How to Design Fair Protocols in the Multi-Blockchain Setting

Sivanarayana Gaddam[*] Ranjit Kumaresan[†] Srinivasan Raghuraman[‡]
Rohit Sinha[§]

May 26, 2023

Abstract

Recently, there have been several proposals for secure computation with *fair output delivery* that require the use of a *bulletin board* abstraction (in addition to a trusted execution environment (TEE)). These proposals require all protocol participants to have read/write access to the bulletin board. These works envision the use of (public or permissioned) blockchains to implement the bulletin board abstractions. With the advent of consortium blockchains which place restrictions on who can read/write contents on the blockchain, it is not clear how to extend prior proposals to a setting where (1) not all parties have read/write access on a single consortium blockchain, and (2) not all parties prefer to post on a public blockchain.

In this paper, we address the above by showing the first protocols for fair secure computation in the *multi-blockchain* setting. More concretely, in a n-party setting where at most $t < n$ parties are corrupt, our protocol for fair secure computation works as long as (1) t parties have access to a TEE (e.g., Intel SGX), and (2) each of the above t parties are on some blockchain with each of the other parties. Furthermore, only these t parties need write access on the blockchains.

In an optimistic setting where parties behave honestly, our protocol runs completely off-chain.[1]

Keywords: Fair exchange, contract signing, secure multiparty computation, trusted execution environment, blockchain.

[*]. This work was done in part while the author was at Visa Research.

[†]Visa Research, rakumare@visa.com.

[‡]Visa Research, srraghur@visa.com.

[§]. This work was done in part while the author was at Visa Research.

Contents

1 Introduction

Secure multiparty computation (MPC) allows a set of mutually distrusting parties to perform a joint computation on their inputs that reveals only the final output and nothing else. Showing feasibility [Yao86, GMW87, BGW88, CCD88, RB89] of this seemingly impossible task has been a major achievement for modern cryptography. Today, secure computation is widely believed to be practical, and is seen as an important technology that is likely to enable, among other things, new business applications resulting from secure data sharing [MZ17].

While secure computation indeed provides the best possible notion of privacy, correctness, and security, it cannot provide *fairness* in settings where a majority of the participants are corrupt [Cle86]. For instance, in a two-party setting, a malicious party can *abort* a secure computation protocol *after* getting its output, leaving the other (honest) party no recourse to getting its output. Addressing this deficiency of secure computation is critical, as it is not appealing for, say a business entity to engage in a secure computation protocol with its partners/competitors where it may not learn the final outcome (while they might).

In the light of Cleve's impossibility result [Cle86], several lines of research have investigated the possibility of achieving fairness via non-standard security notions.[2] *Partial fairness* [GK10, BLOO11] provides a relaxed notion of fairness in secure computation where fairness may be breached with some parameterizable (inverse polynomial) probability. *Gradual release mechanisms* [Pin03, GMPY11] and "Δ-fairness" [PST17] study models where the honest party can recover the final output using additional computational resources. *Secure computation with penalties* [ADMM14, ADMM16, BK14, KMB15] study models where the honest party may be monetarily compensated (via cryptocurrencies) in the event that fairness is breached.

Other lines of work have investigated augmenting the computation model to overcome the problem of fairness. Examples include *optimistic fair exchange* [ASW97, ASW00, KL12] where the goal is to minimize the use of a trusted third party to restore fairness. More recently, [CGJ$^+$17] showed that the use of a *blockchain* modeled as a *bulletin board* can help in achieving fair secure computation. More concretely, they rely on the interpretation that blockchain can provide a *proof of publication* [KGM19] for the content posted on it. Then assuming either the existence of a witness encryption scheme or the existence of a *trusted execution environment* (TEE), along with a (public or permissioned) blockchain where every party has read/write access on the blockchain. Following their work, [SGK19] show how to minimize the use of TEE. Specifically, they show that only t parties need to possess a TEE in an n-party setting where at most $t < n$ parties are corrupt. In particular, in a 2-party setting, only 1 processor needs to possess a TEE. Recently, [KRS20] proposed a two-party primitive "synchronizable exchange" that is complete for n-party fair computation.

Multiblockchain settings. Our work follows the line of research in [CGJ$^+$17, PS19, SGK19] and shows constructions of fair protocols using TEEs and blockchains in new settings. Concretely, we investigate settings where not all parties have read/write access on a single common blockchain. Such a setting might seem unnatural given the existence of public blockchains such as Bitcoin [Nak08], where there is no restriction on who can read/write. However, we envision business settings, where for compliance, legal, or regulatory reasons, businesses may prefer not to use a public blockchain. Such a scenario may not be far fetched as we already see consortium blockchains gaining rapid popularity and adoption [AAB$^+$19, Mor16, RAA$^+$19]. Going forward, it seems likely

[2] [GHKL11, GK09, Ash14, ABMO15] show *restricted* classes of functions for which the standard notion of fair secure computation is possible.

that such consortiums will co-exist with one another, and will likely have overlapping members. This is precisely the type of setting that we target in this work. Extending the results of [CGJ+17, SGK19] to this new setting turns out to be quite non-trivial. Specifically, these works rely on a blockchain to provide a *proof of publication* that will be accepted by a TEE and its host.[3] Unfortunately, dealing with proofs of publication from different blockchains is a problem. We elaborate below.

Can we use proofs of publication *across blockchains?* Since membership in a consortium may cost significant fees, it is expected that members of a consortium C_1 may be restricted from performing read/write operations on the blockchain of consortium C_2 if they are not members of C_2. Without any visibility into the blockchain of C_2, members of C_1 may not trust C_2 to carry out a reliable blockchain read/write. While members within C_2 may trust that their blockchain B_2 is maintained properly, there is no reason for P_1, a member of C_1 but not C_2 to trust that this is the case.

More concretely, let us focus on the circumstances under which $P_1 \in C_1 \setminus C_2$ may accept a proof of publication on B_2. A proof of publication in a permissioned blockchain with m participants would likely be $t+1$ signatures from participating parties, where $t < m/3$ (or $t < m/2$) may be the threshold for a distributed consensus protocol (e.g., Paxos) that is used to maintain the permissioned blockchain. This means that for P_1 to trust a proof of publication must essentially trust that there are at most t colluding parties in C_2. Since P_1 has no control over membership constraints in C_2, it may not find such a trust assumption reasonable. Similar issues apply also to other consensus methods (e.g., proof of work, proof of stake) since these typically rely on some form of threshold assumption on the consortium. A more serious problem is that P_1 may not be able to obtain a (valid) proof of publication on B_2 which is otherwise accessible to every member of C_2. Next, we provide a high level overview of the protocols in [CGJ+17, SGK19], and show how the above issues translate into concrete problems in the construction of fair protocols.

Fairness in a single blockchain setting. Consider a setting with parties P_1, P_2, P_3 each possessing a TEE and each having access to a single blockchain B. Let us refer to T_j as the TEE hosted by P_j. We assume that each P_i has a secure channel established with T_j (i.e., P_j cannot read those messages). In the first step, each T_i sends their host input to T_j, following which T_j can compute the function output on the provided inputs. However, T_j cannot yet release the outputs to its host P_j as there may be some T_j that has not received all the inputs. To ensure that all TEEs received the inputs, the protocol in [CGJ+17] asks each T_j to (1) post a token (of a specific form) on the blockchain indicating that all inputs were received, and (2) receiving from its host P_j all 3 tokens from T_1, T_2, T_3 with their respective proofs of publication on B, and (3) only then, release the function output to P_j. To see why the above protocol is fair, note that if some T_j released the function output to P_j, then all 3 tokens must have been recorded on B. This is because, by assumption, the proof of publication π_v of posting a value v is unforgeable. That is, it is (computationally) infeasible for a party to compute π_v without posting v on the blockchain. This in turn implies that every T_i obtained inputs from all participants. Furthermore, since all 3 tokens are recorded on B, these are available to T_i (through P_i), following which T_i will release the final output to P_i.

Problems in extending to a multiblockchain setting. For concreteness, suppose there are three consortiums $C_{\{1,2\}}, C_{\{2,3\}}, C_{\{1,3\}}$ such that $P_i, P_j \in C_{\{i,j\}}$ but $P_k \notin C_{\{i,j\}}$ for $k \notin \{i,j\}$. Suppose $C_{\{i,j\}}$ maintain blockchain $B_{\{i,j\}}$. It follows from the discussion above that P_k for $k \notin \{i,j\}$ may

[3] Consortium blockchains may rely on TEEs to enable, among other things, private computations [SGK19, RAA+19].

not share the trust assumptions on $C_{\{i,j\}}$ as P_i and P_j, and consequently may not trust proofs of publication on $B_{\{i,j\}}$. Alternatively, P_k may not have (read) access to proofs on $B_{\{i,j\}}$, and consequently will not get the output. Therefore, a protocol which relies on read/write on a single blockchain, say $B_{\{i,j\}}$ will not work. Given this, one may be tempted to use proofs from multiple blockchains. For instance, one may design a protocol where T_k relies on proofs from both $B_{\{i,k\}}$ and $B_{\{j,k\}}$ to release the output. With this strategy, the difficulty comes in ensuring that tokens are recorded on all three blockchains (so that each TEE can make progress) or none. To see why, consider the following sequence of events. Suppose P_k is honest and as above, T_k records its token on $B_{\{i,k\}}$ and $B_{\{j,k\}}$. Now if P_i and P_j behave honestly like P_k above, then all three parties will have access to all 3 tokens. However, if P_i does not record its token on $B_{\{i,k\}}$, but records it on $B_{\{i,j\}}$. Recall that P_k does not have visibility into $B_{\{i,j\}}$ which is the only blockchain that has P_i's token recorded on it. Now if P_j records its token on $B_{\{i,j\}}$ and $B_{\{j,k\}}$, then note that P_j has access to all 3 tokens while P_k does not. Thus, loosely speaking, the problem reduces to a simultaneous writing of the tokens on 3 blockchains, and is somewhat similar to the original problem of fair exchange that we started with. Perhaps surprisingly, we show that designing fair protocols is indeed possible in the multiblockchain setting described above. In fact, our protocols work in the above setting when only two out of the 3 parties have a TEE (but every pair has a common blockchain).

Our results. We design protocols for fair secure computation in the multiblockchain setting. More concretely, in a n-party setting where at most $t < n$ parties are corrupt, our protocol for fair secure computation works as long as (1) t parties have access to a TEE, and (2) each of the above t parties are on some blockchain with each of the other parties. Furthermore, only these t parties need write access to the blockchains.[4] Our protocols use the blockchain only when participants behave maliciously, i.e., in an *optimistic* setting our protocol can be entirely run *off-chain*.

Remark. To better understand our result, consider a setting with 3 *permissioned* consortiums C_1, C_2, C_3. Suppose consortium C_i has m_i members and members of C_i assume a corruption threshold $t_i < m_i/10$ within C_i. Now suppose there is a centralized adversary $\mathcal{A}$ that controls and co-ordinates all the corrupt parties across consortiums. Further, assume $\mathcal{A}$ corrupts a subset A_i in C_i with $|A_i| = t_i$. In such a setting, we have $|\cup_i A_i| < |\cup_i C_i|/3$, in which case one may use any standard honest majority MPC protocol involving members of all consortiums C_1, C_2, C_3 to obtain a fair protocol.[5] We emphasize that our protocol works in the above setting even when $\mathcal{A}$ may corrupt more than t_i within some consortium, or even when $\mathcal{A}$ corrupts all members within a given consortium (in which case an honest majority may not exist in the union of all consortiums).

Our method. Technically, we rely on the recently proposed "synchronizable fair exchange" $\mathcal{F}_{\mathsf{SyX}}$ abstraction [KRS20]. $\mathcal{F}_{\mathsf{SyX}}$ is a 2-party primitive, and is shown in [KRS20] to be complete for secure *multiparty* computation with fairness where all parties are pairwise connected by independent instances of $\mathcal{F}_{\mathsf{SyX}}$. More concretely, in a setting as discussed above with 3 parties P_1, P_2, P_3, if for all i, j, parties P_i and P_j are connected to an instance of $\mathcal{F}_{\mathsf{SyX}}$, then the result of [KRS20] yields a 3-party fair secure computation protocol. Given the above, it is natural to try and apply [KRS20] to our multiblockchain setting to solve the fair computation problem. Unfortunately, [KRS20] do

[4] Note that the parameters t, n above are independent of the size of the consortiums, i.e., the number of participants in the consensus protocols for maintaining the blockchain.

[5] As an aside, it is worthwhile to note that designing fair protocols in a *single permissioned blockchain* setting with a threshold assumption on the number of corrupt parties is somewhat trivial since the threshold constraints for consensus (typically, $t_i < m_i/3$) admit completely fair MPC protocols. We emphasize that the works [CGJ+17, SGK19] yield fair protocols even with a *public* blockchain.

not provide an implementation of $\mathcal{F}_{\mathsf{SyX}}$. One of the main contributions of our work is to provide the first concrete implementation of $\mathcal{F}_{\mathsf{SyX}}$. Our implementation of $\mathcal{F}_{\mathsf{SyX}}$ between a pair of parties P_i, P_j, perhaps unsurprisingly requires both parties (1) to each possess a TEE, and (2) to share a common blockhain. Given this, and applying the protocol of [KRS20], we obtain a protocol for fair secure computation in the 3-party multiblockchain setting discussed above. Similarly, we can solve n-party fair secure computation in the multiblockchain setting as long as each pair of parties share a common blockchain.[6] Note that the above idea does not extend to settings where only $t < n$ parties possess a TEE. This is because now there may be pairs of parties where one party does not possess a TEE, and thus this pair may not be able to implement an instance of $\mathcal{F}_{\mathsf{SyX}}$ between them. To derive results in this setting, we work with a restricted variant of $\mathcal{F}_{\mathsf{SyX}}$ which we can implement when only one of the two parties possesses a TEE. Note that even with this variant, there may be pairs of parties where neither possesses a TEE, and thus may not be able to implement an instance of $\mathcal{F}_{\mathsf{SyX}}$ between them. In this setting, we design new fair multiparty protocols in a $\mathcal{F}_{\mathsf{SyX}}$-hybrid model where not all pairs of parties are connected by an $\mathcal{F}_{\mathsf{SyX}}$ instance.

Interpretation of our abstraction. As an analogy, consider a 3-party implementation of information-theoretic (unfair) secure computation in the OT-hybrid model [IPS08,Kil88]. Note that each of the 3 OT instances may be implemented under different cryptographic assumptions. For concreteness, suppose the OT instance (1) between P_1, P_2 is implemented under DDH, (2) between P_2, P_3 is implemented under RSA, and (3) between P_1, P_3 is implemented under LWE. Furthermore assume that none of the parties believe that all of DDH, RSA, LWE assumptions hold. That is, P_1 may believe that RSA is broken, P_2 may believe that LWE is broken, and P_3 may believe that DDH is broken. Still the information-theoretic MPC guarantees that security and privacy is guaranteed, say for honest P_1, even when RSA is indeed broken. That is, as long as the honest party's assumption is correct, its inputs remain private, and the MPC guarantees hold for that party. However, suppose DDH is broken, then honest P_1's inputs may leak to the adversary (controlling P_2). Likewise, in our multiblockchain setting, we abstract away the consortiums (and their blockchains), and instead replace these with pairwise $\mathcal{F}_{\mathsf{SyX}}$ instances. Suppose in our 3 party setting, if (honest) P_1 believes that P_2, P_3 blockchain $B_{\{2,3\}}$ is completely compromised, then P_1's guarantees still hold if indeed $B_{\{2,3\}}$'s security is breached. However, if $B_{\{1,3\}}$ or $B_{\{1,2\}}$ is breached, then P_1's guarantees with respect to fairness do not hold (but the TEE/unfair MPC guarantees that P_1's inputs remain private).

2 Overview

To understand our contributions, we first describe the synchronizable exchange primitive $\mathcal{F}_{\mathsf{SyX}}$ from [KRS20]. Then, we describe our variant, and its implementation, and finally our new protocols. Finally, we discuss modifications to our protocol that allow preprocessing, and in particular minimize the use of blockchain.

[6] To clarify, our protocols apply to a n-party setting (1) with $\binom{n}{2}$ distinct blockchains, or (2) where all n parties have access to one common blockchain (a la [CGJ+17,SGK19]), or (3) a setting with $1 \leq b \leq \binom{n}{2}$ blockchains where each pair of parties both have read/write access on one of the b blockchains.

2.1 Implementing Synchronizable Exchange

Synchronizable exchange $\mathcal{F}_{\mathsf{SyX}}$ [KRS20] is a two-party symmetric primitive which is reactive and works in two phases called *load* and *trigger*. In the load phase, parties submit their private inputs x_1, x_2 along with public inputs $(f_1, f_2, \phi_1, \phi_2)$. Here f_1, f_2 are 2-output functions, and ϕ_1, ϕ_2 are boolean predicates. Upon receiving these inputs, $\mathcal{F}_{\mathsf{SyX}}$ computes $f_1(x_1, x_2)$ and delivers the respective outputs to both parties. Next, in the trigger phase, which can be initiated at any later time after the load phase, party P_i can send a "witness" w_i to $\mathcal{F}_{\mathsf{SyX}}$ following which $\mathcal{F}_{\mathsf{SyX}}$ checks if $\phi_i(w_i) = 1$. If that is indeed the case, then $\mathcal{F}_{\mathsf{SyX}}$ computes $f_2(x_1, x_2, w_i)$ and delivers the respective outputs along with w_i to both parties in a fair manner. Next, we describe our variant.

Synchronizable exchange with one-sided trigger. Here, we restrict $\mathcal{F}_{\mathsf{SyX}}$ by giving only one designated party, say P_i, the ability to trigger $\mathcal{F}_{\mathsf{SyX}}$. This is done easily by setting $\phi_j \equiv 0$, thereby ensuring that P_j can never trigger $\mathcal{F}_{\mathsf{SyX}}$. Note that P_i will still need to provide a valid witness that satisfies ϕ_i. Next, we show how to implement this variant when only P_i possesses a TEE.

Implementing $\mathcal{F}_{\mathsf{SyX}}$ with one-sided trigger. We now sketch the $\mathcal{F}_{\mathsf{SyX}}$ implementation with parties P_i and P_j both of which have access to a blockchain B, but only P_i possesses a TEE. First, P_i and P_j supply their inputs to T_i. At this point T_i computes $y_1 = f_1(x_1, x_2)$, and outputs $e_1 = \mathsf{Enc}(pk_j, y_1)$ where pk_j is the public key of P_j. Following this, P_i posts e_1 on the blockchain, and obtain the corresponding proof of publication π_1. Then P_i feeds π_1 to T_i which then releases the output y_1 to P_i. Note that P_j can recover y_1 by reading B and decrypting e_1 with its secret key sk_j. For the trigger phase, suppose P_i has a valid witness w_i. Then P_i feeds w_i to T_i, which verifies if $\phi_i(w_i) = 1$, and if so computes $y_2 = f_2(x_1, x_2, w_i)$, and outputs $e_2 = \mathsf{Enc}(pk_j, w_i \| y_2)$ to P_i. As before, P_i posts e_2 on B, gets the proof of publication π_2, then feeds π_2 to T_i which outputs y_2 to P_i. P_j reads e_2 from B, and decrypts it to get w_i and y_2.

It is easy to see that the trigger phase can be initiated only by P_i (hence "one-sided trigger") since only T_i can compute $f_2(x_1, x_2, w_i)$. Next, note that the outputs of both f_1 and f_2 are delivered to both parties in a fair manner. In particular, note that P_1 cannot obtain the output of f_1 (resp. f_2) without posting e_1 (resp. e_2) on the blockchain. This is because T_i reveals $y_1 = f_1(x_1, x_2)$ (resp. $y_2 = f_2(x_1, x_2, w_i)$) only after obtain π_1 (resp. π_2) from the blockchain. This in turn ensures that e_1 (resp. e_2) was posted on the blockchain and hence available for P_j to decrypt and obtain y_1 (resp. y_2). Note that P_i can prevent the evaluation of f_1 (or f_2), but as we described above, if P_i indeed gets the outputs of f_1 (or f_2), then P_j will get the output as well. Also, note that load phase may be completed, but (a corrupt) P_i may not trigger even if instructed by the higher level protocol. On the other hand, note that an honest P_i's trigger will always result in P_i learning the output, and in particular, there is no way a corrupt P_j can prevent P_i from learning the output of f_1 or f_2. Next, we sketch how to extend these ideas to implement $\mathcal{F}_{\mathsf{SyX}}$ where either party can trigger.

Implementing $\mathcal{F}_{\mathsf{SyX}}$. Now, we assume that both parties P_i, P_j possess a TEE T_i, T_j respectively. Further, assume that T_i and T_j share a symmetric key ek. As before, the protocol begins by letting P_i and P_j supply their inputs to T_i and T_j. (We omit details on standard techniques that ensure that parties submit the same inputs to both TEEs.) Then, we let T_i and T_j each post a token on the blockchain indicating that they received the inputs. The TEEs do not proceed with the load phase unless they receive two proofs of publication of both tokens from the blockchain. Given these proofs, both T_i and T_j locally output (respectively to P_i and P_j) the value $f_1(x_1, x_2)$, and terminate the load phase. Next, we describe the trigger phase when P_i wishes to trigger (the case when P_j wishes to trigger is analogous). First, P_i provides a witness w_i to T_i, following which T_i checks if $\phi_i(w_i) = 1$, then outputs $e = \mathsf{Enc}(ek, w_i \| f_2(x_1, x_2, w_i))$ to P_i. P_i then posts e on the blockchain to obtain a proof of publication π, which it then sends to T_i. Upon receiving the proof π, T_i outputs $f_2(x_1, x_2, w_i)$ to P_i, and terminates the trigger phase. Upon seeing the token e on the blockchain, P_j sends e, π to T_j, which then first checks if π is a valid proof of publication, then decrypts e using ek to obtain $w_i, f_2(x_1, x_2, w_i)$, and then checks if $\phi_i(w_i) = 1$, and if so finally outputs $f_2(x_1, x_2, w_i)$ to P_j.

As before, the outputs of f_1 and f_2 are delivered to both parties in a fair manner. More concretely, P_i cannot obtain the output of f_1 unless tokens from both T_i, T_j indicating that they received the inputs are recorded on the blockchain. When these tokens are recorded on the blockchain, there is no way for P_i to prevent P_j from reading these tokens and submitting the tokens along with proofs to T_j which results in P_j obtaining the output of f_1. Now, suppose P_i obtains the output of f_2, say by providing the trigger witness w_i. Then we argue that P_i has no way of preventing P_j from learning the output of f_2. To see why, note that P_i needs to provide proof π that e was posted on the blockchain. Since it is infeasible for P_i to obtain this proof without posting e on the blockchain, it follows that e can be read by P_j, following which P_j can feed e to T_j and obtain the final output. It should also be clear from the description above that the triggering party will always obtain the output if it behaves honestly. That is, if P_i initiates the trigger phase, then there is no way for a corrupt P_j to prevent P_i from learning the output of f_2.

