Cryptographic Asynchronous Multi-party Computation with Optimal Resilience (Extended Abstract). In *Advances in Cryptology – EUROCRYPT 2005*, pages 322–340, 2005.

[30] M. Hirt, J. B. Nielsen, and B. Przydatek. Asynchronous Multi-Party Computation with Quadratic Communication. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 473–485, 2008.

[31] Y. Ishai, M. Prabhakaran, and A. Sahai. Founding Cryptography on Oblivious Transfer - Efficiently. In *Advances in Cryptology – CRYPTO 2008*, pages 572–591, 2008.

[32] J. Katz, U. Maurer, B. Tackmann, and V. Zikas. Universally Composable Synchronous Computation. In *Theory of Cryptography, 10th Theory of Cryptography Conference, TCC 2013*, pages 477–498, 2013.

[33] P. Mukherjee and D. Wichs. Two Round MPC from LWE via Multi-Key FHE. Cryptology ePrint Archive, Report 2015/345, 2015. http://eprint.iacr.org/.

[34] J. B. Nielsen. A Threshold Pseudorandom Function Construction and Its Applications. In *Advances in Cryptology – CRYPTO 2002*, pages 401–416, 2002.

[35] A. Patra, A. Choudhary, and C. P. Rangan. Communication Efficient Statistical Asynchronous Multiparty Computation with Optimal Resilience. In *Proceedings of the 5th International Conference on Information Security and Cryptology (Inscrypt)*, pages 179–197, 2009.

[36] A. Patra, A. Choudhury, and C. P. Rangan. Efficient Asynchronous Verifiable Secret Sharing and Multiparty Computation. *Journal of Cryptology*, 28(1):49–109, 2015.

[37] B. Prabhu, K. Srinathan, and C. P. Rangan. Asynchronous Unconditionally Secure Computation: An Efficiency Improvement. In *Proceedings of the Third International Conference on Cryptology in India (INDOCRYPT)*, pages 93–107, 2002.

[38] T. Rabin and M. Ben-Or. Verifiable Secret Sharing and Multiparty Protocols with Honest Majority (Extended Abstract). In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing (STOC)*, pages 73–85, 1989.

[39] A. Shamir. How to Share a Secret. *Communications of the ACM*, 22(11):612–613, 1979.

[40] K. Srinathan and C. P. Rangan. Efficient Asynchronous Secure Multiparty Distributed Computation. In *Proceedings of the First International Conference on Cryptology in India (INDOCRYPT)*, pages 117–129, 2000.

[41] S. Toueg. Randomized Byzantine Agreements. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 163–178, 1984.