Formal implementation. In the main body of the paper, we provide a formal description of the protocols above. Our implementation of $\mathcal{F}_{\mathsf{SyX}}$ (and its variant with one-sided trigger) will be described in the $(\mathcal{G}_{\mathsf{att}}, \mathcal{F}_{\mathsf{BB}}, \mathcal{G}_{\mathsf{acrs}})$-hybrid model, where (1) $\mathcal{G}_{\mathsf{att}}$ is a global ideal functionality described in [PST17] that captures attested executions, and (2) $\mathcal{F}_{\mathsf{BB}}$ is the ideal blockchain functionality as described in several prior works, and in particular provides an interface for obtaining proofs of publication as in [CGJ+17], and (3) $\mathcal{G}_{\mathsf{acrs}}$ is the global ideal functionality for *augmented common reference string* as described and used in [PST17].

2.2 Fair Protocols in the Multiblockchain Setting

In this section, we provide a sketch of our new protocols for achieving fairness in the multiblockchain setting. Recall that in a n-party setting where at most $t < n$ parties are corrupt, we assume that (1) at least t parties have access to a TEE, and (2) each of the above t parties are on some blockchain with each of the other parties. To construct fair protocols in this setting, we first make the following transformation: for every pair of parties P_i, P_j, we add an $\mathcal{F}_{\mathsf{SyX}}$ instance between them if and only if (1) at least one of P_i, P_j possesses a TEE, and (2) if P_i and P_j are on some common blockchain. An $\mathcal{F}_{\mathsf{SyX}}$ instance between P_i and P_j is one-sided iff exactly only of P_i, P_j possesses a TEE. With the above transformation we have abstracted away both TEEs and blockchains, and are in a setting with n parties some of which are connected by $\mathcal{F}_{\mathsf{SyX}}$ instances. For the sake of simplicity, we represent this setting with an "$\mathcal{F}_{\mathsf{SyX}}$-digraph" G, where the vertices represent the n parties, and edges represent $\mathcal{F}_{\mathsf{SyX}}$ instances. More concretely, if t parties possess a TEE, then G consists of $\mathcal{O}(nt)$ edges. Specifically, there is a directed edge between i and j in G if (1) P_i possesses a TEE, and (2) if P_i and P_j share a common blockchain.

Note that the prior work of [KRS20] showed fair protocols when the "$\mathcal{F}_{\mathsf{SyX}}$-digraph" is complete, i.e., with $2\binom{n}{2}$ edges. Now, we describe a fair protocol when G is of the form above, and in particular when G is not a complete digraph. As in prior work [GIM+10, CGJ+17, SGK19, KRS20], we reduce fair secure computation to fair reconstruction of a secret sharing scheme. Specifically, we let the parties run an (unfair) MPC protocol for a function f that computes the function output y, then computes secret shares of the function output $\{y_i\}_{i\in[n]}$, and then computes commitments on these secret shares. Denote these commitments by $\mathbf{c} = (c_1, \ldots, c_n)$. The MPC protocol outputs to party P_i the values $y_i, \mathbf{c}$. Note that if some honest party does not obtain its output from the (unfair) MPC protocol, then all parties terminate (and no one gets the final output y). Now to get a fair evaluation of f, we only need to ensure that either all parties learn all the commitment openings or none of them learns all the openings.

Next, we describe how we use the $\mathcal{F}_{\mathsf{SyX}}$ instances to achieve fair reconstruction of y. Without loss of generality assume that $P_1, \ldots, P_t$ possess a TEE and $P_{t+1}, \ldots, P_n$ do not. Therefore, when $i \le t$, we have that $(i, j) \in G$ for all $j \in [n]$ (and these are the only edges in G). Consider the $\mathcal{F}_{\mathsf{SyX}}$ instance associated with $(i, j) \in G$ with $i \le t$ and $i < j$. We will set up this instance such that (1) it can be triggered only in round $j+1$, and (2) the predicate ϕ_i associated with this instance checks for valid openings of $c_1, \ldots, c_{j-1}$, and (3) upon trigger, the value y_j is released to both parties.

To reconstruct y, in round $j + 1$ for $2 \le j \le n$, each party i with $i \le t$ and $i < j$, triggers (if possible) the $\mathcal{F}_{\mathsf{SyX}}$ instance associated with $(i, j) \in G$. Finally, in round $n + 2$, if any honest party obtained all the openings, i.e., the values $y_1, \ldots, y_n$, then they broadcast all these openings to all parties. This completes the overview of the protocol. Next, we sketch why the above steps suffice. Suppose the adversary learns y at the end of the protocol, then we need to show that all honest parties learn y too. Let P_j be honest such that for $k < j$, party P_k is corrupt. To learn y, the adversary must learn y_j by triggering an $\mathcal{F}_{\mathsf{SyX}}$ instance associated with j. This consequently means that P_j would learn all values $y_1, \ldots, y_j$. If $j \le t$, then for all $k > j$, in round $k + 1$, party P_j can trigger the (j, k) $\mathcal{F}_{\mathsf{SyX}}$ instance using the witnesses $y_1, \ldots, y_{k-1}$ to learn y_k. Then, in round $n + 2$, party P_j would broadcast all openings to all honest parties, and therefore all honest parties would obtain the final output. If $j > t$, then P_n must be honest since there are at most t corrupt parties. To obtain y_n (and consequently the final output y), the adversary needs some corrupt P_i with $i \le t$ to trigger the $\mathcal{F}_{\mathsf{SyX}}$ instance associated with $(i, n) \in G$. If P_i triggers this $\mathcal{F}_{\mathsf{SyX}}$ instance, then honest P_n would learn the openings $y_1, \ldots, y_{n-1}$, and therefore would know all openings, and it would broadcast these openings in round $n + 2$, leading to all honest parties obtaining the final output.

3 Preliminaries

3.1 Notation and definitions

For $n \in \mathbb{N}$, let $[n] = \{1, 2, \ldots, n\}$. Let $\lambda \in \mathbb{N}$ denote the security parameter. Symbols in with an arrow over them such as $\overrightarrow{a}$ denote vectors. By a_i we denote the i-th element of the vector $\overrightarrow{a}$. For a vector $\overrightarrow{a}$ of length $n \in \mathbb{N}$ and an index set $I \subseteq [n]$, we denote by $\overrightarrow{a}|_I$ the vector consisting of (ordered) elements from the set $\{a_i\}_{i\in I}$. By $\mathrm{poly}(\cdot)$, we denote any function which is bounded by a polynomial in its argument. An algorithm $\mathcal{T}$ is said to be PPT if it is modeled as a probabilistic Turing machine that runs in time polynomial in λ. Informally, we say that a function is negligible, denoted by negl, if it vanishes faster than the inverse of any polynomial. If S is a set, then $x \xleftarrow{\$} S$ indicates the process of selecting x uniformly at random over S (which in particular assumes that S can be sampled efficiently). Similarly, $x \xleftarrow{\$} \mathcal{A}(\cdot)$ denotes the random variable that is the output of a randomized algorithm $\mathcal{A}$. Let $\mathcal{X}, \mathcal{Y}$ be two probability distributions over some set S. Their *statistical distance* is

$$\mathbf{SD}(\mathcal{X}, \mathcal{Y}) \stackrel{\text{def}}{=} \max_{T \subseteq S} \{|\Pr[\mathcal{X} \in T] - \Pr[\mathcal{Y} \in T]|\}$$

We say that $\mathcal{X}$ and $\mathcal{Y}$ are ϵ-close if $\mathbf{SD}(\mathcal{X}, \mathcal{Y}) \leq \epsilon$ and this is denoted by $\mathcal{X} \approx_\epsilon \mathcal{Y}$. We say that $\mathcal{X}$ and $\mathcal{Y}$ are identical if $\mathbf{SD}(\mathcal{X}, \mathcal{Y}) = 0$ and this is denoted by $\mathcal{X} \equiv \mathcal{Y}$.

3.2 Secure Computation

We recall most of the definitions regarding secure computation from [GHKL11] and [CL17]. We present them here for the sake of completeness and self-containedness. Consider the scenario of n parties $P_1, \ldots, P_n$ with private inputs $x_1, \ldots, x_n \in \mathcal{X}$[7]. We denote $\overrightarrow{x} = (x_1, \ldots, x_n) \in \mathcal{X}^n$.

3.2.1 Functionalities

A functionality f is a randomized process that maps n-tuples of inputs to n-tuples of outputs, that is, $f : \mathcal{X}^n \to \mathcal{Y}^n$[8]. We write $f = (f^1, \ldots, f^n)$ if we wish to emphasize the n outputs of f, but stress that if $f^1, \ldots, f^n$ are randomized, then the outputs of $f^1, \ldots, f^n$ are correlated random variables.

3.2.2 Adversaries

We consider security against *static t-threshold adversaries*, that is, adversaries that corrupt a set of at most t parties, where $0 \leq t < n$[9]. We assume the adversary to be malicious. That is, the corrupted parties may deviate arbitrarily from an assigned protocol.

3.2.3 Model

We assume the parties are connected via a fully connected point-to-point network; we refer to this model as the $\mathsf{point\text{-}to\text{-}point}$ model. We sometimes assume that the parties are given access to a physical broadcast channel (defined in Section 3.7)[10] in addition to the point-to-point network; we refer to this model as the $\mathsf{broadcast}$ model. The communication lines between parties are assumed to be ideally authenticated and private (and thus an adversary cannot read or modify messages sent between two honest parties). Furthermore, the delivery of messages between honest parties is guaranteed. We sometimes assume the parties are connected via a fully pairwise connected network of oblivious transfer channels (defined in Section 3.6)[11] in addition to a fully connected point-to-point network; we refer to this model as the $\mathsf{OT\text{-}network}$ model. We sometimes assume that the parties are given access to a physical broadcast channel in addition to the complete pairwise oblivious transfer network and a fully connected point-to-point network; we refer to this model as the $\mathsf{OT\text{-}broadcast}$ model[12].

[7] Here we have assumed that the domains of the inputs of all parties is $\mathcal{X}$ for simplicity of notation. This can be easily adapted to consider setting where the domains are different.

[8] Here we have assumed that the domains of the outputs of all parties is $\mathcal{Y}$ for simplicity of notation. This can be easily adapted to consider setting where the domains are different.

[9] Note that when $t = n$, there is nothing to prove.

[10] This can also be viewed as working in the $\mathcal{F}_{\mathsf{bc}}$-hybrid model. See Section 3.3.

[11] This can also be viewed as working in the $\mathcal{F}_{\mathsf{OT}}$-hybrid model. See Section 3.3.

[12] This can also be viewed as working in the $(\mathcal{F}_{\mathsf{bc}}, \mathcal{F}_{\mathsf{OT}})$-hybrid model. See Section 3.3.

3.2.4 Protocol

An n-party protocol for computing a functionality f is a protocol running in polynomial time and satisfying the following functional requirement: if for every $i \in [n]$, party P_i begins with private input $x_i \in \mathcal{X}$, then the joint distribution of the outputs of the parties is statistically close to $(f^1(\overrightarrow{x}), \ldots, f^n(\overrightarrow{x}))$. We assume that the protocol is executed in a synchronous network, that is, the execution proceeds in rounds: each round consists of a *send phase* (where parties send their message for this round) followed by a *receive* phase (where they receive messages from other parties). The adversary, being malicious, is also *rushing* which means that it can see the messages the honest parties send in a round, before determining the messages that the corrupted parties send in that round.

3.2.5 Security with Guaranteed Output Delivery

The security of a protocol is analyzed by comparing what an adversary can do in a real protocol execution to what it can do in an ideal scenario that is secure by definition. This is formalized by considering an *ideal* computation involving an incorruptible *trusted party* to whom the parties send their inputs. The trusted party computes the functionality on the inputs and returns to each party its respective output. Loosely speaking, a protocol is secure if any adversary interacting in the real protocol (where no trusted party exists) can do no more harm than if it were involved in the above-described ideal computation.

Execution in the ideal model. The parties are $P_1, \ldots, P_n$, and there is an adversary $\mathcal{A}$ who has corrupted at most t parties, where $0 \le t < n$. Denote by $\mathcal{I} \subseteq [n]$ the set of indices of the parties corrupted by $\mathcal{A}$. An ideal execution for the computation of f proceeds as follows:

- **Inputs:** $P_1, \ldots, P_n$ hold their private inputs $x_1, \ldots, x_n \in \mathcal{X}$; the adversary $\mathcal{A}$ receives an auxiliary input z.
- **Send inputs to trusted party:** The honest parties send their inputs to the trusted party. The corrupted parties controlled by $\mathcal{A}$ may send any values of their choice. Denote the inputs sent to the trusted party by $x'_1, \ldots, x'_n$.
- **Trusted party sends outputs:** If $x'_i \notin \mathcal{X}$ for any $i \in [n]$, the trusted party sets x'_i to some default input in $\mathcal{X}$. Then, the trusted party chooses r uniformly at random and sends $f^i(x'_1, \ldots, x'_n; r)$ to party P_i for every $i \in [n]$.
- **Outputs:** The honest parties output whatever was sent by the trusted party. The corrupted parties output nothing and $\mathcal{A}$ outputs an arbitrary (probabilistic polynomial-time computable) function of its view.

We let $\textsc{Ideal}^{\mathsf{g.d.}}_{f,\mathcal{I},\mathcal{S}(z)}(\overrightarrow{x}, \lambda)$ be the random variable consisting of the output of the adversary and the output of the honest parties following an execution in the ideal model described above.

Execution in the real model. We next consider the real model in which an n-party protocol π is executed by $P_1, \ldots, P_n$ (and there is no trusted party). In this case, the adversary $\mathcal{A}$ gets the inputs of the corrupted party and sends all messages on behalf of these parties, using an arbitrary polynomial-time strategy. The honest parties follow the instructions of π.

Let f be as above and let π be an n-party protocol computing f. Let $\mathcal{A}$ be a non-uniform probabilistic polynomial-time machine with auxiliary input z. We let $\textsc{Real}_{\pi,\mathcal{I},\mathcal{A}(z)}(x_1,\ldots,x_n,\lambda)$ be the random variable consisting of the view of the adversary and the output of the honest parties following an execution of π where P_i begins by holding x_i for every $i \in [n]$.

Security as emulation of an ideal execution in the real model. Having defined the ideal and real models, we can now define security of a protocol. Loosely speaking, the definition asserts that a secure protocol (in the real model) emulates the ideal model (in which a trusted party exists). This is formulated as follows.

Definition 1. *Protocol π is said to securely compute f with guaranteed output delivery if for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}$ in the real model, there exists a non-uniform probabilistic polynomial-time adversary $\mathcal{S}$ in the ideal model such that for every $\mathcal{I} \subseteq [n]$ with $|\mathcal{I}| \leq t$,*

$$\left\{\textsc{Ideal}^{\mathsf{g.d.}}_{f,\mathcal{I},\mathcal{S}(z)}(\vec{x},\lambda)\right\}_{\vec{x}\in\mathcal{X}^n, z\in\{0,1\}^*} \equiv \left\{\textsc{Real}_{\pi,\mathcal{I},\mathcal{A}(z)}(\vec{x},\lambda)\right\}_{\vec{x}\in\mathcal{X}^n, z\in\{0,1\}^*}$$

We will sometimes relax security to statistical or computational definitions. A protocol is statistically secure if the random variables $\textsc{Ideal}^{\mathsf{g.d.}}_{f,\mathcal{I},\mathcal{S}(z)}(\vec{x},\lambda)$ and $\textsc{Real}_{\pi,\mathcal{I},\mathcal{A}(z)}(\vec{x},\lambda)$ are statistically close, and computationally secure if they are computationally indistinguishable.

3.2.6 Security with Fairness

In this definition, the execution of the protocol can terminate in two possible ways: the first is when all parties receive their prescribed output (as in the case of guaranteed output delivery) and the second is when all parties (including the corrupted parties) abort without receiving output. The only change from the definition in Section 3.2.5 is with regard to the ideal model for computing f, which is now defined as follows:

Execution in the ideal model. The parties are $P_1,\ldots,P_n$, and there is an adversary $\mathcal{A}$ who has corrupted at most t parties, where $0 \leq t < n$. Denote by $\mathcal{I} \subseteq [n]$ the set of indices of the parties corrupted by $\mathcal{A}$. An ideal execution for the computation of f proceeds as follows:

- **Inputs:** $P_1,\ldots,P_n$ hold their private inputs $x_1,\ldots,x_n \in \mathcal{X}$; the adversary $\mathcal{A}$ receives an auxiliary input z.
- **Send inputs to trusted party:** The honest parties send their inputs to the trusted party. The corrupted parties controlled by $\mathcal{A}$ may send any values of their choice. In addition, there exists a special abort input. Denote the inputs sent to the trusted party by $x'_1,\ldots,x'_n$.
- **Trusted party sends outputs:** If $x'_i \notin \mathcal{X}$ for any $i \in [n]$, the trusted party sets x'_i to some default input in $\mathcal{X}$. If there exists an $i \in [n]$ such that $x'_i = \mathsf{abort}$, the trusted party sends $\perp$ to all the parties. Otherwise, the trusted party chooses r uniformly at random, computes $z_i = f^i(x'_1,\ldots,x'_n;r)$ for every $i \in [n]$ and sends z_i to P_i for every $i \in [n]$.
- **Outputs:** The honest parties output whatever was sent by the trusted party. The corrupted parties output nothing and $\mathcal{A}$ outputs an arbitrary (probabilistic polynomial-time computable) function of its view.

We let $\textsc{Ideal}^{\mathsf{fair}}_{f,\mathcal{I},\mathcal{S}(z)}(\overrightarrow{x},\lambda)$ be the random variable consisting of the output of the adversary and the output of the honest parties following an execution in the ideal model described above.

Definition 2. *Protocol π is said to securely compute f with fairness if for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}$ in the real model, there exists a non-uniform probabilistic polynomial-time adversary $\mathcal{S}$ in the ideal model such that for every $\mathcal{I} \subseteq [n]$ with $|\mathcal{I}| \leq t$,*

$$\left\{\textsc{Ideal}^{\mathsf{fair}}_{f,\mathcal{I},\mathcal{S}(z)}(\overrightarrow{x},\lambda)\right\}_{\overrightarrow{x}\in\mathcal{X}^n,z\in\{0,1\}^*} \equiv \left\{\textsc{Real}_{\pi,\mathcal{I},\mathcal{A}(z)}(\overrightarrow{x},\lambda)\right\}_{\overrightarrow{x}\in\mathcal{X}^n,z\in\{0,1\}^*}$$

We will sometimes relax security to statistical or computational definitions. A protocol is statistically secure if the random variables $\textsc{Ideal}^{\mathsf{fair}}_{f,\mathcal{I},\mathcal{S}(z)}(\overrightarrow{x},\lambda)$ and $\textsc{Real}_{\pi,\mathcal{I},\mathcal{A}(z)}(\overrightarrow{x},\lambda)$ are statistically close, and computationally secure if they are computationally indistinguishable.

3.2.7 Security with Fairness and Identifiable Abort

This definition is identical to the one for fairness, except that if the adversary aborts the computation, all honest parties learn the identity of one of the corrupted parties. The only change from the definition in Section 3.2.5 is with regard to the ideal model for computing f, which is now defined as follows:

Execution in the ideal model. The parties are $P_1,\ldots,P_n$, and there is an adversary $\mathcal{A}$ who has corrupted at most t parties, where $0 \leq t < n$. Denote by $\mathcal{I} \subseteq [n]$ the set of indices of the parties corrupted by $\mathcal{A}$. An ideal execution for the computation of f proceeds as follows:

- **Inputs:** $P_1,\ldots,P_n$ hold their private inputs $x_1,\ldots,x_n \in \mathcal{X}$; the adversary $\mathcal{A}$ receives an auxiliary input z.
- **Send inputs to trusted party:** The honest parties send their inputs to the trusted party. The corrupted parties controlled by $\mathcal{A}$ may send any values of their choice. In addition, there exists a special abort input. In case the adversary instructs P_i to send abort, it chooses an index of a corrupted party $i^* \in \mathcal{I}$ and sets $x'_i = (\mathsf{abort}, i^*)$. Denote the inputs sent to the trusted party by $x'_1,\ldots,x'_n$.
- **Trusted party sends outputs:** If $x'_i \notin \mathcal{X}$ for any $i \in [n]$, the trusted party sets x'_i to some default input in $\mathcal{X}$. If there exists an $i \in [n]$ such that $x'_i = (\mathsf{abort}, i^*)$ and $i^* \in \mathcal{I}$, the trusted party sends $(\perp, i^*)$ to all the parties. Otherwise, the trusted party chooses r uniformly at random, computes $z_i = f^i(x'_1,\ldots,x'_n;r)$ for every $i \in [n]$ and sends z_i to P_i for every $i \in [n]$.
- **Outputs:** The honest parties output whatever was sent by the trusted party. The corrupted parties output nothing and $\mathcal{A}$ outputs an arbitrary (probabilistic polynomial-time computable) function of its view.

We let $\textsc{Ideal}^{\mathsf{id\text{-}fair}}_{f,\mathcal{I},\mathcal{S}(z)}(\overrightarrow{x},\lambda)$ be the random variable consisting of the output of the adversary and the output of the honest parties following an execution in the ideal model described above.

Definition 3. *Protocol π is said to securely compute f with fairness and identifiable abort if for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}$ in the real model, there exists a non-uniform probabilistic polynomial-time adversary $\mathcal{S}$ in the ideal model such that for every $\mathcal{I} \subseteq [n]$ with $|\mathcal{I}| \leq t$,*

$$\left\{\textsc{Ideal}^{\mathsf{id\text{-}fair}}_{f,\mathcal{I},\mathcal{S}(z)}(\overrightarrow{x},\lambda)\right\}_{\overrightarrow{x}\in\mathcal{X}^n,z\in\{0,1\}^*} \equiv \left\{\textsc{Real}_{\pi,\mathcal{I},\mathcal{A}(z)}(\overrightarrow{x},\lambda)\right\}_{\overrightarrow{x}\in\mathcal{X}^n,z\in\{0,1\}^*}$$

We will sometimes relax security to statistical or computational definitions. A protocol is statistically secure if the random variables $\textsc{Ideal}^{\textsf{id-fair}}_{f,\mathcal{I},\mathcal{S}(z)}(\overrightarrow{x},\lambda)$ and $\textsc{Real}_{\pi,\mathcal{I},\mathcal{A}(z)}(\overrightarrow{x},\lambda)$ are statistically close, and computationally secure if they are computationally indistinguishable.

3.2.8 Security with Abort

This definition is the standard one for secure computation [Gol04] in that it allows *early abort*; that is, the adversary may receive its own output even though the honest party does not. However, if one honest party receives output, then so do all honest parties. Thus, this is the notion of *unanimous abort*. The only change from the definition in Section 3.2.5 is with regard to the ideal model for computing f, which is now defined as follows:

Execution in the ideal model. The parties are $P_1,\ldots,P_n$, and there is an adversary $\mathcal{A}$ who has corrupted at most t parties, where $0 \le t < n$. Denote by $\mathcal{I} \subseteq [n]$ the set of indices of the parties corrupted by $\mathcal{A}$. An ideal execution for the computation of f proceeds as follows:

- **Inputs:** $P_1,\ldots,P_n$ hold their private inputs $x_1,\ldots,x_n \in \mathcal{X}$; the adversary $\mathcal{A}$ receives an auxiliary input z.
- **Send inputs to trusted party:** The honest parties send their inputs to the trusted party. The corrupted parties controlled by $\mathcal{A}$ may send any values of their choice. In addition, there exists a special abort input. Denote the inputs sent to the trusted party by $x'_1,\ldots,x'_n$.
- **Trusted party sends outputs to the adversary:** If $x'_i \notin \mathcal{X}$ for any $i \in [n]$, the trusted party sets x'_i to some default input in $\mathcal{X}$. If there exists an $i \in [n]$ such that $x'_i = \mathsf{abort}$, the trusted party sends $\perp$ to all the parties. Otherwise, the trusted party chooses r uniformly at random, computes $z_i = f^i(x'_1,\ldots,x'_n;r)$ for every $i \in [n]$ and sends z_i to P_i for every $i \in \mathcal{I}$ (that is, to the adversary $\mathcal{A}$).
- **Trusted party sends outputs to the honest parties:** After receiving its output (as described above), the adversary either sends abort or $\mathsf{continue}$ to the trusted party. In the former case the trusted party sends $\perp$ to the honest parties, and in the latter case the trusted party send z_j to P_j for every $j \in [n] \setminus \mathcal{I}$.
- **Outputs:** The honest parties output whatever was sent by the trusted party. The corrupted parties output nothing and $\mathcal{A}$ outputs an arbitrary (probabilistic polynomial-time computable) function of its view.

We let $\textsc{Ideal}^{\textsf{abort}}_{f,\mathcal{I},\mathcal{S}(z)}(\overrightarrow{x},\lambda)$ be the random variable consisting of the output of the adversary and the output of the honest parties following an execution in the ideal model described above.

Definition 4. *Protocol π is said to securely compute f with abort if for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}$ in the real model, there exists a non-uniform probabilistic polynomial-time adversary $\mathcal{S}$ in the ideal model such that for every $\mathcal{I} \subseteq [n]$ with $|\mathcal{I}| \le t$,*

$$\left\{\textsc{Ideal}^{\textsf{abort}}_{f,\mathcal{I},\mathcal{S}(z)}(\overrightarrow{x},\lambda)\right\}_{\overrightarrow{x}\in\mathcal{X}^n, z\in\{0,1\}^*} \equiv \left\{\textsc{Real}_{\pi,\mathcal{I},\mathcal{A}(z)}(\overrightarrow{x},\lambda)\right\}_{\overrightarrow{x}\in\mathcal{X}^n, z\in\{0,1\}^*}$$

We will sometimes relax security to statistical or computational definitions. A protocol is statistically secure if the random variables $\textsc{Ideal}^{\textsf{abort}}_{f,\mathcal{I},\mathcal{S}(z)}(\overrightarrow{x},\lambda)$ and $\textsc{Real}_{\pi,\mathcal{I},\mathcal{A}(z)}(\overrightarrow{x},\lambda)$ are statistically close, and computationally secure if they are computationally indistinguishable.

3.2.9 Security with Identifiable Abort

This definition is identical to the one for abort, except that if the adversary aborts the computation, all honest parties learn the identity of one of the corrupted parties. The only change from the definition in Section 3.2.5 is with regard to the ideal model for computing f, which is now defined as follows:

Execution in the ideal model. The parties are $P_1, \ldots, P_n$, and there is an adversary $\mathcal{A}$ who has corrupted at most t parties, where $0 \leq t < n$. Denote by $\mathcal{I} \subseteq [n]$ the set of indices of the parties corrupted by $\mathcal{A}$. An ideal execution for the computation of f proceeds as follows:

- **Inputs:** $P_1, \ldots, P_n$ hold their private inputs $x_1, \ldots, x_n \in \mathcal{X}$; the adversary $\mathcal{A}$ receives an auxiliary input z.
- **Send inputs to trusted party:** The honest parties send their inputs to the trusted party. The corrupted parties controlled by $\mathcal{A}$ may send any values of their choice. In addition, there exists a special abort input. In case the adversary instructs P_i to send abort, it chooses an index of a corrupted party $i^* \in \mathcal{I}$ and sets $x'_i = (\mathsf{abort}, i^*)$. Denote the inputs sent to the trusted party by $x'_1, \ldots, x'_n$.
- **Trusted party sends outputs to the adversary:** If $x'_i \notin \mathcal{X}$ for any $i \in [n]$, the trusted party sets x'_i to some default input in $\mathcal{X}$. If there exists an $i \in [n]$ such that $x'_i = (\mathsf{abort}, i^*)$ and $i^* \in \mathcal{I}$, the trusted party sends $(\perp, i^*)$ to all the parties. Otherwise, the trusted party chooses r uniformly at random, computes $z_i = f^i(x'_1, \ldots, x'_n; r)$ for every $i \in [n]$ and sends z_i to P_i for every $i \in \mathcal{I}$ (that is, to the adversary $\mathcal{A}$).
- **Trusted party sends outputs to the honest parties:** After receiving its output (as described above), the adversary either sends (abort, i^*) where $i^* \in \mathcal{I}$, or $\mathsf{continue}$ to the trusted party. In the former case the trusted party sends $(\perp, i^*)$ to the honest parties, and in the latter case the trusted party send z_j to P_j for every $j \in [n] \setminus \mathcal{I}$.
- **Outputs:** The honest parties output whatever was sent by the trusted party. The corrupted parties output nothing and $\mathcal{A}$ outputs an arbitrary (probabilistic polynomial-time computable) function of its view.

We let $\textsc{Ideal}^{\mathsf{id\text{-}abort}}_{f,\mathcal{I},\mathcal{S}(z)}(\overrightarrow{x}, \lambda)$ be the random variable consisting of the output of the adversary and the output of the honest parties following an execution in the ideal model described above.

Definition 5. *Protocol π is said to securely compute f with identifiable abort if for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}$ in the real model, there exists a non-uniform probabilistic polynomial-time adversary $\mathcal{S}$ in the ideal model such that for every $\mathcal{I} \subseteq [n]$ with $|\mathcal{I}| \leq t$,*

$$\left\{\textsc{Ideal}^{\mathsf{id\text{-}abort}}_{f,\mathcal{I},\mathcal{S}(z)}(\overrightarrow{x}, \lambda)\right\}_{\overrightarrow{x} \in \mathcal{X}^n, z \in \{0,1\}^*} \equiv \left\{\textsc{Real}_{\pi,\mathcal{I},\mathcal{A}(z)}(\overrightarrow{x}, \lambda)\right\}_{\overrightarrow{x} \in \mathcal{X}^n, z \in \{0,1\}^*}$$

We will sometimes relax security to statistical or computational definitions. A protocol is statistically secure if the random variables $\textsc{Ideal}^{\mathsf{id\text{-}abort}}_{f,\mathcal{I},\mathcal{S}(z)}(\overrightarrow{x}, \lambda)$ and $\textsc{Real}_{\pi,\mathcal{I},\mathcal{A}(z)}(\overrightarrow{x}, \lambda)$ are statistically close, and computationally secure if they are computationally indistinguishable.

3.3 The Hybrid Model

We recall the definition of the hybrid model from [GHKL11] and [CL17]. The hybrid model combines both the real and ideal worlds. Specifically, an execution of a protocol π in the $\mathcal{G}$-hybrid model, for some functionality $\mathcal{G}$, involves parties sending normal messages to each other (as in the real model) and, in addition, having access to a trusted party computing $\mathcal{G}$. The parties communicate with this trusted party in exactly the same way as in the ideal models described above; the question of which ideal model is taken (that with or without abort) must be specified. In this paper, we always consider a hybrid model where the functionality $\mathcal{G}$ is computed according to the ideal model *with abort*. In all our protocols in the $\mathcal{G}$-hybrid model there will only be *sequential* calls to $\mathcal{G}$, that is, there is at most a single call to $\mathcal{G}$ per round, and no other messages are sent during any round in which $\mathcal{G}$ is called. This is especially important for reactive functionalities, where the calls to f are carried out in phases, and a new invocation of f cannot take place before all the phases of the previous invocation complete.

Let $\mathsf{type} \in \{\mathsf{g.d.}, \mathsf{fair}, \mathsf{id\text{-}fair}, \mathsf{abort}, \mathsf{id\text{-}abort}\}$. Let $\mathcal{G}$ be a functionality and let π be an n-party protocol for computing some functionality f, where π includes real messages between the parties as well as calls to $\mathcal{G}$. Let $\mathcal{A}$ be a non-uniform probabilistic polynomial-time machine with auxiliary input z. $\mathcal{A}$ corrupts at most t parties, where $0 \leq t < n$. Denote by $\mathcal{I} \subseteq [n]$ the set of indices of the parties corrupted by $\mathcal{A}$. Let $\textsc{Hybrid}^{\mathcal{G},\mathsf{type}}_{\pi,\mathcal{I},\mathcal{A}(z)}(\overrightarrow{x}, \lambda)$ be the random variable consisting of the view of the adversary and the output of the honest parties, following an execution of π with ideal calls to a trusted party computing $\mathcal{G}$ according to the ideal model "type" where P_i begins by holding x_i for every $i \in [n]$. Security in the model "type" can be defined via natural modifications of Definitions 1, 2, 3, 4 and 5. We call this the $(\mathcal{G}, \mathsf{type})$-hybrid model.

The hybrid model gives a powerful tool for proving the security of protocols. Specifically, we may design a real-world protocol for securely computing some functionality f by first constructing a protocol for computing f in the $\mathcal{G}$-hybrid model. Letting π denote the protocol thus constructed (in the $\mathcal{G}$-hybrid model), we denote by π^ρ the real-world protocol in which calls to $\mathcal{G}$ are replaced by sequential execution of a real-world protocol ρ that computes $\mathcal{G}$ in the ideal model "type". "Sequential" here implies that only one execution of ρ is carried out at any time, and no other π-protocol messages are sent during the execution of ρ. The results of [Can00] then imply that if π securely computes f in the $(\mathcal{G}, \mathsf{type})$-hybrid model, and ρ securely computes $\mathcal{G}$, then the composed protocol π^ρ securely computes f (in the real world). For completeness, we state this result formally as we will use it in this work.

Lemma 1. *Let* $\mathsf{type}_1, \mathsf{type}_2 \in \{\mathsf{g.d.}, \mathsf{fair}, \mathsf{id\text{-}fair}, \mathsf{abort}, \mathsf{id\text{-}abort}\}$. *Let* $\mathcal{G}$ *be an* n*-party functionality. Let* ρ *be a protocol that securely computes* $\mathcal{G}$ *with* type_1*, and let* π *be a protocol that securely computes* f *with* type_2 *in the* $(\mathcal{G}, \mathsf{type}_1)$*-hybrid model. Then protocol* π^ρ *securely computes* f *with* type_2 *in the real model.*

Sometimes, while working in a hybrid model, say the $(\mathcal{G}, \mathsf{type})$-hybrid model, we will suppress type and simply state that we are working in the $\mathcal{G}$-hybrid model. This is because type is implied by the context, $\mathcal{G}$. For instance, unless specified otherwise:

- When $\mathcal{G} = \mathcal{F}_{\mathsf{bc}}$[13], $\mathsf{type} = \mathsf{g.d.}$.
- When $\mathcal{G} = \mathcal{F}_{\mathsf{OT}}$[14], $\mathsf{type} = \mathsf{abort}$.

[13] See Section 3.7.
[14] See Section 3.6.

- When $\mathcal{G} = \mathcal{F}_{\mathsf{2PC}}$[15], $\mathsf{type} = \mathsf{abort}$.
- When $\mathcal{G} = \mathcal{F}_{\mathsf{MPC}}$[16], $\mathsf{type} = \mathsf{abort}$.
- When $\mathcal{G} = \mathcal{F}_{\mathsf{SyX}}$[17], $\mathsf{type} = \mathsf{g.d.}$.

When working in a hybrid model that uses multiple ideal functionalities, $\mathcal{G}_1, \ldots, \mathcal{G}_k$ with associated types $\mathsf{type}_1, \ldots, \mathsf{type}_k$ for some $k \in \mathbb{N}$, we call it the $(\mathcal{G}_1, \mathsf{type}_1, \ldots, \mathcal{G}_k, \mathsf{type}_k)$-hybrid model. Furthermore, we will suppress type_j when type_j is implied by the context, $\mathcal{G}_j$ for $j \in [k]$.

3.4 Fairness versus Guaranteed Output Delivery

We recall here some of the results from [CL17].

Lemma 2. *[CL17] Consider n parties $P_1, \ldots, P_n$ in a model without a broadcast channel. Then, there exists a functionality $f : \mathcal{X}^n \to \mathcal{Y}^n$ such that f cannot be securely computed with guaranteed output delivery in the presence of t-threshold adversaries for $n/3 \leq t < n$.*

Lemma 3. *[CL17] Consider n parties $P_1, \ldots, P_n$ in a model with a broadcast channel. Then, assuming the existence of one-way functions, for any functionality $f : \mathcal{X}^n \to \mathcal{Y}^n$, if there exists a protocol π which securely computes f with fairness, then there exists a protocol π' which securely computes f with guaranteed output delivery.*

Lemma 4. *[CL17] Consider n parties $P_1, \ldots, P_n$ in a model with a broadcast channel. Then, assuming the existence of one-way functions, for any functionality $f : \mathcal{X}^n \to \mathcal{Y}^n$, if there exists a protocol π which securely computes f with fairness, then there exists a protocol π' which securely computes f with fairness and does not make use of the broadcast channel.*

3.5 Computing with an Honest Majority

We recall here some of the known results regarding feasibility of information-theoretic multiparty computation in the presence of an honest majority.

Lemma 5. *[GMW87] Consider n parties $P_1, \ldots, P_n$ in the* point-to-point *model. Then, there exists a protocol π which securely computes $\mathcal{F}_{\mathsf{MPC}}$ with guaranteed output delivery in the presence of t-threshold adversaries for any $0 \leq t < n/3$.*

Lemma 6. *[FGMvR02] Consider n parties $P_1, \ldots, P_n$ in the* point-to-point *model. Then, there exists a protocol π which securely computes $\mathcal{F}_{\mathsf{MPC}}$ with fairness in the presence of t-threshold adversaries for any $0 \leq t < n/2$.*

Lemma 7. *[GMW87, RB89] Consider n parties $P_1, \ldots, P_n$ in the* broadcast *model. Then, there exists a protocol π which securely computes $\mathcal{F}_{\mathsf{MPC}}$ with guaranteed output delivery in the presence of t-threshold adversaries for any $0 \leq t < n/2$.*

Preliminaries: $x_0, x_1 \in \{0,1\}^m$; $b \in \{0,1\}$. The functionality proceeds as follows:

- Upon receiving inputs (x_0, x_1) from the sender P_1 and b from the receiver P_2, send $\perp$ to P_1 and x_b to P_2.

Figure 1: The ideal functionality $\mathcal{F}_{\mathsf{OT}}$.

Preliminaries: $x_1, \ldots, x_n \in \{0,1\}^*$; $f_1, \ldots, f_n$ is an n-input, n-output functionalities. The functionality proceeds as follows:

- Upon receiving inputs (x_i, f_i) from P_i for all $i \in [n]$, check if $f = f_i$ for all $i \in [n]$. If not, abort. Else, send $f^i(x_1, \ldots, x_n)$ to P_i for all $i \in [n]$.

Figure 2: The ideal functionality $\mathcal{F}_{\mathsf{MPC}}$.

3.6 Oblivious Transfer

In this work, oblivious transfer, or OT, refers to 1-out-of-2 oblivious transfer defined as in Figure 1. We note that in the definition of $\mathcal{F}_{\mathsf{OT}}$, one party, namely P_1, is seen as the sender, while the other, namely P_2, is seen as the receiver. However, from [WW06], OT is symmetric, which implies that the roles of the sender and the receiver can be reversed. Thus, if two parties P_1 and P_2 have access to the ideal functionality $\mathcal{F}_{\mathsf{OT}}$, they can perform 1-out-of-2 oblivious transfer with either party as a sender and the other as the receiver. It is known that OT is complete for secure multiparty computation with abort. We state this result formally below.

Lemma 8. *[Kil88, GV87, IPS08] Consider n parties $P_1, \ldots, P_n$ in the* OT-network *model. Then, there exists a protocol π which securely computes $\mathcal{F}_{\mathsf{MPC}}$ with abort in the presence of t-threshold adversaries for any $0 \le t < n$.*

3.7 Broadcast

Broadcast is defined as in Figure 3. We recall that the ideal functionality for broadcast, namely $\mathcal{F}_{\mathsf{bc}}$, can be securely computed with guaranteed output delivery in the presence of t-threshold adversaries if and only if $0 \le t < n/3$ [PSL80, LSP82]. Furthermore, $\mathcal{F}_{\mathsf{bc}}$ can be securely computed with fairness in the presence of t-threshold adversaries for any $0 \le t < n$ [FGH$^+$02]. Furthermore,

[15] See Section 3.14.
[16] See Section 3.6.
[17] See Section 3.14.

Preliminaries: $x \in \{0,1\}^*$. The functionality proceeds as follows:

- Upon receiving the input x from the sender P_1, send x to all parties $P_1, \ldots, P_n$.

Figure 3: The ideal functionality $\mathcal{F}_{\mathsf{bc}}$.

these results hold irrespective of the model we are working in so long as we do not have explicit access to $\mathcal{F}_{\mathsf{bc}}$.

3.8 Authentication Scheme with Public Verification

Definition 6. *An authentication scheme for message space $\mathcal{M}_\lambda$ is a triple of PPT algorithms $\mathcal{T} = (\mathsf{Gen}, \mathsf{Tag}, \mathsf{Verify})$ such that for all $\lambda \in \mathbb{N}$ and all messages $m \in \mathcal{M}_\lambda$,*

$$\Pr\left[\begin{array}{c} \mathsf{sk} \xleftarrow{\$} \mathcal{T}.\mathsf{Gen}(1^\lambda) \\ \sigma = \mathcal{T}.\mathsf{Tag}(m; \mathsf{sk}) \end{array} : \mathcal{T}.\mathsf{Verify}(\sigma, m) = 1\right] = 1$$

An authentication scheme for message space $\mathcal{M}_\lambda$ is existentially unforgeable if for any PPT adversary $\mathcal{A}$, the following probability is negligible in λ:

$$\Pr\left[\begin{array}{c} \mathsf{sk} \xleftarrow{\$} \mathcal{T}.\mathsf{Gen}(1^\lambda) \\ \mathcal{Q} = \emptyset \\ \left\{\begin{array}{c} m_i \xleftarrow{\$} \mathcal{A}(1^\lambda, \mathcal{Q}) \\ \sigma_i = \mathcal{T}.\mathsf{Tag}(m_i; \mathsf{sk}) \\ \mathcal{Q} = \mathcal{Q} \cup \{(m_i, \sigma_i)\} \end{array}\right\}_i \\ (m, \sigma) \xleftarrow{\$} \mathcal{A}(1^\lambda, \mathcal{Q}) \end{array} : \mathcal{T}.\mathsf{Verify}(\sigma, m) = 1 \ \wedge \ (m, \sigma) \notin \mathcal{Q}\right]$$

We remark that both the Tag and Verify algorithms are deterministic.

3.9 Honest-Binding Commitment Schemes

We recall the notion of honest-binding commitments from [GKKZ11]. Commitment schemes are a standard cryptographic tool. Roughly, a commitment scheme allows a sender S to generate a commitment c to a message m in such a way that (1) the sender can later open the commitment to the original value m (correctness); (2) the sender cannot generate a commitment that can be opened to two different values (binding); and (3) the commitment reveals nothing about the sender's value m until it is opened (hiding). For our application, we need a variant of standard commitments that guarantees binding when the sender is honest but ensures that binding can be violated if the sender is dishonest. (In the latter case, we need some additional properties as well; these will become clear in what follows.) Looking ahead, we will use such commitment schemes to enable a simulator in security proofs to generate a commitment dishonestly. This will give the simulator the flexibility to break binding and open the commitment to any desired message (if needed), while also being able to ensure binding (when desired) by claiming that it generated the commitment honestly.

We consider only non-interactive commitment schemes. For simplicity, we define our schemes in such a way that the decommitment information consists of the sender's random coins ω that it used when generating the commitment.

Definition 7. *A (non-interactive) commitment scheme for message space $\mathcal{M}_\lambda$ is a pair of PPT algorithms $(\mathsf{Com}, \mathsf{Open})$ such that for all $\lambda \in \mathbb{N}$, all messages $m \in \mathcal{M}_\lambda$, and all random coins ω it holds that*

$$\mathsf{Open}(\mathsf{Com}(1^\lambda, m; \omega), \omega, m) = 1$$

A commitment scheme for message space $\mathcal{M}_\lambda$ is honest-binding if it satisfies the following:

Binding (for an honest sender). *For all PPT algorithms $\mathcal{A}$ (that maintain state throughout their execution), the following probability is negligible in λ:*

$$\Pr\left[\begin{array}{c} m \xleftarrow{\$} \mathcal{A}(1^k); \omega \xleftarrow{\$} \{0,1\}^* \\ c = \mathsf{Com}(1^\lambda, m; \omega) \\ (m', \omega') \xleftarrow{\$} \mathcal{A}(c, \omega) \end{array} : \mathsf{Open}(c, m', \omega') = 1 \ \wedge \ m \neq m' \right]$$

Equivocation. *There is a pair of algorithms $\left(\widetilde{\mathsf{Com}}, \widetilde{\mathsf{Open}}\right)$ such that for all PPT algorithms $\mathcal{A}$ (that maintain state throughout their execution), the following quantity is negligible in λ:*

$$\left| \begin{array}{l} \Pr\left[m \xleftarrow{\$} \mathcal{A}(1^\lambda); \omega \xleftarrow{\$} \{0,1\}^*; c = \mathsf{Com}(1^\lambda, m; \omega) : \mathcal{A}(1^\lambda, c, \omega) = 1\right] \\ \qquad - \Pr\left[(c, \mathsf{state}) \xleftarrow{\$} \widetilde{\mathsf{Com}}(1^\lambda), m \xleftarrow{\$} \mathcal{A}(1^\lambda); \omega \xleftarrow{\$} \widetilde{\mathsf{Open}}(\mathsf{state}, m) : \mathcal{A}(1^\lambda, c, \omega) = 1\right] \end{array} \right|$$

Equivocation implies the standard hiding property, namely, that for all PPT algorithms $\mathcal{A}$ (that maintain state throughout their execution) the quantity is negligible in λ:

$$\left| \Pr\left[(m_0, m_1) \xleftarrow{\$} \mathcal{A}(1^\lambda); b \xleftarrow{\$} \{0,1\}; c \xleftarrow{\$} \mathsf{Com}(1^\lambda, m_b) : \mathcal{A}(c) = b\right] \right|$$

We also observe that if (c, ω) are generated by $\left(\widetilde{\mathsf{Com}}, \widetilde{\mathsf{Open}}\right)$ for some message m as in the definition above, then binding still holds: namely, no PPT adversary given (m, c, ω) can find (m', ω') with $m' \neq m$ such that $\mathsf{Open}(c, m', \omega') = 1$.

We will sometimes use the notation $(c, \omega) \xleftarrow{\$} \mathsf{Com}(m)$ to mean $c = \mathsf{Com}(1^\lambda, m; \omega)$, suppressing λ when it is clear from the context and having the committing algorithm Com return the commitment and the decommitment information or opening. [GKKZ11] provides constructions of honest-binding commitments for bits assuming the existence of one-way functions.

3.10 Digital Signatures

Definition 8. *A (digital) signature scheme for message space $\mathcal{M}_\lambda$ is triple of PPT algorithms $\mathcal{V} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ such that for all $\lambda \in \mathbb{N}$ and all messages $m \in \mathcal{M}_\lambda$,*

$$\Pr\left[\begin{array}{c} (\mathsf{vk}, \mathsf{sk}) \xleftarrow{\$} \mathcal{V}.\mathsf{Gen}(1^\lambda) \\ \sigma \xleftarrow{\$} \mathcal{V}.\mathsf{Sign}(m; \mathsf{sk}) \end{array} : \mathcal{V}.\mathsf{Verify}(\sigma, m; \mathsf{vk}) = 1 \right] = 1$$

A signature scheme for message space $\mathcal{M}_\lambda$ is existentially unforgeable if for any PPT adversary $\mathcal{A}$, the following probability is negligible in λ:

$$\Pr\left[\begin{array}{c} (\mathsf{vk}, \mathsf{sk}) \xleftarrow{\$} \mathcal{V}.\mathsf{Gen}(1^\lambda) \\ \mathcal{Q} = \emptyset \\ \left\{ \begin{array}{c} m_i \xleftarrow{\$} \mathcal{A}(1^\lambda, \mathcal{Q}) \\ \sigma_i \xleftarrow{\$} \mathcal{V}.\mathsf{Sign}(m_i; \mathsf{sk}) \\ \mathcal{Q} = \mathcal{Q} \cup \{(m_i, \sigma_i)\} \end{array} \right\}_i \end{array} : \mathcal{V}.\mathsf{Verify}(\sigma, m; \mathsf{vk}) = 1 \ \wedge \ (m, \sigma) \notin \mathcal{Q} \right]$$

[Rom90] provides constructions of existentially unforgeable signatures assuming the existence of one-way functions.

3.11 Authenticated Encryption

Formally, an authenticated encryption scheme $\mathcal{E}$ is a symmetric key encryption scheme that consists of the following three PPT algorithms:

- $\mathcal{E}.\mathsf{Gen}(1^\lambda)$: Given the security parameter, λ, the key generation algorithm outputs a secret key. This is denoted by: $\mathsf{sk} \stackrel{\$}{\leftarrow} \mathcal{E}.\mathsf{Gen}(1^\lambda)$. This implicitly defines a message space $\mathcal{M}_\lambda$.
- $\mathcal{E}.\mathsf{Enc}(m; \mathsf{sk})$: Given the secret key sk and a message $m \in \mathcal{M}_\lambda$, the encryption algorithm returns a ciphertext $\mathsf{ct} \stackrel{\$}{\leftarrow} \mathcal{E}.\mathsf{Enc}(m; \mathsf{sk})$.
- $\mathcal{E}.\mathsf{Dec}(\mathsf{ct}; \mathsf{sk})$: Given the secret key sk and a ciphertext ct, the decryption algorithm returns a message $m \stackrel{\$}{\leftarrow} \mathcal{E}.\mathsf{Dec}(\mathsf{ct}; \mathsf{sk})$, where $m \in \mathcal{M}_\lambda \cup \{\perp\}$.

We make the standard correctness requirement; namely, for any sk output by $\mathcal{E}.\mathsf{Gen}$ and any $m \in \mathcal{M}_\lambda$, we have $\mathcal{E}.\mathsf{Dec}(\mathcal{E}.\mathsf{Enc}(m; \mathsf{sk}); \mathsf{sk}) = m$. We now give the formal definition of security.

Definition 9. *Let $\mathcal{E}$ be an authenticated encryption scheme. We say that $\mathcal{E}$ is semantically secure if the advantage of any PPT algorithm $\mathcal{A}$ in the game below is negligible in λ:*

1. *The key generation algorithm $\mathcal{E}.\mathsf{Gen}(1^\lambda)$ is run to get sk. The algorithm $\mathcal{A}$ is given 1^λ as input.*
2. *$\mathcal{A}$ outputs a challenge message pair $(m_0, m_1) \in \mathcal{M}_\lambda^2$.*
3. *A bit b is chosen at random and a ciphertext $\mathsf{ct} \stackrel{\$}{\leftarrow} \mathcal{E}.\mathsf{Enc}(m_b; \mathsf{sk})$ is computed. $\mathcal{A}$ receives ct.*
4. *$\mathcal{A}$ outputs a bit $b' \in \{0, 1\}$.*

The advantage of $\mathcal{A}$ is defined as $2 \cdot \left|\Pr[b = b'] - \frac{1}{2}\right|$.

For authenticated encryption schemes, we are also considered with the notion of *integrity of ciphertexts*. Informally, this means that an adversary, given access to any polynomial number of ciphertexts, cannot come up with a different (unseen) valid ciphertext (one that decrypts to a non-$\perp$ value in $\mathcal{M}_\lambda$). We define this formally below.

Definition 10. *Let $\mathcal{E}$ be an authenticated encryption scheme. We say that $\mathcal{E}$ is INT-CTXT-secure if the advantage of any PPT algorithm $\mathcal{A}$ in the game below is negligible in λ:*

1. *The key generation algorithm $\mathcal{E}.\mathsf{Gen}(1^\lambda)$ is run to get sk and $S = \emptyset$ is initialized. The algorithm $\mathcal{A}$ is given 1^λ as input.*
2. *$\mathcal{A}$ may request (repeatedly) for the encryptions of messages of its choice. If $\mathcal{A}$ supplies a message $m \in \mathcal{M}_\lambda$, a ciphertext $\mathsf{ct} \stackrel{\$}{\leftarrow} \mathcal{E}.\mathsf{Enc}(m; \mathsf{sk})$ is computed, $S = S \cup \{\mathsf{ct}\}$ is updated and $\mathcal{A}$ receives ct.*
3. *$\mathcal{A}$ outputs a challenge ciphertext ct^*. The output of the decryption $m^* \stackrel{\$}{\leftarrow} \mathcal{E}.\mathsf{Dec}(\mathsf{ct}^*; \mathsf{sk})$ is computed. If $m^* \neq \perp$, the value $\mathsf{res} = 1$ is output. Otherwise, the value $\mathsf{res} = 0$ is output.*

The advantage of $\mathcal{A}$ is defined as $\Pr[\mathsf{res} = 1]$.

For correctness and ease of exposition, as in [PST17], we will leverage Diffie-Hellman for key-exchange and an authenticated encryption scheme. It is not hard to modify our protocols for any secure key-exchange protocol. Since the existence of secure key exchange protocols imply the existence of authenticated encryption, it would suffice to assume secure key-exchange.

3.12 Receiver Non-Committing Encryption

We recall the notion of receiver non-committing encryption from [CHK05]. On a high level, a receiver non-committing encryption scheme is one in which a simulator can generate a single "fake ciphertext" and later "open" this ciphertext (by showing an appropriate secret key) as any given message. These "fake ciphertexts" should be indistinguishable from real ciphertexts, even when an adversary is given access to a decryption oracle before the fake ciphertext is known.

Formally, a receiver non-committing encryption scheme $\mathcal{E}$ consists of the following five PPT algorithms:

- $\mathcal{E}.\mathsf{Gen}(1^\lambda)$: Given the security parameter, λ, the key generation algorithm outputs a key-pair and some auxiliary information. This is denoted by: $(\mathsf{pk}, \mathsf{sk}, z) \xleftarrow{\$} \mathcal{E}.\mathsf{Gen}(1^\lambda)$. The public key pk defines a message space $\mathcal{M}_\lambda$.
- $\mathcal{E}.\mathsf{Enc}(m; \mathsf{pk})$: Given the public key pk and a message $m \in \mathcal{M}_\lambda$, the encryption algorithm returns a ciphertext $\mathsf{ct} \xleftarrow{\$} \mathcal{E}.\mathsf{Enc}(m; \mathsf{pk})$.
- $\mathcal{E}.\mathsf{Dec}(\mathsf{ct}; \mathsf{sk})$: Given the secret key sk and a ciphertext ct, the decryption algorithm returns a message $m \xleftarrow{\$} \mathcal{E}.\mathsf{Dec}(\mathsf{ct}; \mathsf{sk})$, where $m \in \mathcal{M}_\lambda \cup \{\perp\}$.
- $\mathcal{E}.\widetilde{\mathsf{Enc}}(\mathsf{pk}, \mathsf{sk}, z)$: Given the triple $(\mathsf{pk}, \mathsf{sk}, z)$ output by $\mathcal{E}.\mathsf{Gen}$, the fake encryption algorithm outputs a "fake ciphertext" $\widetilde{\mathsf{ct}} \xleftarrow{\$} \mathcal{E}.\widetilde{\mathsf{Enc}}(\mathsf{pk}, \mathsf{sk}, z)$.
- $\mathcal{E}.\widetilde{\mathsf{Dec}}(\mathsf{pk}, \mathsf{sk}, z, \widetilde{\mathsf{ct}}, m)$: Given the triple $(\mathsf{pk}, \mathsf{sk}, z)$ output by $\mathcal{E}.\mathsf{Gen}$, a "fake ciphertext" $\widetilde{\mathsf{ct}}$ output by $\mathcal{E}.\widetilde{\mathsf{Enc}}$ and a message $m \in \mathcal{M}_\lambda$, the "fake decryption" algorithm outputs a "fake secret key" $\widetilde{\mathsf{sk}} \xleftarrow{\$} \mathcal{E}.\widetilde{\mathsf{Dec}}(\mathsf{pk}, \mathsf{sk}, z, \widetilde{\mathsf{ct}}, m)$. (Intuitively, $\widetilde{\mathsf{sk}}$ is a valid-looking secret key for which $\widetilde{\mathsf{ct}}$ decrypts to m.)

We make the standard correctness requirement; namely, for any $(\mathsf{pk}, \mathsf{sk}, z)$ output by $\mathcal{E}.\mathsf{Gen}$ and any $m \in \mathcal{M}_\lambda$, we have $\mathcal{E}.\mathsf{Dec}(\mathcal{E}.\mathsf{Enc}(m; \mathsf{pk}); \mathsf{sk}) = m$. Our definition of security requires, informally, that for any message m an adversary cannot distinguish whether it has been given a "real" encryption of m along with a "real" secret key, or a "fake" ciphertext along with a "fake" secret key under which the ciphertext decrypts to m. This should hold even when the adversary has non-adaptive access to a decryption oracle. We now give the formal definition.

Definition 11. *Let $\mathcal{E}$ be a receiver non-committing encryption scheme. We say that $\mathcal{E}$ is secure if the advantage of any PPT algorithm $\mathcal{A}$ in the game below is negligible in λ:*

1. *The key generation algorithm $\mathcal{E}.\mathsf{Gen}(1^\lambda)$ is run to get $(\mathsf{pk}, \mathsf{sk}, z)$.*
2. *The algorithm $\mathcal{A}$ is given 1^λ and pk as input, and is also given access to a decryption oracle $\mathcal{E}.\mathsf{Dec}(\cdot; \mathsf{sk})$. It then outputs a challenge message $m \in \mathcal{M}_\lambda$.*
3. *A bit b is chosen at random. If $b = 1$ then a ciphertext $\mathsf{ct} \xleftarrow{\$} \mathcal{E}.\mathsf{Enc}(m; \mathsf{pk})$ is computed, and $\mathcal{A}$ receives $(\mathsf{ct}, \mathsf{sk})$. Otherwise, a "fake" ciphertext $\widetilde{\mathsf{ct}} \xleftarrow{\$} \mathcal{E}.\widetilde{\mathsf{Enc}}(\mathsf{pk}, \mathsf{sk}, z)$ and a "fake" secret key $\widetilde{\mathsf{sk}} \xleftarrow{\$} \mathcal{E}.\widetilde{\mathsf{Dec}}(\mathsf{pk}, \mathsf{sk}, z, \widetilde{\mathsf{ct}}, m)$ are computed, and $\mathcal{A}$ receives $(\widetilde{\mathsf{ct}}, \widetilde{\mathsf{sk}})$. (After this point, $\mathcal{A}$ can no longer query its decryption oracle.) $\mathcal{A}$ outputs a bit $b \in \{0, 1\}$.*

The advantage of $\mathcal{A}$ is defined as $2 \cdot \left|\Pr[b = b'] - \frac{1}{2}\right|$.

3.13 Non-interactive Non-Committing Encryption

We recall the notion of non-interactive non-committing encryption from [Nie02]. We do so in two ways. The first way of looking at non-interactive non-committing encryption is that it is the same as receiver non-committing encryption, except that it can equivocate multiple ciphertexts as opposed to one. On a high level, a non-interactive non-committing encryption scheme is one in which a simulator can generate multiple "fake ciphertexts" and later "open" them (by showing an appropriate secret key) as any given message vector. We first note that the receiver non-committing encryption scheme of [CHK05] can be extended, as noted by them, to support equivocation of any bounded number of ciphertexts. However, the size of the key of the scheme would grow linearly with the number of outstanding ciphertexts. Such schemes can be constructed based on standard assumptions such as the quadratic residuosity assumption. If no bound on the number of outstanding texts is known *apriori*, then as noted in [Nie02], constructing such schemes is impossible in the standard model. The other way of looking at non-interactive non-committing encryption is that it is a realization of the ideal functionality for public key encryption, namely, $\mathcal{F}_{\mathsf{PKE}}$. We refer the reader to [CKN03, CHK05] for further details.

For the sake of completeness and ease of later presentation, we recall the non-interactive non-committing encryption scheme of [Nie02] in the random-oracle model. Let $\mathcal{F} = (\mathcal{K}, F)$ be a collection of trapdoor permutations, where $\mathcal{K}$ denotes an index set and $F = \{f_k\}_{k\in\mathcal{K}}$ is a set of permutations with efficiently samplable domains. For every $k \in \mathcal{K}$, we denote by t_k the trapdoor associated with k which enables inversion of f_k. We assume the existence of a generation algorithm $\mathcal{G}$ which on input the security parameter λ outputs a key-trapdoor pair (k, t_k) uniformly at random. Let $H : \{0,1\}^* \to \{0,1\}^{\ell(\lambda)}$ be a random oracle (instantiated by an appropriate hash function). The non-interactive non-committing encryption scheme $\mathcal{E}$ consists of the following algorithms:

- $\mathcal{E}.\mathsf{Gen}(1^\lambda)$: Given the security parameter, λ, the key generation algorithm obtains (k, t_k) by executing $\mathcal{G}$ with the security parameter λ as input. It then outputs the public and private keys $\mathsf{pk} = (k, f_k, H)$ and $\mathsf{sk} = t_k$. The message space is defined to be $\mathcal{M}_\lambda = \{0,1\}^{\ell(\lambda)}$.

- $\mathcal{E}.\mathsf{Enc}(m; \mathsf{pk})$: Given the public key pk and a message $m \in \mathcal{M}_\lambda$, the encryption algorithm samples x from the domain of f_k and returns a ciphertext $\mathsf{ct} = (f_k(x), H(x) \oplus m)$.

- $\mathcal{E}.\mathsf{Dec}(\mathsf{ct}; \mathsf{sk})$: Given the secret key sk and a ciphertext $\mathsf{ct} = (\mathsf{ct}^1, \mathsf{ct}^2)$, the decryption algorithm computes x by inverting ct^1 using t_k and returns the message $m = H(x) \oplus \mathsf{ct}^2$.

We refer the reader to [Nie02] for a complete proof that the scheme defined above is a non-interactive non-committing encryption scheme. The sketch the proof here. The scheme is clearly non-interactive. We now need to design a simulator $\mathcal{S}$ which can generate multiple "fake ciphertexts" and later "open" them to an arbitrary sequence of messages. Note that this is easy to do. To generate n "fake ciphertexts", $\mathcal{S}$ samples $x_1, \ldots, x_n$ independently at random from the domain of f_k. It then samples $y_1, \ldots, y_n \xleftarrow{\$} \{0,1\}^{\ell(\lambda)}$. The m ciphertexts are defined to be $\{\mathsf{ct}_i\}_{i\in[n]}$ where $\mathsf{ct}_i = (f_k(x_i), y_i)$. Then, in order to open the n ciphertexts to a message vector $\overrightarrow{m} = (m_1, \ldots, m_n) \in \mathcal{M}_\lambda^n$, $\mathcal{S}$ would program the random oracle H such that $H(x_i) = m_i \oplus y_i$. Note that this ensures that the "fake ciphertexts" do in fact "open" to the message vector $\overrightarrow{m}$. We also stress, as this will be required for us later, that the simulator need not know n in advance, that is, it can produce any (polynomially bounded) number of "fake ciphertexts" and later "open" them as required. This is also precisely the difference from receiver non-committing encryption as described earlier which necessitates the use of random oracles as noted in [Nie02].

Preliminaries: $x_1, x_2 \in \{0,1\}^*$; f_1, f_2 are 2-input, 2-output functions; ϕ_1, ϕ_2 are boolean predicates. The functionality proceeds as follows:

- **Input phase.** Upon receiving inputs $(x_1, f = (f_1, f_2, \phi_1, \phi_2))$ from P_1 and (x_2, f') from P_2, check if $f = f'$. If not, abort. Else, compute $f_1(x_1, x_2)$. If $f_1(x_1, x_2) = \perp$[a], abort. Else, send $f_1(x_1, x_2)$ to both parties, and go to next phase.
- **Trigger phase.** Upon receiving input w from party P_i, check if $\phi_i(w) = 1$. If yes, then send $(w, f_2(x_1, x_2, w))$ to both P_1 and P_2.

[a]We crucially require that $\perp$ is a special symbol different from the empty string. We use $\perp$ as a means of signalling that the input phase of $\mathcal{F}_{\mathsf{SyX}}$ did not complete successfully. We will however allow parties to attempt to invoke the input phase of the functionality at a later time. However, as we proceed, we will also have our functionality be clock-aware and thus only accept invocations to the input phase until a certain point in time. After the input phase times out, the functionality is rendered completely unusable. Similarly, if the input phase has been completed successfully, a clock-oblivious version of the functionality can be triggered at any point in time as long as a valid witness is provided, no matter the number of failed attempts. The clock-aware version of the functionality, however, will only accept invocations of the trigger phase until a certain point in time. After the trigger phase times out, the functionality is rendered completely unusable.

Figure 4: The ideal functionality $\mathcal{F}_{\mathsf{SyX}}$.

3.14 Synchronizable Exchange

Synchronizable exchange is defined as in Figure 4. In order to guarantee termiantion, we will need our ideal functionality to be "clock-aware". In this work, we stick to the formalism outlined in [PST17]. We recall that in this model, we assume that every party and every invocation of the ideal functionality $\mathcal{F}_{\mathsf{SyX}}$ has access to a variable r that reflects the current round number. More generally, every function and predicate that is part of the specification of $\mathcal{F}_{\mathsf{SyX}}$ may also take r as an input. Finally, the functionality may also time out after a pre-programmed amount of time. We describe this clock-aware functionality in Figure 5. It is known that $\mathcal{F}_{\mathsf{SyX}}$ is complete for fair secure multiparty computation. We state this result formally below.

Lemma 9. *[KRS20] Consider n parties $P_1, \ldots, P_n$ in the* point-to-point *model. Then, assuming the existence of one-way functions, there exists a protocol π which securely computes $\mathcal{F}_{\mathsf{MPC}}$ with fairness in the presence of t-threshold adversaries for any $0 \leq t < n$ in the $\mathcal{F}_{\mathsf{SyX}}$-hybrid model.*

Lemma 10. *[KRS20] Consider n parties $P_1, \ldots, P_n$ in the* point-to-point *model. Then, assuming the existence of one-way permutations, there exists a protocol π in the programmable random oracle model which securely preprocesses for and computes an arbitrary (polynomial) number of instances of $\mathcal{F}_{\mathsf{MPC}}$ with fairness in the presence of t-threshold adversaries for any $0 \leq t < n$ in the $\mathcal{F}_{\mathsf{SyX}}$-hybrid model.*

3.15 Attested Execution Secure Processors

In this section, we recall the $\mathcal{G}_{\mathsf{att}}$ abstraction from [PST17] capturing the essence of SGX-like secure processors that provide anonymous attestation (see Figure 6). We review the abstraction and explain some technicalities in the modeling.

Preliminaries: $x_1, x_2 \in \{0,1\}^*$; f_1, f_2 are 2-output functions; ϕ_1, ϕ_2 are boolean predicates; r denotes the current round number; $\mathsf{INPUT\_TIMEOUT} < \mathsf{TRIGGER\_TIMEOUT}$ are round numbers representing time outs. The functionality proceeds as follows:

- **Load phase.** If $r > \mathsf{INPUT\_TIMEOUT}$, abort. Otherwise, upon receiving inputs of the form $(x_1, f = (f_1, f_2, \phi_1, \phi_2))$ from P_1 and (x_2, f') from P_2, check if $f = f'$. If not, abort. Else, compute $f_1(x_1, x_2, r)$. If $f_1(x_1, x_2, r) = \perp$, abort. Else, send $f_1^i(x_1, x_2, r)$ to P_i for $i \in \{1, 2\}$, and go to next phase.
- **Trigger phase.** If $r > \mathsf{TRIGGER\_TIMEOUT}$, abort. Otherwise, upon receiving input w from party P_i, check if $\phi_i(w, r) = 1$. If yes, then send $(w, f_2^j(x_1, x_2, w, r))$ to both parties P_j for $j \in \{1, 2\}$.

Figure 5: The clock-aware ideal functionality $\mathcal{F}_{\mathsf{SyX}}$.

$\mathcal{G}_{\mathsf{att}}[\mathcal{V}, \mathsf{reg}]$

// initialization:
On initialize: $(\mathsf{vk_{att}}, \mathsf{sk_{att}}) := \mathcal{V}.\mathsf{Gen}(1^\lambda)$, $T = \emptyset$

// public query interface:
On receive* getpk() from some P: send $\mathsf{vk_{att}}$ to P

Enclave operations

// local interface – install an enclave:
On receive* install(idx, prog) from some $P \in \mathsf{reg}$:
if P is honest, assert $idx = sid$
generate nonce $eid \in \{0,1\}^\lambda$, store $T[eid, P] := \left(idx, \mathsf{prog}, \vec{0}\right)$, send eid to P

// local interface – resume an enclave:
On receive* resume(eid, inp) from some $P \in \mathsf{reg}$:
let $(idx, \mathsf{prog}, \mathsf{mem}) := T[eid, P]$, abort if not found
let $(\mathsf{outp}, \mathsf{mem}) := \mathsf{prog}(\mathsf{inp}, \mathsf{mem})$, update $T[eid, P] := (idx, \mathsf{prog}, \mathsf{mem})$
let $\sigma := \mathcal{V}.\mathsf{Sign}\,(idx, eid, \mathsf{prog}, \mathsf{outp}; \mathsf{sk_{att}})$, and send (outp, σ) to P

Figure 6: The ideal functionality $\mathcal{G}_{\mathsf{att}}$ – **a global functionality modeling an** SGX**-like secure processor.** Blue (and starred*) activation points denote *reentrant* activation points. Green activation points are executed at most once. The enclave program prog may be probabilistic and this is important for privacy-preserving applications. Enclave program outputs are included in an anonymous attestation σ. For honest parties, the functionality verifies that installed enclaves are parametrized by the session ID, *sid* of the current protocol instance.

1. **Registry.** First, $\mathcal{G}_{\mathsf{att}}$ is parametrized with a static registry reg this is meant to capture all platforms that are equipped with an attested execution processor.

2. **Stateful enclave operations.** A platform P that is in the registry reg may invoke enclave operations, including:

   - $\mathsf{install}$: installing a new enclave with a program prog, henceforth referred to as the enclave program. Upon installation, $\mathcal{G}_{\mathsf{att}}$ simply generates a fresh enclave identifier *eid* and returns the *eid*. This enclave identifier may now be used to uniquely identify the enclave instance.

   - resume: resuming the execution of an existing enclave with inputs inp. Upon a resume call, $\mathcal{G}_{\mathsf{att}}$ execute the prog over the inputs inp, an outputs an output outp. $\mathcal{G}_{\mathsf{att}}$ would then sign the prog together with outp as well as additional metadata, and return both outp and the resulting attestation. Each installed enclave can be resumed multiple times, and we stress that the enclave operations store state across multiple resume invocations.

3. **Anonymous attestation.** Anonymous attestation allows a user to verify that the attestation is produced by some attested execution processor, without identifying which one. To capture such anonymous attestation, the $\mathcal{G}_{\mathsf{att}}$ functionality has a manufacturer public key and secret key pair denoted by $(\mathsf{mpk}, \mathsf{msk})$, and is parametrized by a signature scheme $\mathcal{V}$. When an enclave resume operation is invoked, $\mathcal{G}_{\mathsf{att}}$ signs any output to be attested with msk using the signature scheme $\mathcal{V}$. $\mathcal{G}_{\mathsf{att}}$ provides the manufacturer public key mpk to any party upon query, using which, any party can verify an anonymous attestation signed by $\mathcal{G}_{\mathsf{att}}$.

The enclave program prog and all inputs inp are observable by the platform P that owns the secure processor, since P must be an intermediary in all interactions with its local secure processor.

3.16 Witness Indistinguishable Proof Systems

Definition 12. *A non-interactive witness indistinguishable proof system, henceforth denoted by* NIWI*, for an NP language* $\mathcal{L}$ *consists of the following algorithms:*

- $\mathsf{crs} \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$*: The generation algorithm takes as input*[18] *the security parameter* λ *and generates a common reference string* crs*.*

- $\pi \xleftarrow{\$} \mathsf{Prove}(\mathsf{crs}, \mathsf{stmt}, w)$*: The proof generation algorithm takes as input the common reference string* crs*, a statement* stmt *and a witness* w *such that* $(\mathsf{stmt}, w) \in R_{\mathcal{L}}$*, and produces a proof* π*.*

- $b \xleftarrow{\$} \mathsf{Verify}(\mathsf{crs}, \mathsf{stmt}, \pi)$*: The proof verification algorithm takes as input the common reference string* crs*, a statement* stmt *and a proof* π*, and outputs* 0 *or* 1*, denoting accept or reject.*

Perfect Completeness. *A non-interactive proof system is said to be perfectly complete, if an honest prover with a valid witness can always convince an honest verifier. Formally, for any* $(\mathsf{stmt}, w) \in R_{\mathcal{L}}$*, we have that*

$$\Pr[\mathsf{crs} \xleftarrow{\$} \mathsf{Gen}(1^\lambda), \pi \xleftarrow{\$} \mathsf{Prove}(\mathsf{crs}, \mathsf{stmt}, w) : \mathsf{Verify}(\mathsf{crs}, \mathsf{stmt}, \pi) = 1] = 1$$

[18] We assume that a description of the language $\mathcal{L}$ is also provided implicitly as an input.

$\mathcal{G}_{\mathsf{acrs}}$

On initialize: $(\mathsf{pk}_{\mathcal{E},\mathsf{acrs}}, \mathsf{sk}_{\mathcal{E},\mathsf{acrs}}) \xleftarrow{\$} \mathcal{E}.\mathsf{Gen}(1^\lambda)$, $(\mathsf{vk}_{\mathcal{V},\mathsf{acrs}}, \mathsf{sk}_{\mathcal{V},\mathsf{acrs}}) \xleftarrow{\$} \mathcal{V}.\mathsf{Gen}(1^\lambda)$, $\mathsf{crs} \xleftarrow{\$} \mathsf{NIWI}.\mathsf{Gen}(1^\lambda)$
On receive* "crs" from P: return $\mathcal{G}_{\mathsf{acrs}}.\mathsf{mpk} := (\mathsf{pk}_{\mathcal{E},\mathsf{acrs}}, \mathsf{vk}_{\mathcal{V},\mathsf{acrs}}, \mathsf{crs})$
On receive* "idk" from P: assert P is corrupt, and then return $\mathcal{V}.\mathsf{Sign}(P; \mathsf{sk}_{\mathcal{V},\mathsf{acrs}})$

Figure 7: The ideal functionality $\mathcal{G}_{\mathsf{acrs}}$ – **a global augmented common reference string.** Generates a public encryption key pair, a signing key pair, and a common reference string for the witness indistinguishable proof system. Upon query from a (corrupt) party, returns a signature on the party's identifier henceforth called the *identity key*.

Computational Soundness. *A non-interactive proof system is said to be computationally sound if for all PPT adversaries $\mathcal{A}$, the following probability is negligible in λ:*

$$\Pr[\mathsf{crs} \xleftarrow{\$} \mathsf{Gen}(1^\lambda), (\mathsf{stmt}, \pi) \xleftarrow{\$} \mathcal{A}(\mathsf{crs}) : \mathsf{Verify}(\mathsf{crs}, \mathsf{stmt}, \pi) = 1 \ \wedge \ \mathsf{stmt} \notin \mathcal{L}]$$

Witness Indistinguishability. *A non-interactive proof system is said to be witness indistinguishable if for all PPT adversaries $\mathcal{A}$, the following quantity is negligible in λ:*

$$\left| \Pr\left[\begin{array}{c} \mathsf{crs} \xleftarrow{\$} \mathsf{Gen}(1^\lambda), \\ (\mathsf{stmt}, w_0, w_1) \xleftarrow{\$} \mathcal{A}(\mathsf{crs}) \\ \pi \xleftarrow{\$} \mathsf{Prove}(\mathsf{crs}, \mathsf{stmt}, w_0) \end{array} : \begin{array}{c} (\mathsf{stmt}, w_0) \in R_{\mathcal{L}} \ \wedge \ (\mathsf{stmt}, w_1) \in R_{\mathcal{L}} \\ \wedge \ \mathcal{A}(\pi) = 1 \end{array} \right] - \Pr\left[\begin{array}{c} \mathsf{crs} \xleftarrow{\$} \mathsf{Gen}(1^\lambda), \\ (\mathsf{stmt}, w_0, w_1) \xleftarrow{\$} \mathcal{A}(\mathsf{crs}) \\ \pi \xleftarrow{\$} \mathsf{Prove}(\mathsf{crs}, \mathsf{stmt}, w_1) \end{array} : \begin{array}{c} (\mathsf{stmt}, w_0) \in R_{\mathcal{L}} \ \wedge \ (\mathsf{stmt}, w_1) \in R_{\mathcal{L}} \\ \wedge \ \mathcal{A}(\pi) = 1 \end{array} \right] \right|$$

3.17 Augmented Global CRS

The augmented global CRS functionality denoted by $\mathcal{G}_{\mathsf{acrs}}$ is described in Figure 7. We recall the need for this functionality in the context of [PST17]. $\mathcal{G}_{\mathsf{acrs}}$ was first proposed by [CDPW07]. $\mathcal{G}_{\mathsf{acrs}}$ provides a common reference string that is honestly generated. Honest parties never have to query $\mathcal{G}_{\mathsf{acrs}}$ for any additional information. On the other hand, $\mathcal{G}_{\mathsf{acrs}}$ leaves a backdoor for the adversary, such that the adversary can obtain identity keys pertaining to their party identifiers. In practice, the $\mathcal{G}_{\mathsf{acrs}}$ functionality can be implemented by having a trusted third party (which may be the trusted hardware manufacturer) that generates the reference string and hands out the appropriate secret keys as in [CDPW07].

3.18 Bulletin Board

We borrow the bulletin board abstraction of a shared ledger, defined in [CGJ+17] and [SGK19]. The bulletin board models a public ledger that lets parties publish arbitrary strings. On publishing the string on the bulletin board, the party receives a proof to establish that the string was indeed published. We model these proofs via authentication tags that can be publicly verified and the string subsequently publicly accessible. The bulletin board also guarantees that strings that were

$\mathcal{F}_{\mathsf{BB}}$

// initialization:
On initialize: $\mathsf{sk}_{\mathsf{BB}} \xleftarrow{\$} \mathcal{T}.\mathsf{Gen}(1^\lambda)$, $t = 0$, $\mathsf{Ledger} = \emptyset$

// public query interface:
On receive* getCurrentCounter() from P: send t to P

// posting to the bulletin board:
On receive* post(x) from P:
update $t = t + 1$
let $\sigma = \mathcal{T}.\mathsf{Tag}(t \| x; \mathsf{sk}_{\mathsf{BB}})$
update $\mathsf{Ledger} = \mathsf{Ledger} \cup \{(t, x, \sigma)\}$
send (t, σ) to P

// reading from the bulletin board:
On receive* getContent(T) from P:
assert $T \leq t$
compute (x, σ) such that $(T, x, \sigma) \in \mathsf{Ledger}$
send (x, σ) to P

Figure 8: The ideal functionality $\mathcal{F}_{\mathsf{BB}}$ – **a public shared ledger.** Generates a key for authentication, a counter and a set (for elements published on the ledger, along with their timestamps and tags). Upon query, it returns the current counter, which is the number of items published on the ledger; or the content published at a given time. It also allows parties to post items to the ledger – the functionality computes a tag on it, appends the item along with its tag and timestamp to the ledger set and returns the tag and timestamp.

successfully published will never be modified or deleted. For security, we require that the authentication tags follow the standard notion of unforgeability described in Section 3.8. In addition, the bulletin board implements a counter. Each time a string is published on the bulletin board, the counter is incremented and the authentication tag is produced on the string-counter pair. While the counter value of the bulletin board is assumed to be publicly accessible, we model is as an explicit query. We model the bulletin board as an ideal functionality $\mathcal{F}_{\mathsf{BB}}$ as described in Figure 8. The bulletin board abstraction can be instantiated using fork-less blockchains, such as permissioned blockchains and potentially even by blockchains based on proof-of-stake [AAB+19, ABB+18].

4 Synchronizable Exchange in the $(\mathcal{G}_{\mathsf{att}}, \mathcal{F}_{\mathsf{BB}})$-Hybrid Model

In this section, we show how the ideal functionality for synchronizable exchange $\mathcal{F}_{\mathsf{SyX}}$ can be realized in the $(\mathcal{G}_{\mathsf{att}}, \mathcal{F}_{\mathsf{BB}})$-hybrid model. The idea for the construction is the following. Each party loads their inputs to $\mathcal{F}_{\mathsf{SyX}}$ into their own instances of a secure attested execution processor. The processors then exchange the inputs that have loaded into them. In more detail, the processors generate authenticated encryptions of the inputs which the parties exchange and feed into their respective processors. Once the exchange is complete, one of the processors (arbitrary) generates an authenticated encryption of the output of the load phase of $\mathcal{F}_{\mathsf{SyX}}$. The party that obtains this ciphertext then publishes it using the ideal functionality $\mathcal{F}_{\mathsf{BB}}$ and obtains a proof that it did so. Note that this proof also tracks the time at which the ciphertext was published. At this point, if both parties behaved honestly, they are both in a position to obtain the output of the load phase. The party that obtained a proof from $\mathcal{F}_{\mathsf{BB}}$ feeds the proof into its processor which then ensures that the ciphertext was posted correctly and within the input timeout and then releases the output of the load phase in the clear. The other party obtains the ciphertext from the ideal functionality $\mathcal{F}_{\mathsf{BB}}$ and feeds it into its processor which then decrypts it and releases the output of the load phase in the clear, provided the ciphertext was valid and was posted within the input timeout.

When a party wishes to trigger $\mathcal{F}_{\mathsf{SyX}}$, it feeds the triggering witness into its processor which then checks that the witness is valid and that the trigger timeout has not elapsed. If so, the processor generates an authenticated encryption of the output of the trigger phase of $\mathcal{F}_{\mathsf{SyX}}$. The party then publishes the ciphertext using the ideal functionality $\mathcal{F}_{\mathsf{BB}}$ and obtains a proof that it did so. The party then feeds the proof into its processor which then ensures that the ciphertext was posted correctly and within the trigger timeout and then releases the output of the trigger phase in the clear. The other party obtains the ciphertext from the ideal functionality $\mathcal{F}_{\mathsf{BB}}$ and feeds it into its processor which then decrypts it and releases the output of the trigger phase in the clear, provided the ciphertext was valid and was posted within the trigger timeout.

We formalize this construction in Figures 9, 10, 11 and 12.

Theorem 1. *Assuming that DDH holds in the relevant group, $\mathcal{E}$ is perfectly correct, and satisfies semantic security and INT-CTXT security, and that $\mathcal{T}$ and $\mathcal{V}$ are existentially unforgeable, it holds that the protocol described in Figures 9, 10, 11 and 12 securely realizes $\mathcal{F}_{\mathsf{SyX}}$ with guaranteed output delivery.*

Proof. We now prove the above theorem. When both parties are honest, it is not difficult to construct a simulation. We focus on the more interesting case when one party is corrupt. First, we consider the case when P_1 is honest and P_2 is corrupt.

We can now construct a simulator $\mathcal{S}$ as described below.

- Unless otherwise noted later, $\mathcal{S}$ passes through messages between P_2 and $\mathcal{G}_{\mathsf{att}}$.
- $\mathcal{S}$ calls $eid_1 := \mathcal{G}_{\mathsf{att}}.\mathsf{install}(sid, \mathsf{prog}_{\mathsf{SyX},1}[P_1])$, and $(g^{y_1}, \sigma_1) := \mathcal{G}_{\mathsf{att}}.\mathsf{resume}(eid_1, (\text{“keyex”}))$ and sends $(eid_1, g^{y_1}, \sigma_1)$ to P_2. $\mathcal{S}$ waits to receive the first message $(eid_2, g^{y_2}, \sigma_2)$ from P_2 – if this tuple was not the answer to a previous $\mathcal{G}_{\mathsf{att}}$ query, jump to the exception handler denoted **except**. At this point, eid_2 is called the challenge eid.
- $\mathcal{S}$ checks that $\mathcal{V}.\mathsf{Verify}(\sigma_2, (sid, eid_2, \mathsf{prog}_{\mathsf{SyX},2}[P_2], g^{y_2}); \mathsf{vk}_{\mathsf{att}})$ succeeds. If not, jump to the exception handler denoted **except**.
- $\mathcal{S}$ calls $(\mathsf{ct}_1, \_) := \mathcal{G}_{\mathsf{att}}.\mathsf{resume}(eid_1, (\text{“send”}, g^{y_2}, \vec{0}))$ and sends ct_1 to P_2.
- The first time P_2 calls $\mathcal{G}_{\mathsf{att}}.\mathsf{resume}(eid_2, (\text{“loadstart”}, \mathsf{inp}_2, \mathsf{ct}_1))$ for some input inp_2 where eid_2 is the challenge eid, and ct_1 is what $\mathcal{S}$ has sent, the simulator $\mathcal{S}$ extracts and saves inp_2.
- $\mathcal{S}$ waits to receive the message ct_2 from P_2. If $(\mathsf{ct}_2, \_)$ is the not the result of the first $\mathcal{G}_{\mathsf{att}}.\mathsf{resume}(eid_2, (\text{“send”}, g^{y_1}))$ call where eid_2 is the challenge eid, and g^{y_1} was what $\mathcal{S}$ previously sent to P_2, or if no such call has taken place, then jump to the exception handler denoted **except**.

$\mathsf{prog}_{\mathsf{SyX},1}[P_1]$

On input ("keyex"): $y_1 \xleftarrow{\$} \mathbb{Z}_p$, return g^{y_1}

On input ("send", g^{y_2}, inp_1): assert "keyex" has been called, $\mathsf{sk} := (g^{y_2})^{y_1}$, $\mathsf{ct} := \mathcal{E}.\mathsf{Enc}(\mathsf{inp}_1; \mathsf{sk})$, return ct

On input ("loadstart", ct_2):
assert "send" has been called, assert ct_2 not seen
$\mathsf{inp}_2 := \mathcal{E}.\mathsf{Dec}(\mathsf{ct}_2; \mathsf{sk})$, assert that decryption succeeds
parse $\mathsf{inp}_1 := (sid, \mathsf{INPUT\_TIMEOUT}, \mathsf{TRIGGER\_TIMEOUT}, x_1, f)$
parse $\mathsf{inp}_2 := (sid', \mathsf{INPUT\_TIMEOUT}', \mathsf{TRIGGER\_TIMEOUT}', x_2, f')$
assert that: $sid = sid'$, $f = f'$, $\mathsf{INPUT\_TIMEOUT} = \mathsf{INPUT\_TIMEOUT}'$, and
$\mathsf{TRIGGER\_TIMEOUT} = \mathsf{TRIGGER\_TIMEOUT}'$
assert $r \leq \mathsf{INPUT\_TIMEOUT}$, parse $f = (f_1, f_2, \phi_1, \phi_2)$
$z_1^1 = f_1^1(x_1, x_2, r)$, $z_1^2 = f_1^2(x_1, x_2, r)$, $\mathsf{ct}_{\mathsf{load}} := \mathcal{E}.\mathsf{Enc}(z_1^2; \mathsf{sk})$, return $(\mathsf{ct}_{\mathsf{load}}$, "ok")

On input ("loadfinish", $\sigma_3, t_{\mathsf{load}}, \sigma_4, v$):
if $v \neq \perp$, return (v, "ok")
assert "loadstart" has been called and returned ($\cdot$, "ok")
assert $\mathcal{V}.\mathsf{Verify}(\sigma_3, (sid, eid_1, \mathsf{prog}_{\mathsf{SyX},1}[P_1], (\mathsf{ct}_{\mathsf{load}}, \text{"ok"})); \mathsf{vk}_{\mathsf{att}})$
assert $\mathcal{T}.\mathsf{Verify}(\sigma_4, t_{\mathsf{load}} \| \text{"load"} \| sid \| \mathsf{ct}_{\mathsf{load}} \| \sigma_3)$
assert $t_{\mathsf{load}} \leq \mathsf{INPUT\_TIMEOUT}$, set $i_1 := 0$, $i_2 := 0$, return (z_1^1, "ok")

On input* ("triggerstart", w_1):
assert "loadfinish" has been called and returned ($\cdot$, "ok")
assert $i_1 = 0$ or "triggerfinish" has been called and returned ($\cdot, i_1 - 1$, "ok")
and "triggerstart" has never previously returned ($\cdot, i_1$, "ok")
assert $\phi_1(w_1) = 1$ and that $r \leq \mathsf{TRIGGER\_TIMEOUT}$
$i_1 := i_1 + 1$, $z_{2,i_1} := f_2(x_1, x_2, w_1, r)$, $\mathsf{ct}_{\mathsf{trigger1},i_1} := \mathcal{E}.\mathsf{Enc}(z_{2,i_1}; \mathsf{sk})$
return ($\mathsf{ct}_{\mathsf{trigger1},i_1}, i_1$, "ok")

On input* ("triggerfinish", $\sigma_{5,i_1}, t_{\mathsf{trigger1},i_1}, \sigma_{6,i_1}$):
assert "triggerstart" has been called and returned ($\cdot, i_1$, "ok")
assert "triggerfinish" has never previously returned ($\cdot, i_1$, "ok")
assert $\mathcal{V}.\mathsf{Verify}(\sigma_{5,i_1}, (sid, eid_1, \mathsf{prog}_{\mathsf{SyX},1}[P_1], (\mathsf{ct}_{\mathsf{trigger1},i_1}, i_1, \text{"ok"})); \mathsf{vk}_{\mathsf{att}})$
assert $\mathcal{T}.\mathsf{Verify}(\sigma_{6,i_1}, t_{\mathsf{trigger1},i_1} \| \text{"trigger1"} \| sid \| i_1 \| \mathsf{ct}_{\mathsf{trigger1},i_1} \| \sigma_{5,i_1})$
assert $t_{\mathsf{trigger1},i_1} \leq \mathsf{TRIGGER\_TIMEOUT}$, return ($z_{2,i_1}, i_1$, "ok")

On input* ("triggerbyother", $\mathsf{ct}_{\mathsf{trigger2},i_2+1}, \sigma_{7,i_2+1}, t_{\mathsf{trigger2},i_2+1}, \sigma_{8,i_2+1}, v$):
if $v \neq \perp$, $i_2 := i_2 + 1$, return (v, i_2, "ok")
assert "loadfinish" has been called and returned ($\cdot$, "ok")
assert $i_2 = 0$ or "triggerbyother" has been called and returned ($\cdot, i_2 - 1$, "ok")
and "triggerbyother" has never previously returned ($\cdot, i_2$, "ok")
assert $\mathcal{V}.\mathsf{Verify}(\sigma_{7,i_2+1}, (sid, eid_2, \mathsf{prog}_{\mathsf{SyX},2}[P_2], (\mathsf{ct}_{\mathsf{trigger2},i_2+1}, i_2 + 1, \text{"ok"})); \mathsf{vk}_{\mathsf{att}})$, and
$\mathcal{T}.\mathsf{Verify}(\sigma_{8,i_2+1}, t_{\mathsf{trigger2},i_2+1} \| \text{"trigger2"} \| sid \| (i_2 + 1) \| \mathsf{ct}_{\mathsf{trigger2},i_2+1} \| \sigma_{7,i_2+1})$
assert $t_{\mathsf{trigger2},i_2+1} \leq \mathsf{TRIGGER\_TIMEOUT}$
$z_{3,i_2+1} := \mathcal{E}.\mathsf{Dec}(\mathsf{ct}_{\mathsf{trigger2},i_2+1}; \mathsf{sk})$, assert that decryption succeeds, $i_2 := i_2 + 1$, return (z_{3,i_2}, i_2, "ok")

Figure 9: Program installed in the secure hardware of Party P_1.

$\mathsf{prog}_{\mathsf{SyX},2}[P_2]$

On input ("keyex"): $y_2 \xleftarrow{\$} \mathbb{Z}_p$, return g^{y_2}

On input ("loadstart", $\mathsf{inp}_2, \mathsf{ct}_1$):
assert "keyex" has been called, assert ct_1 not seen
$\mathsf{inp}_1 := \mathcal{E}.\mathsf{Dec}(\mathsf{ct}_1; \mathsf{sk})$, assert that decryption succeeds
parse $\mathsf{inp}_1 := (sid, \mathsf{INPUT\_TIMEOUT}, \mathsf{TRIGGER\_TIMEOUT}, x_1, f)$
parse $\mathsf{inp}_2 := (sid', \mathsf{INPUT\_TIMEOUT}', \mathsf{TRIGGER\_TIMEOUT}', x_2, f')$
assert that: $sid = sid'$, $f = f'$, $\mathsf{INPUT\_TIMEOUT} = \mathsf{INPUT\_TIMEOUT}'$, and
$\mathsf{TRIGGER\_TIMEOUT} = \mathsf{TRIGGER\_TIMEOUT}'$
assert $r \leq \mathsf{INPUT\_TIMEOUT}$, parse $f = (f_1, f_2, \phi_1, \phi_2)$, return "ok"

On input ("send", g^{y_1}):
assert "loadstart" has been called and returned "ok", $\mathsf{sk} := (g^{y_1})^{y_2}$, $\mathsf{ct} := \mathcal{E}.\mathsf{Enc}(\mathsf{inp}_2; \mathsf{sk})$, return ct

On input ("loadfinish", $\mathsf{ct}_{\mathsf{load}}, \sigma_3, t_{\mathsf{load}}, \sigma_4, v$):
if $v \neq \perp$, return (v, "ok")
assert "loadstart" has been called and returned "ok", assert $\mathsf{ct}_{\mathsf{load}}$ not seen
assert $\mathcal{V}.\mathsf{Verify}(\sigma_3, (sid, eid_1, \mathsf{prog}_{\mathsf{SyX},1}[P_1], (\mathsf{ct}_{\mathsf{load}}, \text{"ok"})); \mathsf{vk}_{\mathsf{att}})$
assert $\mathcal{T}.\mathsf{Verify}(\sigma_4, t_{\mathsf{load}} \| \text{"load"} \| sid \| \mathsf{ct}_{\mathsf{load}} \| \sigma_3)$
assert $t_{\mathsf{load}} \leq \mathsf{INPUT\_TIMEOUT}$
$z_1^2 := \mathcal{E}.\mathsf{Dec}(\mathsf{ct}_{\mathsf{load}}; \mathsf{sk})$, assert that decryption succeeds, set $i_1 := 0$, $i_2 := 0$, return (z_1^2, "ok")

On input* ("triggerstart", w_2):
assert "loadfinish" has been called and returned ($\cdot$, "ok")
assert $i_2 = 0$ or "triggerfinish" has been called and returned ($\cdot, i_2 - 1$, "ok")
and "triggerstart" has never previously returned ($\cdot, i_2$, "ok")
assert $\phi_2(w_2) = 1$ and that $r \leq \mathsf{TRIGGER\_TIMEOUT}$
$i_2 := i_2 + 1$, $z_{3,i_2} := f_2(x_1, x_2, w_2, r)$, $\mathsf{ct}_{\mathsf{trigger2},i_2} := \mathcal{E}.\mathsf{Enc}(z_{3,i_2}; \mathsf{sk})$, return ($\mathsf{ct}_{\mathsf{trigger2},i_2}, i_2$, "ok")

On input* ("triggerfinish", $\sigma_{7,i_2}, t_{\mathsf{trigger2},i_2}, \sigma_{8,i_2}$):
assert "triggerstart" has been called and returned ($\cdot, i_2$, "ok")
assert "triggerfinish" has never previously returned ($\cdot, i_2$, "ok")
assert $\mathcal{V}.\mathsf{Verify}(\sigma_{7,i_2}, (sid, eid_2, \mathsf{prog}_{\mathsf{SyX},2}[P_2], (\mathsf{ct}_{\mathsf{trigger2},i_2}, i_2, \text{"ok"})); \mathsf{vk}_{\mathsf{att}})$
assert $\mathcal{T}.\mathsf{Verify}(\sigma_{8,i_2}, t_{\mathsf{trigger2},i_2} \| \text{"trigger2"} \| sid \| i_2 \| \mathsf{ct}_{\mathsf{trigger2},i_2} \| \sigma_{7,i_2})$
assert $t_{\mathsf{trigger2},i_2} \leq \mathsf{TRIGGER\_TIMEOUT}$, return ($z_{3,i_2}, i_2$, "ok")

On input* ("triggerbyother", $\mathsf{ct}_{\mathsf{trigger1},i_1+1}, \sigma_{5,i_1+1}, t_{\mathsf{trigger1},i_1+1}, \sigma_{6,i_1+1}, v$):
if $v \neq \perp$, $i_1 := i_1 + 1$, return (v, i_1, "ok")
assert "loadfinish" has been called and returned ($\cdot$, "ok")
assert $i_1 = 0$ or "triggerbyother" has been called and returned ($\cdot, i_1 - 1$, "ok")
and "triggerbyother" has never previously returned ($\cdot, i_1$, "ok")
assert $\mathcal{V}.\mathsf{Verify}(\sigma_{5,i_1+1}, (sid, eid_1, \mathsf{prog}_{\mathsf{SyX},1}[P_1], (\mathsf{ct}_{\mathsf{trigger1},i_1+1}, i_1 + 1, \text{"ok"})); \mathsf{vk}_{\mathsf{att}})$, and
$\mathcal{T}.\mathsf{Verify}(\sigma_{6,i_1+1}, t_{\mathsf{trigger1},i_1+1} \| \text{"trigger1"} \| sid \| (i_1 + 1) \| \mathsf{ct}_{\mathsf{trigger1},i_1+1} \| \sigma_{5,i_1+1})$
assert $t_{\mathsf{trigger1},i_1+1} \leq \mathsf{TRIGGER\_TIMEOUT}$
$z_{2,i_1+1} := \mathcal{E}.\mathsf{Dec}(\mathsf{ct}_{\mathsf{trigger1},i_1+1}; \mathsf{sk})$, assert that decryption succeeds, $i_1 := i_1 + 1$, return (z_{2,i_1}, i_1, "ok")

Figure 10: Program installed in the secure hardware of Party P_2.

$$\mathbf{Prot}_{\mathsf{SyX},1}[sid, P_1]$$

$\mathsf{inp}_1 := (sid, \mathsf{INPUT\_TIMEOUT}, \mathsf{TRIGGER\_TIMEOUT}, x_1, f = (f_1, f_2, \phi_1, \phi_2))$

Initialization

$eid_1 := \mathcal{G}_{\mathsf{att}}.\mathsf{install}(sid, \mathsf{prog}_{\mathsf{SyX},1}[P_1])$
henceforth denote $\mathcal{G}_{\mathsf{att}}.\mathsf{resume}(\cdot) := \mathcal{G}_{\mathsf{att}}.\mathsf{resume}(eid_1, \cdot)$
$(g^{y_1}, \sigma_1) := \mathcal{G}_{\mathsf{att}}.\mathsf{resume}(\text{"keyex"})$
send $(eid_1, g^{y_1}, \sigma_1)$ to P_2, await $(eid_2, g^{y_2}, \sigma_2)$
assert $\mathcal{V}.\mathsf{Verify}(\sigma_2, (sid, eid_2, \mathsf{prog}_{\mathsf{SyX},2}[P_2], g^{y_2}); \mathsf{vk}_{\mathsf{att}})$

Load Phase

$(\mathsf{ct}_1, \_) := \mathcal{G}_{\mathsf{att}}.\mathsf{resume}(\text{"send"}, g^{y_2}, \mathsf{inp}_1)$, send ct_1 to P_2, await ct_2
$(\mathsf{ct}_{\mathsf{load}}, \text{"ok"}, \sigma_3) := \mathcal{G}_{\mathsf{att}}.\mathsf{resume}(\text{"loadstart"}, \mathsf{ct}_2)$
$(t_{\mathsf{load}}, \sigma_4) := \mathcal{F}_{\mathsf{BB}}.\mathsf{post}(\text{"load"} \| sid \| \mathsf{ct}_{\mathsf{load}} \| \sigma_3)$
$(z_1^1, \text{"ok"}, \_) := \mathcal{G}_{\mathsf{att}}.\mathsf{resume}(\text{"loadfinish"}, \sigma_3, t_{\mathsf{load}}, \sigma_4, \perp)$, set $i_1 := 0$, $i_2 := 0$

Trigger Phase

Triggers by P_1

$$\left\{\begin{array}{l} i_1 := i_1 + 1 \\ (\mathsf{ct}_{\mathsf{trigger1},i_1}, i_1, \text{"ok"}, \sigma_{5,i_1}) := \mathcal{G}_{\mathsf{att}}.\mathsf{resume}(\text{"triggerstart"}, w_1) \\ (t_{\mathsf{trigger1},i_1}, \sigma_{6,i_1}) := \mathcal{F}_{\mathsf{BB}}.\mathsf{post}(\text{"trigger1"} \| sid \| i_1 \| \mathsf{ct}_{\mathsf{trigger1},i_1} \| \sigma_{5,i_1}) \\ (z_{2,i_1}, i_1, \text{"ok"}, \_) := \mathcal{G}_{\mathsf{att}}.\mathsf{resume}(\text{"triggerfinish"}, \sigma_{5,i_1}, t_{\mathsf{trigger1},i_1}, \sigma_{6,i_1}) \end{array}\right\}$$

Triggers by P_2

$$\left\{\begin{array}{l} \text{obtain a } t_{\mathsf{trigger2},i_2+1} \leq \mathsf{TRIGGER\_TIMEOUT} \text{ such that} \\ \qquad (\text{"trigger2"} \| sid \| (i_2+1) \| \mathsf{ct}_{\mathsf{trigger2},i_2+1} \| \sigma_{7,i_2+1}, \sigma_{8,i_2+1}) := \mathcal{F}_{\mathsf{BB}}.\mathsf{getContent}(t_{\mathsf{trigger2},i_2+1}) \\ (z_{3,i_2+1}, i_2+1, \text{"ok"}, \_) := \mathcal{G}_{\mathsf{att}}.\mathsf{resume}(\text{"triggerbyother"}, \mathsf{ct}_{\mathsf{trigger2},i_2+1}, \sigma_{7,i_2+1}, t_{\mathsf{trigger2},i_2+1}, \sigma_{8,i_2+1}) \\ i_2 := i_2 + 1 \end{array}\right\}$$

Figure 11: Protocol executed by Party P_1 in realizing $\mathcal{F}_{\mathsf{SyX}}$.

$$\mathbf{Prot}_{\mathsf{SyX},2}[sid, P_2]$$

$\mathsf{inp}_2 := (sid, \mathsf{INPUT\_TIMEOUT}, \mathsf{TRIGGER\_TIMEOUT}, x_2, f = (f_1, f_2, \phi_1, \phi_2))$

Initialization

$eid_2 := \mathcal{G}_{\mathsf{att}}.\mathsf{install}(sid, \mathsf{prog}_{\mathsf{SyX},2}[P_2])$
henceforth denote $\mathcal{G}_{\mathsf{att}}.\mathsf{resume}(\cdot) := \mathcal{G}_{\mathsf{att}}.\mathsf{resume}(eid_2, \cdot)$
$(g^{y_2}, \sigma_2) := \mathcal{G}_{\mathsf{att}}.\mathsf{resume}(\text{“keyex”})$
send $(eid_2, g^{y_2}, \sigma_2)$ to P_1, await $(eid_1, g^{y_1}, \sigma_1)$
assert $\mathcal{V}.\mathsf{Verify}(\sigma_1, (sid, eid_1, \mathsf{prog}_{\mathsf{SyX},1}[P_1], g^{y_1}); \mathsf{vk}_{\mathsf{att}})$

Load Phase

Await ct_1
$(\text{“ok”}, \_) := \mathcal{G}_{\mathsf{att}}.\mathsf{resume}(\text{“loadstart”}, \mathsf{inp}_2, \mathsf{ct}_1)$
$(\mathsf{ct}_2, \_) := \mathcal{G}_{\mathsf{att}}.\mathsf{resume}(\text{“send”}, g^{y_1})$, send ct_2 to P_1
obtain a $t_{\mathsf{load}} \leq \mathsf{INPUT\_TIMEOUT}$ such that
$(\text{“load”} \| sid \| \mathsf{ct}_{\mathsf{load}} \| \sigma_3, \sigma_4) := \mathcal{F}_{\mathsf{BB}}.\mathsf{getContent}(t_{\mathsf{load}})$
$(z_1^2, \text{“ok”}, \_) := \mathcal{G}_{\mathsf{att}}.\mathsf{resume}(\text{“loadfinish”}, \mathsf{ct}_{\mathsf{load}}, \sigma_3, t_{\mathsf{load}}, \sigma_4, \perp)$, set $i_1 := 0$, $i_2 := 0$

Trigger Phase

Triggers by P_2

$$\left\{\begin{array}{l} i_2 := i_2 + 1 \\ (\mathsf{ct}_{\mathsf{trigger2},i_2}, i_2, \text{“ok”}, \sigma_{7,i_2}) := \mathcal{G}_{\mathsf{att}}.\mathsf{resume}(\text{“triggerstart”}, w_2) \\ (t_{\mathsf{trigger2},i_2}, \sigma_{8,i_2}) := \mathcal{F}_{\mathsf{BB}}.\mathsf{post}(\text{“trigger2”} \| sid \| i_2 \| \mathsf{ct}_{\mathsf{trigger2},i_2} \| \sigma_{7,i_2}) \\ (z_{3,i_2}, i_2, \text{“ok”}, \_) := \mathcal{G}_{\mathsf{att}}.\mathsf{resume}(\text{“triggerfinish”}, \sigma_{7,i_2}, t_{\mathsf{trigger2},i_2}, \sigma_{8,i_2}) \end{array}\right\}$$

Triggers by P_1

$$\left\{\begin{array}{l} \text{obtain a } t_{\mathsf{trigger1},i_1+1} \leq \mathsf{TRIGGER\_TIMEOUT} \text{ such that} \\ \qquad (\text{“trigger1”} \| sid \| (i_1+1) \| \mathsf{ct}_{\mathsf{trigger1},i_1+1} \| \sigma_{5,i_1+1}, \sigma_{6,i_1+1}) := \mathcal{F}_{\mathsf{BB}}.\mathsf{getContent}(t_{\mathsf{trigger1},i_2+1}) \\ (z_{2,i_1+1}, i_1+1, \text{“ok”}, \_) := \mathcal{G}_{\mathsf{att}}.\mathsf{resume}(\text{“triggerbyother”}, \mathsf{ct}_{\mathsf{trigger1},i_1+1}, \sigma_{5,i_1+1}, t_{\mathsf{trigger1},i_1+1}, \sigma_{6,i_1+1}) \\ i_1 := i_1 + 1 \end{array}\right\}$$

Figure 12: Protocol executed by Party P_2 in realizing $\mathcal{F}_{\mathsf{SyX}}$.

- At this point, the load phase of $\mathcal{F}_{\mathsf{SyX}}$ has been completed. $\mathcal{S}$ sends inp_2[19] to the trusted party computing $\mathcal{F}_{\mathsf{SyX}}$ with guaranteed output delivery. It receives the corrupt party's output for the load phase, namely, z_1^2.

- $\mathcal{S}$ calls $(\mathsf{ct}_{\mathsf{load}}, \text{"ok"}, \sigma_3) := \mathcal{G}_{\mathsf{att}}.\mathsf{resume}(eid_1, (\text{"loadstart"}, \mathsf{ct}_2))$ and. $\mathcal{S}$ then calls $(t_{\mathsf{load}}, \sigma_4) := \mathcal{F}_{\mathsf{BB}}.\mathsf{post}(\text{"load"} \| sid \| \mathsf{ct}_{\mathsf{load}} \| \sigma_3)$.

- When $\mathcal{S}$ receives $\mathcal{G}_{\mathsf{att}}.\mathsf{resume}(eid_2, (\text{"loadfinish"}, \mathsf{ct}_{\mathsf{load}}, \sigma_3, t_{\mathsf{load}}, \sigma_4, v))$ from P_2, if $v \neq \perp$, pass through the call. Else, when $\mathcal{S}$ first receives such a call with $v = \perp$, $\mathcal{S}$ calls $(z_1^2, \text{"ok"}, \sigma_{\mathsf{sim},1}) := \mathcal{G}_{\mathsf{att}}.\mathsf{resume}(eid_1, (\text{"loadfinish"}, \sigma_3, t_{\mathsf{load}}, \sigma_4, z_1^2))$ and returns $(z_1^2, \text{"ok"}, \sigma_{\mathsf{sim},1})$.

Hybrid 0. Identical to the simulation, except that every occurrence of the challenge $\mathsf{sk} = g^{y_1 y_2}$ is replaced with a random key.

Claim 1. *Assume that DDH holds, then Hybrid 0 is computationally indistinguishable from the simulation.*

Proof. Straightforward reduction to DDH security. □

Hybrid 1. Identical to Hybrid 0, except that every time the exception handler is triggered in the simulation, if the real-world P_1 would not have had an assertion failure or awaited a message that did not arrive at the end of a round, abort the simulation.

Claim 2. *Assume that $\mathcal{T}$ and $\mathcal{V}$ are existentially unforgeable and that $\mathcal{E}$ has INT-CTXT security, then Hybrid 1 aborts with negligible probability.*

Proof. If the exception handler is triggered in the simulation, and the real-world P_1 did not have a signature verification failure or a ct-related failure (that is, either ct was seen before or decryption of ct did not succeed or yield the expected result), then one can easily leverage P_2 to build a reduction that either breaks the unforgeability of $\mathcal{T}$, $\mathcal{V}$ or the INT-CTXT security of $\mathcal{E}$. □

Hybrid 2. Identical to Hybrid 1, except that encryption of the $\vec{0}$ vector is replaced with encryption of the honest client's true input.

Claim 3. *Assume that $\mathcal{E}$ is semantically secure, then Hybrid 2 is computationally indistinguishable from Hybrid 1.*

Proof. Straightforward reduction to the semantic security of $\mathcal{E}$. □

[19] We note that the format of $\mathsf{inp}_2 := (sid', \mathsf{INPUT\_TIMEOUT}', \mathsf{TRIGGER\_TIMEOUT}', x_2, f')$ is different from the input format that the ideal functionality $\mathcal{F}_{\mathsf{SyX}}$ described in Figure 5 expects to receive. In particular, the differences are that there is a new variable representing the session identifier sid and that the round numbers corresponding to the time outs $\mathsf{INPUT\_TIMEOUT}', \mathsf{TRIGGER\_TIMEOUT}'$ are not pre-programmed but are part of the input to the functionality itself. These syntactical differences are merely to make the functionality presented in Figure 5 more readable and do not have any impact on the correctness of the definition or realization. In other words, the ideal functionality defined in Figure 5 can be readily modified to expect inputs of the form $\mathsf{inp}_2 := (sid', \mathsf{INPUT\_TIMEOUT}', \mathsf{TRIGGER\_TIMEOUT}', x_2, f')$. We omit the details for simplicity here.

Hybrid 3. Identical to Hybrid 2, except that the challenge sk is now replaced with the true $g^{y_1 y_2}$ again.

Claim 4. *Assume that DDH holds, then Hybrid 3 is computationally indistinguishable from Hybrid 2.*

Proof. By straightforward reduction to DDH security. □

Claim 5. *Conditioned on simulation not aborting, Hybrid 3 is identically distributed as the real execution.*

Proof. Straightforward to observe. □

□

Combining the above Theorem 1 and Lemma 9, we obtain the following theorem.

Theorem 2. *Consider n parties $P_1, \ldots, P_n$ in the* point-to-point *model. Then, assuming the existence of one-way functions and key-exchange, there exists a protocol π which securely computes $\mathcal{F}_{\mathsf{MPC}}$ with fairness in the presence of t-threshold adversaries for any $0 \leq t < n$ in the $(\mathcal{G}_{\mathsf{att}}, \mathcal{F}_{\mathsf{BB}})$-hybrid model.*

5 Synchronizable Exchange with One-Sided Triggers

In this work, we consider the primitive of synchronizable exchange where only one of the parties may trigger the ideal functionality in the trigger phase. In other words, we inspect the power of the primitive where either $\phi_1 \equiv 0$ or $\phi_2 \equiv 0$ (as defined in Figures 4 and 5). The motivation for this is the following. As noted in Section 4, it is possible to realize synchronizable exchange in the $(\mathcal{G}_{\mathsf{att}}, \mathcal{F}_{\mathsf{BB}})$-hybrid model. Furthermore, only a party that wishes to trigger the ideal functionality in the trigger phase needs to possess an instance of secure hardware. Since, we would view such secure hardware as a resource whose requirement we would like to minimize, we in turn model the ability to trigger the ideal functionality in the trigger phase as a resource whose requirement we would like to minimize. If neither of the parties may trigger the functionality in the trigger phase, that is, if $\phi_1 = \phi_2 \equiv 0$, then, the functionality is equivalent to the ideal functionality $\mathcal{F}_{\mathsf{2PC}}$ ($\mathcal{F}_{\mathsf{MPC}}$ for $n = 2$ parties; refer Figure 2). However, it is unclear what the power of the primitive is when either $\phi_1 \equiv 0$ or $\phi_2 \equiv 0$ but not both.

In the context of a multiparty network, this allows to interpret things in another way. It is easy to see that parties may use a single instance of secure hardware and yet interact with multiple instances of the ideal functionality $\mathcal{F}_{\mathsf{SyX}}$ and trigger all of them in their respective trigger phases. Thus, the only parties that need to possess instances of secure hardware are those that wish to trigger some instance of the ideal functionality $\mathcal{F}_{\mathsf{SyX}}$. We can thus model the ability to trigger an instance of the ideal functionality $\mathcal{F}_{\mathsf{SyX}}$ in its trigger phase as a resource whose requirement we would like to minimize. This would in turn minimize the number of parties that would need to possess an instance of secure hardware while realizing our protocols in the $(\mathcal{G}_{\mathsf{att}}, \mathcal{F}_{\mathsf{BB}})$-hybrid model.

5.1 Synchronizable Exchange with One-Sided Triggers in the $(\mathcal{G}_{\mathsf{att}}, \mathcal{G}_{\mathsf{acrs}}, \mathcal{F}_{\mathsf{BB}})$-Hybrid Model

We show how the ideal functionality for synchronizable exchange $\mathcal{F}_{\mathsf{SyX}}$ with one-sided trigger can be realized in the $(\mathcal{G}_{\mathsf{att}}, \mathcal{G}_{\mathsf{acrs}}, \mathcal{F}_{\mathsf{BB}})$-hybrid model. The idea for the construction is the following. The party not possessing the secure attestation processor loads its input into the secure attestation processor of the other party. In more detail, the processor generate a key to which the other party uses to encrypt its input, along with a key for an authenticated encryption scheme that the two would now share. The processor waits to receive an authenticated encryption of the message "continue" from the the other party which is meant to indicate that the right ciphertext has been fed into the processor, that is, the processor now correctly possesses the other party's input. Now, as in Section 4, the processor generates an authenticated encryption of the output of the load phase of $\mathcal{F}_{\mathsf{SyX}}$. The party that obtains this ciphertext then publishes it using the ideal functionality $\mathcal{F}_{\mathsf{BB}}$ and obtains a proof that it did so. Note that this proof also tracks the time at which the ciphertext was published. At this point, if both parties behaved honestly, they are both in a position to obtain the output of the load phase. The party that obtained a proof from $\mathcal{F}_{\mathsf{BB}}$ feeds the proof into its processor which then ensures that the ciphertext was posted correctly and within the input timeout and then releases the output of the load phase in the clear. The other party obtains the ciphertext from the ideal functionality $\mathcal{F}_{\mathsf{BB}}$ and decrypts to obtain the output of the load phase.

When the party with the processor wishes to trigger $\mathcal{F}_{\mathsf{SyX}}$, it feeds the triggering witness into its processor which then checks that the witness is valid and that the trigger timeout has not elapsed. If so, the processor generates an authenticated encryption of the output of the trigger phase of $\mathcal{F}_{\mathsf{SyX}}$. The party then publishes the ciphertext using the ideal functionality $\mathcal{F}_{\mathsf{BB}}$ and obtains a proof that it did so. The party then feeds the proof into its processor which then ensures that the ciphertext was posted correctly and within the trigger timeout and then releases the output of the trigger phase in the clear. The other party obtains the ciphertext from the ideal functionality $\mathcal{F}_{\mathsf{BB}}$ and decrypts to obtain the output of the trigger phase.

We formalize this construction in Figures 13, 14 and 15.

Theorem 3. *Assuming that DDH holds in the relevant group, $\mathcal{E}_1$ is perfect correct, and satisfies semantic security, $\mathcal{E}_2$ is perfectly correct, and satisfies semantic security and INT-CTXT security, the proof system satisfies computational soundness and witness indistinguishability, and that $\mathcal{T}$ and $\mathcal{V}$ are existentially unforgeable, it holds that the protocol described in Figures 13, 14 and 15 securely realizes $\mathcal{F}_{\mathsf{SyX}}$ with one-sided trigger with guaranteed output delivery.*

6 Fair Secure Computation using t instances of secure hardware

In this section, we show how the ideal functionality for secure computation $\mathcal{F}_{\mathsf{MPC}}$ can be realized in the $(\mathcal{G}_{\mathsf{att}}, \mathcal{F}_{\mathsf{BB}})$-hybrid model where only t of the parties have access to an instance of secure hardware. We do so by designing a protocol that realizes $\mathcal{F}_{\mathsf{MPC}}$ in the $\mathcal{F}_{\mathsf{SyX}}$-hybrid model where only t of the parties can trigger an instance of the ideal functionality $\mathcal{F}_{\mathsf{SyX}}$. In order to gain some intuition, we present the warm-up case of three parties.

6.1 The case $n = 3$

We consider the case where $n = 3$ and $t = 2$ ($t < 2$ is an honest majority). Let P_1, P_2, and P_3 be the three parties with inputs x_1, x_2 and x_3 respectively. For $i, j \in \{1, 2, 3\}$ with $i < j$, we have that

$$\mathsf{prog}_{\text{one-sided-SyX},1}[\mathcal{G}_{\mathsf{acrs}}.\mathsf{mpk}, P_1]$$

On input ("init"): $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathcal{E}_1.\mathsf{Gen}(1^\lambda)$, return pk

On input ("extract", idk):
  if $\mathsf{check}(\mathcal{G}_{\mathsf{acrs}}.\mathsf{mpk}, P_2, \mathsf{idk}) = 1$, $v := \mathsf{sk}$, else $v := \perp$, return v

On input ("loadstart", ct_2): $(\mathsf{inp}_2, k) := \mathcal{E}_1.\mathsf{Dec}(\mathsf{ct}_2; \mathsf{sk})$, return ct_2

On input ("loadcontinue", $\mathsf{ct}_{\mathsf{continue}}, \mathsf{inp}_1$):
  assert "loadstart" has been called
  assert $\mathcal{E}_2.\mathsf{Dec}(\mathsf{ct}_{\mathsf{continue}}; k) =$ "continue", assert that decryption succeeds
  parse $\mathsf{inp}_1 := (sid, \mathsf{INPUT\_TIMEOUT}, \mathsf{TRIGGER\_TIMEOUT}, x_1, f)$
  parse $\mathsf{inp}_2 := (sid', \mathsf{INPUT\_TIMEOUT}', \mathsf{TRIGGER\_TIMEOUT}', x_2, f')$
  assert that:
    $sid = sid'$, $f = f'$
    $\mathsf{INPUT\_TIMEOUT} = \mathsf{INPUT\_TIMEOUT}'$
    $\mathsf{TRIGGER\_TIMEOUT} = \mathsf{TRIGGER\_TIMEOUT}'$
  assert $r \leq \mathsf{INPUT\_TIMEOUT}$, parse $f = (f_1, f_2, \phi_1, \phi_2)$
  $z_1 = f_1(x_1, x_2, r)$, $\mathsf{ct}_{\mathsf{load}} := \mathcal{E}_2.\mathsf{Enc}(z_1; k)$, return ($\mathsf{ct}_{\mathsf{load}}$, "ok")

On input ("loadfinish", $\sigma_3, t_{\mathsf{load}}, \sigma_4$):
  assert "loadcontinue" has been called and returned ($\cdot$, "ok")
  assert $\mathcal{V}.\mathsf{Verify}(\sigma_3, (sid, eid_1, \mathsf{prog}_{\text{one-sided-SyX},1}[P_1], (\mathsf{ct}_{\mathsf{load}}, \text{"ok"})); \mathsf{vk}_{\mathsf{att}})$
  assert $\mathcal{T}.\mathsf{Verify}(\sigma_4, t_{\mathsf{load}} \| \text{"load"} \| sid \| \mathsf{ct}_{\mathsf{load}} \| \sigma_3)$
  assert $t_{\mathsf{load}} \leq \mathsf{INPUT\_TIMEOUT}$, set $i_1 := 0$, return (z_1, "ok")

On input* ("triggerstart", w_1):
  assert "loadfinish" has been called and returned ($\cdot$, "ok")
  assert $i_1 = 0$ or "triggerfinish" has been called and returned ($\cdot, i_1 - 1$, "ok")
      and "triggerstart" has never previously returned ($\cdot, i_1$, "ok")
  assert $\phi_1(w_1) = 1$ and that $r \leq \mathsf{TRIGGER\_TIMEOUT}$
  $i_1 := i_1 + 1$, $z_{2,i_1} := f_2(x_1, x_2, w_1, r)$, $\mathsf{ct}_{\mathsf{trigger1},i_1} := \mathcal{E}_2.\mathsf{Enc}(z_{2,i_1}; k)$
  return ($\mathsf{ct}_{\mathsf{trigger1},i_1}, i_1$, "ok")

On input* ("triggerfinish", $\sigma_{5,i_1}, t_{\mathsf{trigger1},i_1}, \sigma_{6,i_1}$):
  assert "triggerstart" has been called and returned ($\cdot, i_1$, "ok")
  assert "triggerfinish" has never previously returned ($\cdot, i_1$, "ok")
  assert $\mathcal{V}.\mathsf{Verify}(\sigma_{5,i_1}, (sid, eid_1, \mathsf{prog}_{\mathsf{SyX},1}[P_1], (\mathsf{ct}_{\mathsf{trigger1},i_1}, i_1, \text{"ok"})); \mathsf{vk}_{\mathsf{att}})$
  assert $\mathcal{T}.\mathsf{Verify}(\sigma_{6,i_1}, t_{\mathsf{trigger1},i_1} \| \text{"trigger1"} \| sid \| i_1 \| \mathsf{ct}_{\mathsf{trigger1},i_1} \| \sigma_{5,i_1})$
  assert $t_{\mathsf{trigger1},i_1} \leq \mathsf{TRIGGER\_TIMEOUT}$, return ($z_{2,i_1}, i_1$, "ok")

Figure 13: Program installed in the secure hardware of Party P_1.

$$\mathbf{Prot}_{\textsf{one-sided-SyX},1}[sid, \mathcal{G}_{\textsf{acrs}}.\textsf{mpk}, P_1]$$

$\textsf{inp}_1 := (sid, \textsf{INPUT\_TIMEOUT}, \textsf{TRIGGER\_TIMEOUT}, x_1, f = (f_1, f_2, \phi_1, \phi_2))$

Initialization

$eid_1 := \mathcal{G}_{\textsf{att}}.\textsf{install}(sid, \textsf{prog}_{\textsf{one-sided-SyX},1}[P_1])$
henceforth denote $\mathcal{G}_{\textsf{att}}.\textsf{resume}(\cdot) := \mathcal{G}_{\textsf{att}}.\textsf{resume}(eid_1, \cdot)$
$(\textsf{pk}, \sigma_1) := \mathcal{G}_{\textsf{att}}.\textsf{resume}(\text{“init”})$
send $(eid_1, \psi(P_2, \textsf{pk}, \sigma_1))$ to P_2

Load Phase

await $\textsf{ct}_2$ from P_2
$(\textsf{ct}_2', \sigma_2) := \mathcal{G}_{\textsf{att}}.\textsf{resume}(\text{“loadstart”}, \textsf{ct}_2)$
send $\psi(P_2, \textsf{ct}_2', \sigma_2)$ to P_2, await $\textsf{ct}_{\textsf{continue}}$ from P_2
$(\textsf{ct}_{\textsf{load}}, \text{“ok”}, \sigma_3) := \mathcal{G}_{\textsf{att}}.\textsf{resume}(\text{“loadcontinue”}, \textsf{ct}_{\textsf{continue}}, \textsf{inp}_1)$
$(t_{\textsf{load}}, \sigma_4) := \mathcal{F}_{\textsf{BB}}.\textsf{post}(\text{“load”} \| sid \| \textsf{ct}_{\textsf{load}} \| \sigma_3)$
$(z_1, \text{“ok”}, \_) := \mathcal{G}_{\textsf{att}}.\textsf{resume}(\text{“loadfinish”}, \sigma_3, t_{\textsf{load}}, \sigma_4)$, set $i_1 := 0$

Trigger Phase

Triggers by P_1

$$\left\{\begin{array}{l} i_1 := i_1 + 1 \\ (\textsf{ct}_{\textsf{trigger1},i_1}, i_1, \text{“ok”}, \sigma_{5,i_1}) := \mathcal{G}_{\textsf{att}}.\textsf{resume}(\text{“triggerstart”}, w_1) \\ (t_{\textsf{trigger1},i_1}, \sigma_{6,i_1}) := \mathcal{F}_{\textsf{BB}}.\textsf{post}(\text{“trigger1”} \| sid \| i_1 \| \textsf{ct}_{\textsf{trigger1},i_1} \| \sigma_{5,i_1}) \\ (z_{2,i_1}, i_1, \text{“ok”}, \_) := \mathcal{G}_{\textsf{att}}.\textsf{resume}(\text{“triggerfinish”}, \sigma_{5,i_1}, t_{\textsf{trigger1},i_1}, \sigma_{6,i_1}) \end{array}\right\}$$

Figure 14: Protocol executed by Party P_1 in realizing $\mathcal{F}_{\textsf{SyX}}$ with One-Sided Trigger.

$$\textbf{Prot}_{\textsf{one-sided-SyX},2}[sid, P_2]$$

$\mathsf{inp}_2 := (sid, \mathsf{INPUT\_TIMEOUT}, \mathsf{TRIGGER\_TIMEOUT}, x_2, f = (f_1, f_2, \phi_1, \phi_2))$

Load Phase

await (eid_1, ψ) from P_1
// Henceforth for $\tilde{\psi} := (\mathsf{msg}, C, \pi)$, let $\mathsf{Ver}(\tilde{\psi}) :=$
$\mathsf{Ver}(\mathsf{crs}, (sid, eid_1, C, \mathsf{mpk}, \mathcal{G}_{\mathsf{acrs}}.\mathsf{mpk}, P_2, \mathsf{msg}), \pi)$
assert $\mathsf{Ver}(\psi)$, parse $\psi := (\mathsf{pk}, \_, \_)$
$k \xleftarrow{\$} \{0,1\}^\lambda$, $\mathsf{ct}_2 = \mathcal{E}_1.\mathsf{Enc}((\mathsf{inp}_2, k); \mathsf{pk})$, send ct_2 to P_1
await ψ from P_1, assert $\mathsf{Ver}(\psi)$, parse $\psi := (\mathsf{ct}_2', \_, \_)$
assert $\mathsf{ct}_2 = \mathsf{ct}_2'$
$\mathsf{ct}_{\mathsf{continue}} \xleftarrow{\$} \mathcal{E}_2.\mathsf{Enc}(\text{“continue”})$, send $\mathsf{ct}_{\mathsf{continue}}$ to P_1
obtain a $t_{\mathsf{load}} \leq \mathsf{INPUT\_TIMEOUT}$ such that
$(\text{“load”} \| sid \| \mathsf{ct}_{\mathsf{load}} \| \sigma_3, \sigma_4) := \mathcal{F}_{\mathsf{BB}}.\mathsf{getContent}(t_{\mathsf{load}})$
assert $\mathcal{V}.\mathsf{Verify}(\sigma_3, (sid, eid_1, \mathsf{prog}_{\textsf{one-sided-SyX},1}[P_1], (\mathsf{ct}_{\mathsf{load}}, \text{“ok”})); \mathsf{vk}_{\mathsf{att}})$
assert $\mathcal{T}.\mathsf{Verify}(\sigma_4, t_{\mathsf{load}} \| \text{“load”} \| sid \| \mathsf{ct}_{\mathsf{load}} \| \sigma_3)$
assert $t_{\mathsf{load}} \leq \mathsf{INPUT\_TIMEOUT}$
$z_1 := \mathcal{E}_2.\mathsf{Dec}(\mathsf{ct}_{\mathsf{load}}; k)$, assert that decryption succeeds, set $i_1 := 0$

Trigger Phase

Triggers by P_1

$$\left\{\begin{array}{l}
\text{obtain a } t_{\mathsf{trigger1},i_1+1} \leq \mathsf{TRIGGER_TIMEOUT} \text{ such that} \\
\quad (\text{“trigger1”} \| sid \| (i_1+1) \| \mathsf{ct}_{\mathsf{trigger1},i_1+1} \| \sigma_{5,i_1+1}, \sigma_{6,i_1+1}) := \\
\hfill \mathcal{F}_{\mathsf{BB}}.\mathsf{getContent}(t_{\mathsf{trigger1},i_2+1}) \\
\text{assert } \mathcal{V}.\mathsf{Verify}(\sigma_{5,i_1+1}, (sid, eid_1, \mathsf{prog}_{\textsf{one-sided-SyX},1}[P_1], \\
\hfill (\mathsf{ct}_{\mathsf{trigger1},i_1+1}, i_1+1, \text{“ok”})); \mathsf{vk}_{\mathsf{att}}) \\
\text{assert } \mathcal{T}.\mathsf{Verify}(\sigma_{6,i_1+1}, t_{\mathsf{trigger1},i_1+1} \| \text{“trigger1”} \| sid \| (i_1+1) \| \\
\hfill \mathsf{ct}_{\mathsf{trigger1},i_1+1} \| \sigma_{5,i_1+1}) \\
\text{assert } t_{\mathsf{trigger1},i_1+1} \leq \mathsf{TRIGGER_TIMEOUT} \\
z_{2,i_1+1} := \mathcal{E}_2.\mathsf{Dec}(\mathsf{ct}_{\mathsf{trigger1},i_1+1}; k), \text{ assert that decryption succeeds} \\
i_1 := i_1 + 1
\end{array}\right\}$$

Figure 15: Protocol executed by Party P_2 in realizing $\mathcal{F}_{\mathsf{SyX}}$ with One-Sided Trigger.

parties P_i and P_j have access to the ideal functionality $\mathcal{F}_{\mathsf{SyX}}$. In particular, let $\mathcal{F}_{\mathsf{SyX}}^{i,j}$ represent the instantiation of the $\mathcal{F}_{\mathsf{SyX}}$ functionality used by parties P_i, P_j. Furthermore, only parties P_1 and P_2 may trigger the instances of $\mathcal{F}_{\mathsf{SyX}}$ that they have access to. We wish to perform fair secure function evaluation of some 3-input 3-output functionality F.

Reduction to single output functionalities. Let $(y_1, y_2, y_3) \xleftarrow{\$} F(x_1, x_2, x_3)$ be the output of the function evaluation. We define a new four input single output functionality F' such that

$$F'(x_1, x_2, x_3, z) = F^1(x_1, x_2, x_3) \| F^2(x_1, x_2, x_3) \| F^3(x_1, x_2, x_3) \oplus z = y_1 \| y_2 \| y_3 \oplus z$$

where $z = z_1 \| z_2 \| z_3$ and $|y_i| = |z_i|$ for all $i \in [3]$. The idea is that the party P_i will obtain $z' = F'(x_1, x_2, x_3, z)$ and z_i. Viewing $z' = z'_1 \| z'_2 \| z'_3$ where $|z'_i| = |z_i|$[20] for all $i \in [3]$, party P_i reconstructs its output as

$$y_i = z_i \oplus z'_i$$

Now, we may assume that the input of party P_i is (x_i, z_i) (or we can generate random z_is as part of the computation) which determines z. It thus suffices to consider fair secure function evaluation of single output functionalities.

Reduction to fair reconstruction. We will use ideas similar to [GIM+10,KVV16] where instead of focusing on fair secure evaluation of an arbitrary function, we only focus on fair reconstruction of an additive secret sharing scheme. The main idea is to let the three parties run a secure computation protocol that computes the output of the secure function evaluation on the parties' inputs, and then additively secret shares the output. Given this step, fair secure computation then reduces to fair reconstruction of the underlying additive secret sharing scheme.

The underlying additive secret sharing scheme. We use an additive secret sharing of the output y. Let the shares be y_i for $i \in [3]$. That is, it holds that

$$y = \bigoplus_{i \in [3]} y_i$$

We would like party P_i to reconstruct y by obtaining all shares y_i for each $i \in [3]$. Initially, each party P_i is given y_i. Therefore, each party P_i only needs to obtain y_j and y_k for $j, k \neq i$.

Fair reconstruction via $\mathcal{F}_{\mathsf{SyX}}$. We assume that the secure function evaluation also provides commitments to all the shares of the output. That is, P_i receives $(y_i, \overrightarrow{c})$ for each $i \in [3]$, where Com is a commitment scheme and

$$\overrightarrow{c} = \{\mathsf{Com}(y_1), \mathsf{Com}(y_2), \mathsf{Com}(y_3)\}$$

Furthermore, we assume that each party P_i picks its own verification key vk_i and signing key sk_i with respect to a signature scheme with a signing algorithm Sign and a verification algorithm Verify, for each $i \in [3]$. All parties then broadcast their verification keys to all parties. Let

$$\overrightarrow{\mathsf{vk}} = \{\mathsf{vk}_1, \mathsf{vk}_2, \mathsf{vk}_3\}$$

[20] We may assume without loss of generality that the lengths of the outputs of each party are known beforehand.

Each pair of parties P_i and P_j then initializes $\mathcal{F}_{\mathsf{SyX}}^{i,j}$ with inputs

$$x_i = \left(\overrightarrow{\mathsf{vk}}, \mathsf{sk}_i, y_i, \overrightarrow{c}\right)$$

and

$$x_j = \left(\overrightarrow{\mathsf{vk}}, \mathsf{sk}_j, y_j, \overrightarrow{c}\right)$$

The function f_1 checks if both parties provided the same value for $\overrightarrow{\mathsf{vk}}, \overrightarrow{c}$ and checks the y_i and y_j are valid openings to the corresponding commitments. It also checks that the signing keys provided by the parties are consistent with the corresponding verification keys (more precisely, we will ask for randomness provided to the key generation algorithm of the signature scheme). If all checks pass, then $\mathcal{F}_{\mathsf{SyX}}^{i,j}$ computes

$$\sigma_{i,j} = \mathsf{Sign}((i,j);\mathsf{sk}_i)\|\mathsf{Sign}((i,j);\mathsf{sk}_j)$$

This completes the description of f_1.

Synchronization step. The output of f_1 for each of the $\mathcal{F}_{\mathsf{SyX}}^{i,j}$ will provide a way to synchronize all $\mathcal{F}_{\mathsf{SyX}}$ instances. By synchronization, we mean that an $\mathcal{F}_{\mathsf{SyX}}^{i,j}$ instance cannot be triggered unless every other instance has already completed its input phase successfully. We will have the instance $\mathcal{F}_{\mathsf{SyX}}^{i,j}$ to simply output both y_i and y_j to both parties if triggered successfully – this defines f_2. The real ingenuity of the protocol lies in the design of the predicates ϕ. The protocol proceeds in two rounds:

- *Round 1*: The channel $\mathcal{F}_{\mathsf{SyX}}^{1,2}$ accepts a trigger $(\overrightarrow{\sigma}, y_1)$ from P_1. If P_1 provides this trigger, then both P_1 and P_2 receive $(\overrightarrow{\sigma}, y_1, y_2)$.
- *Round 2*: The channel $\mathcal{F}_{\mathsf{SyX}}^{1,3}$ accepts a trigger $(\overrightarrow{\sigma}, y_1, y_2)$ from P_1, and the channel $\mathcal{F}_{\mathsf{SyX}}^{2,3}$ accepts a trigger $(\overrightarrow{\sigma}, y_1, y_2)$ from P_2. If P_1 provides the trigger, then both P_1 and P_3 receive $(\overrightarrow{\sigma}, y_1, y_2, y_3)$, and if P_2 provides the trigger, then both P_2 and P_3 receive $(\overrightarrow{\sigma}, y_1, y_2, y_3)$.

Protocol intuition. We briefly discuss certain malicious behaviors and how we handle them. From the description above, it is clear that parties have no information about the output until one of the $\mathcal{F}_{\mathsf{SyX}}$ instances is triggered. Furthermore, note that this implies that the corrupt parties must successfully complete the input phases of the instances of $\mathcal{F}_{\mathsf{SyX}}$ that it shares with all of the honest parties in order to obtain the witness that can be used to trigger the $\mathcal{F}_{\mathsf{SyX}}$ instances. Following the input phases of all of the $\mathcal{F}_{\mathsf{SyX}}$ instances, we ask each party to broadcast the receipt $\sigma_{i,j}$ obtained from $\mathcal{F}_{\mathsf{SyX}}^{i,j}$. Now suppose parties P_i and P_j are both dishonest, and suppose they do not broadcast $\sigma_{i,j}$. Note also that since P_i and P_j collude, they do not need the help of $\mathcal{F}_{\mathsf{SyX}}$ to compute $\sigma_{i,j}$. Since honest P_k does not know the synchronizing witness $\overrightarrow{\sigma}$, it will not be able to trigger any of the $\mathcal{F}_{\mathsf{SyX}}$ instances. However, note that for the adversary to learn the output of the computation, the corrupt party P_i (without loss of generality) will need to trigger $\mathcal{F}_{\mathsf{SyX}}^{i,k}$ to obtain P_k's share of the key. However, once P_i triggers $\mathcal{F}_{\mathsf{SyX}}^{i,k}$, it follows that P_k would obtain the synchronizing witness $\overrightarrow{\sigma}$ along with some shares of the output. If $k = 1$ or $k = 2$, then P_k obtains $(\overrightarrow{\sigma}, y_1, y_2)$ at the end of round 1 and can successfully trigger $\mathcal{F}_{\mathsf{SyX}}^{1,3}$ or $\mathcal{F}_{\mathsf{SyX}}^{2,3}$ in round 2 to obtain the final share y_3 of the output. If $k = 3$, then P_3 obtains $(\overrightarrow{\sigma}, y_1, y_2, y_3)$ at the end of round 2, thus obtaining all shares of the output. In this way, all live (parties that remain online) parties obtain the output at the end of the protocol even if one of the parties do.

6.2 Extending to arbitrary n

Let $P_1, \ldots, P_n$ be the n parties with inputs $x_1, \ldots, x_n$ respectively. For $i, j \in [n]$ with $i < j$, we have that parties P_i and P_j have access to the ideal functionality $\mathcal{F}_{\mathsf{SyX}}$. In particular, let $\mathcal{F}^{i,j}_{\mathsf{SyX}}$ represent the instantiation of the $\mathcal{F}_{\mathsf{SyX}}$ functionality used by parties P_i, P_j. Furthermore, only parties $P_1, \ldots, P_t$ may trigger the instances of $\mathcal{F}_{\mathsf{SyX}}$ that they have access to. Most of the discussion in the three-party case extends naturally to the n-party setting. The portion that differs is essentially the synchronization step.

Synchronization step. The output of f_1 for each of the $\mathcal{F}^{i,j}_{\mathsf{SyX}}$ will provide a way to synchronize all $\mathcal{F}_{\mathsf{SyX}}$ instances. By synchronization, we mean that an $\mathcal{F}^{i,j}_{\mathsf{SyX}}$ instance cannot be triggered unless every other instance has already completed its input phase successfully. We will have the instance $\mathcal{F}^{i,j}_{\mathsf{SyX}}$ to simply output both y_i and y_j to both parties if triggered successfully – this defines f_2. The real ingenuity of the protocol lies in the design of the predicates ϕ. We attempt to generalize the three-party protocol as follows. The protocol proceeds in $n-1$ rounds. For each $r \in [n-1]$:

- *Round* r: For every $i \in [\min\{t, r\}]$, the channel $\mathcal{F}^{i,r+1}_{\mathsf{SyX}}$ accepts a trigger $(\overrightarrow{\sigma}, y_1, \ldots, y_r)$ from P_i. If P_1 provides this trigger, then both P_i and P_{r+1} receive $(\overrightarrow{\sigma}, y_1, \ldots, y_{r+1})$.

Consider, however, the case of $n = 4$ and $t = 2$. Since P_3 does not have the ability to trigger an instance of $\mathcal{F}_{\mathsf{SyX}}$, it cannot obtain y_4. We thus add the following step, round n, of the protocol:

- *Round* n: For every $i \in [n]$, if P_i has received all shares $y_1, \ldots, y_n$, it broadcasts the shares.

Protocol intuition. From the description above, it is clear that parties have no information about the output until one of the $\mathcal{F}_{\mathsf{SyX}}$ instances is triggered. Furthermore, note that this implies that the corrupt parties must successfully complete the input phases of the instances of $\mathcal{F}_{\mathsf{SyX}}$ that it shares with all of the honest parties in order to obtain the witness that can be used to trigger the $\mathcal{F}_{\mathsf{SyX}}$ instances. Following the input phases of all of the $\mathcal{F}_{\mathsf{SyX}}$ instances, we ask each party to broadcast the receipt $\sigma_{i,j}$ obtained from $\mathcal{F}^{i,j}_{\mathsf{SyX}}$. Suppose the adversary obtains the output at the end of the computation. This implies that there exists a party P_i for some $i \in [n]$ that receives all shares $y_1, \ldots, y_n$ at the end of the protocol. Let P_j be an honest party, for $j \in [n]$. We wish to argue that P_j must have also received all shares of the output at the end of the protocol. Firstly, if P_i were honest, then P_i would broadcast its output and hence P_j would obtain it as well. We now turn our attention to the case that P_i is corrupt. We now wish to argue the existence of an honest party P_k for some $k \in [n]$ that also obtains all the shares of the output at the end of the protocol. Notice that as before this implies that P_j must have also received all shares of the output at the end of the protocol. Consider party P_j. Suppose that in round $j-1$, no party triggered an instance of $\mathcal{F}_{\mathsf{SyX}}$ that involved P_j. We claim that this cannot be possible, as this would imply that no party other than P_j learns y_j at the end of the protocol. We argue this as follows. Rounds 1 through $j-2$ do not involve y_j at all and hence no party other than P_j learns y_j at the end of round $j-2$. Suppose that in round $j-1$, no party triggered an instance of $\mathcal{F}_{\mathsf{SyX}}$ that involved P_j. Then, no party other than P_j learns y_j at the end of round $j-1$. Furthermore, party P_j does not learn $y_1, \ldots, y_{j-1}$ at the end of round $j-1$. In round j, no party, including P_j has a witness that it can use to trigger any instance of $\mathcal{F}_{\mathsf{SyX}}$. It is thus easy to see that in all rounds $r \geq j$, no party has a witness that it can use to trigger any instance of $\mathcal{F}_{\mathsf{SyX}}$. This in particular implies that no party other than P_j learns y_j at the end of the protocol. Since P_i learns y_j at the end of the protocol, it must be the case that there exists a party P_ℓ for some $\ell \in [\min\{t, j-1\}]$ that triggered the channel $\mathcal{F}_{\mathsf{SyX}}^{\ell,j}$ that involved P_j. Then, at the end of round $j-1$, party P_j learns $(\overrightarrow{\sigma}, y_1, \ldots, y_j)$. If $j \le t$, then it is easy to see that P_j will be able to learn all the shares of the output at the end of the protocol as it will be able to successfully trigger the channel $\mathcal{F}_{\mathsf{SyX}}^{j,m}$ in round $m-1$ for every $j+1 \le m \le n$. In this case, $k = j$ and we are done. Notice that this means that if there exists an honest party P_j with $j \le t$, then P_j (and hence, all honest parties) receive the output at the end of the protocol. Suppose $j > t$, and in particular, $P_1, \ldots, P_t$ are corrupt. In this case, we let $k = n$. Suppose that in round $n-1$, no corrupt party triggered an instance of $\mathcal{F}_{\mathsf{SyX}}$ that involved P_n. This, by the definition, means that no party triggered an instance of $\mathcal{F}_{\mathsf{SyX}}$ that involved P_n. We claim that this cannot be possible, as this would imply that no party other than P_n learns y_n at the end of the protocol. We argue this as follows. Rounds 1 through $n-2$ do not involve y_n at all and hence no party other than P_n learns y_n at the end of round $n-2$. Suppose that in round $n-1$, no party triggered an instance of $\mathcal{F}_{\mathsf{SyX}}$ that involved P_n. Then, no party other than P_n learns y_n at the end of round $n-1$. Furthermore, party P_n does not learn $y_1, \ldots, y_{n-1}$ at the end of round $n-1$. In round n, no party, including P_n has obtained all shares of the output and hence no party broadcasts anything. This in particular implies that no party other than P_n learns y_n at the end of the protocol. Since P_i learns y_n at the end of the protocol, it must be the case that there exists a (corrupt) party P_ℓ for some $\ell \in [t]$ that triggered the channel $\mathcal{F}_{\mathsf{SyX}}^{\ell,n}$ that involved P_n. Then, at the end of round $n-1$, party P_n learns $(\overrightarrow{\sigma}, y_1, \ldots, y_n)$. Since P_n has obtained all shares at the end of round $n-1$, it broadcasts them in round n and thus every honest party learns all shares of the output at the end of the protocol. This completes the argument.

6.3 Protocol

We now present the protocol for fair secure computation in the $(\mathcal{F}_{\mathsf{bc}}, \mathcal{F}_{\mathsf{MPC}}, \mathcal{F}_{\mathsf{SyX}})$-hybrid model.

Preliminaries. F is the n-input n-output functionality to be computed; x_i is the input of party P_i for $i \in [n]$; $\mathcal{F}_{\mathsf{SyX}}^{a,b}$ represents the instantiation of the $\mathcal{F}_{\mathsf{SyX}}$ functionality used by parties P_a, P_b with time out round numbers $\mathsf{INPUT\_TIMEOUT} = 0$ and $\mathsf{TRIGGER\_TIMEOUT} = n-1$ for $a < b$, where $a, b \in [n]$; $\left(\mathsf{Com}, \mathsf{Open}, \widetilde{\mathsf{Com}}, \widetilde{\mathsf{Open}}\right)$ is an honest-binding commitment scheme; $\mathcal{V} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ is a signature scheme; r denotes the current round number.

Protocol. The protocol Π_{FMPC} proceeds as follows:

- Define F' to the be the following n-input n-output functionality: On input $\overrightarrow{x} = (x_1, \ldots, x_n)$:
  - Let $(y_1, \ldots, y_n) = F(x_1, \ldots, x_n)$ and let

$$y = y_1 \| \ldots \| y_n$$

    Sample random strings $\alpha_i \xleftarrow{\$} \{0,1\}^*$ such that $|\alpha_i| = |y_i|$ for each $i \in [n]$. Let

$$\alpha = \alpha_1 \| \ldots \| \alpha_n$$

    Let $z = y \oplus \alpha$.

– Sample a random additive n-out-of-n secret sharing $z_1, \ldots, z_n$ of z such that

$$z = \bigoplus_{i \in [n]} z_i$$

– Compute commitments along with their openings $(c_i^z, \omega_i^z) \xleftarrow{\$} \mathsf{Com}(z_i)$ to each of the shares z_i for each $i \in [n]$. Let

$$\overrightarrow{c^z} = (c_1^z, \ldots, c_n^z)$$

– Party P_i receives output $\left(\alpha_i, \overrightarrow{c^z}, \omega_i^z, z_i\right)$ for each $i \in [n]$.

- The parties invoke the ideal functionality $\mathcal{F}_{\mathsf{MPC}}$ with inputs $((x_1, F'), \ldots, (x_n, F'))$. If the ideal functionality returns $\perp$ to party P_i, then P_i aborts for any $i \in [n]$[21]. Otherwise, party P_i receives output $\left(\alpha_i, \overrightarrow{c^z}, \omega_i^z, z_i\right)$ for each $i \in [n]$.

- Each party P_i, for each $i \in [n]$, picks a random $\beta_i \in \{0,1\}^*$ and uses this randomness to pick a signing and verification key pair $(\mathsf{sk}_i, \mathsf{vk}_i) = \mathcal{V}.\mathsf{Gen}(1^\lambda; \beta_i)$. It then invokes the ideal functionality $\mathcal{F}_{\mathsf{bc}}$ and broadcasts vk_i to all other parties. If it does not receive vk_j for all $j \neq i$, it aborts. Otherwise, it obtains

$$\overrightarrow{\mathsf{vk}} = (\mathsf{vk}_1, \ldots, \mathsf{vk}_n)$$

- For each $a, b \in [n]$ with $a < b$, define the following functions.

  – Let $f_1^{a,b}$ be the function that takes as input (γ, γ') and parses

$$\gamma = \left(\overrightarrow{\mathsf{vk}}, \mathsf{sk}, \beta, \overrightarrow{c^z}, \omega^z, z\right)$$

  and

$$\gamma' = \left(\overrightarrow{\mathsf{vk}}', \mathsf{sk}', \beta', \overrightarrow{c^z}', \omega^{z\prime}, z'\right)$$

  It checks that:
  * $\overrightarrow{\mathsf{vk}} = \overrightarrow{\mathsf{vk}}'$, $\overrightarrow{c^z} = \overrightarrow{c^z}'$
  * $(\mathsf{sk}, \mathsf{vk}_a) = \mathcal{V}.\mathsf{Gen}(1^\lambda; \beta)$, $(\mathsf{sk}', \mathsf{vk}_b) = \mathcal{V}.\mathsf{Gen}(1^\lambda; \beta')$
  * $\mathsf{Open}(c_a^z, \omega^z, z) = \mathsf{Open}(c_b^z, \omega^{z\prime}, z') = 1$

  If all of these checks pass, then $f_1^{a,b}$ outputs

$$\sigma_{a,b} = (\mathcal{V}.\mathsf{Sign}((a,b); \mathsf{sk}_a), \mathcal{V}.\mathsf{Sign}((a,b); \mathsf{sk}_b))$$

  and otherwise it outputs $\perp$.

[21] In the $\mathcal{F}_{\mathsf{OT}}$-hybrid model, let $\pi_{F'}$ denote the protocol for the functionality F' defined in Lemma 8. The parties execute $\pi_{F'}$. If the execution of $\pi_{F'}$ aborts, we are assuming that all (honest) parties are aware of the round when the execution of $\pi_{F'}$ aborts, that is, when the adversary has decided to abort the execution of $\pi_{F'}$. Since we are working in the $\mathcal{F}_{\mathsf{MPC}}$-hybrid model, we know that in the ideal model, this is the case when the honest parties receive $\perp$ as their output. If we assume that in the case when the the adversary decides to let the honest parties obtain their outputs, no honest party ever receives $\perp$, this could be used to identify the scenario when the adversary has decided to abort the execution of $\pi_{F'}$. Thus, we could, in principle, replace this instruction with: If party P_i receives $\perp$ as it's output, it aborts. Furthermore, since we are considering the case of *unanimous abort*, if the adversary has decided to abort the execution of $\pi_{F'}$, all honest parties abort the protocol.

– Let $\phi_1^{a,b}$ be the function that takes as input a witness w, which is of the form $\left(\overrightarrow{\sigma}, \overrightarrow{z}, \overrightarrow{\omega^z}\right)$.

  * If $a > t$ or $b \neq r+1$, it outputs 0.
  * If $a \leq t$ and $b = r+1$, it checks that:
    · For all $a, b \in [n]$ with $a < b$,

$$\mathcal{V}.\mathsf{Verify}\left(\sigma_{a,b,1}, (a,b); \mathsf{vk}_a\right) = 1$$

    and

$$\mathcal{V}.\mathsf{Verify}\left(\sigma_{a,b,2}, (a,b); \mathsf{vk}_b\right) = 1$$

    · $|\overrightarrow{z}| = \left|\overrightarrow{\omega^z}\right| = r$
    · $\mathsf{Open}\left(c_j^z, \omega_j^z, z_j\right) = 1$ for every $j \in [r]$.

– Let $\phi_2^{a,b}$ be the function that takes as input a witness w, which is of the form $\left(\overrightarrow{\sigma}, \overrightarrow{z}, \overrightarrow{\omega^z}\right)$.

  * If $b > t$ or $a \neq r+1$, it outputs 0.
  * If $b \leq t$ and $a = r+1$, it checks that:
    · For all $a, b \in [n]$ with $a < b$,

$$\mathcal{V}.\mathsf{Verify}\left(\sigma_{a,b,1}, (a,b); \mathsf{vk}_a\right) = 1$$

    and

$$\mathcal{V}.\mathsf{Verify}\left(\sigma_{a,b,2}, (a,b); \mathsf{vk}_b\right) = 1$$

    · $|\overrightarrow{z}| = \left|\overrightarrow{\omega^z}\right| = r$
    · $\mathsf{Open}\left(c_j^z, \omega_j^z, z_j\right) = 1$ for every $j \in [r]$.

– Let $f_2^{a,b}$ be the function that takes as input (γ, γ') where γ, γ' are as above, and outputs $(\omega^z, z, \omega^{z\prime}, z')$.

- Set $r = 0$[22]. Each party P_a for each $a \in [n]$ will now run the input phase to set up each instance of $\mathcal{F}_{\mathsf{SyX}}$ that it is involved in. For each pair of parties P_a, P_b with $a \neq b$ for $a, b \in [n]$, let $a' = \mathsf{min}(a,b)$ and $b' = \mathsf{max}(a,b)$. For each such pair of parties P_a, P_b, party P_a runs the input phase of $\mathcal{F}_{\mathsf{SyX}}^{a',b'}$, providing inputs (x_a, f), where

$$x_a = \left(\overrightarrow{\mathsf{vk}}, \mathsf{sk}_a, \beta_a, \overrightarrow{c^z}, \omega_a^z, z_a\right)$$

  and

$$f = \left(f_1^{a',b'}, f_2^{a',b'}, \phi_1^{a',b'}, \phi_2^{a',b'}\right)$$

- If $r > n$, abort. Otherwise, while $r \leq n$,

[22] This does not entail actually setting $r = 0$, but rather viewing the current round as round zero and henceforth referencing rounds with respect to it, that is, viewing r as the round number relative to the round number when this statement was executed.

- If a party P_a for $a \in [n]$ receives $\sigma_{a',b'}$ from each $\mathcal{F}_{\mathsf{SyX}}^{a',b'}$ it is involved in, indicating that the input phase of all such $\mathcal{F}_{\mathsf{SyX}}$ functionalities were completed successfully, and $r = 0$, it invokes the ideal functionality $\mathcal{F}_{\mathsf{bc}}$ and broadcasts

$$\overrightarrow{\sigma_a} = \left\{\sigma_{a',b'}\right\}_{a'=a \,\vee\, b'=a}$$

  to all the parties. Otherwise, it invokes the ideal functionality $\mathcal{F}_{\mathsf{bc}}$ when $r = 1$ and broadcasts abort to all the parties and aborts.

- For $i \in [\min\{t, r\}]$ and $r < n$, by the end of round $r - 1$, if P_i has received a witness w, which is of the form $\left(\overrightarrow{\sigma}, \overrightarrow{z}, \overrightarrow{\omega^z}\right)$ such that
  * For all $a, b \in [n]$ with $a < b$,

$$\mathcal{V}.\mathsf{Verify}\left(\sigma_{a,b,1}, (a, b); \mathsf{vk}_a\right) = 1$$

    and

$$\mathcal{V}.\mathsf{Verify}\left(\sigma_{a,b,2}, (a, b); \mathsf{vk}_b\right) = 1$$

  * $|\overrightarrow{z}| = \left|\overrightarrow{\omega^z}\right| = r$
  * $\mathsf{Open}\left(c_j^z, \omega_j^z, z_j\right) = 1$ for every $j \in [r]$.

  then, P_i uses w to invoke the trigger phase of the channel $\mathcal{F}_{\mathsf{SyX}}^{i,r+1}$.

- For $i \in [n]$, if at the end of round $r = n - 1$, party P_i has obtained values $\left(\overrightarrow{z}, \overrightarrow{\omega^z}\right)$ such that
  * $|\overrightarrow{z}| = \left|\overrightarrow{\omega^z}\right| = n$
  * $\mathsf{Open}\left(c_j^z, \omega_j^z, z_j\right) = 1$ for every $j \in [n]$.

  then, P_i invokes the ideal functionality $\mathcal{F}_{\mathsf{bc}}$ in round $r + 1 = n$ and broadcasts $\left(\overrightarrow{z}, \overrightarrow{\omega^z}\right)$ to all the parties.

- For $i \in [n]$, if at the end of round $r = n$, party P_i has obtained values $\left(\overrightarrow{z}, \overrightarrow{\omega^z}\right)$ such that
  * $|\overrightarrow{z}| = \left|\overrightarrow{\omega^z}\right| = n$
  * $\mathsf{Open}\left(c_j^z, \omega_j^z, z_j\right) = 1$ for every $j \in [n]$.

  then, P_i uses the shares $z_1, \ldots, z_n$ to reconstruct z, parses z as $z_1 \| \ldots \| z_n$ where $|z_i| = |y_i|$ for all $i \in [n]$ and computes $y_i = z_i \oplus \alpha_i$ to obtain the output of the computation.

Remark. It is possible to replace the $\mathcal{O}(n^2)$ signatures with n other commitments to n other independent random proof values that can be used to prove that all the instances of $\mathcal{F}_{\mathsf{SyX}}$ completed their input phases successfully.

6.4 Correctness

We sketch the proof of correctness of the above protocol. The correctness of the computation of the functionality F' follows by definition from the correctness of the ideal functionality $\mathcal{F}_{\mathsf{MPC}}$. Furthermore, we have that at the end of the invocation of the ideal functionality $\mathcal{F}_{\mathsf{MPC}}$, either all honest parties *unanimously abort* or all honest parties *unanimously continue*. Thus, assuming that $\mathcal{F}_{\mathsf{MPC}}$ did not abort, every party receives the output of F'. For every $i \in [n]$, let $\overrightarrow{\mathsf{vk}_i}$ denote the set of verification keys that were obtained by party P_i. Note that, by the correctness of the ideal functionality $\mathcal{F}_{\mathsf{bc}}$,

$$\overrightarrow{\mathsf{vk}} = \overrightarrow{\mathsf{vk}_i}$$

for all $i \in [n]$. If $\overrightarrow{\mathsf{vk}}$ does not contain vk_j for every $i \in [n]$, which would happen in the case that some corrupt parties do not broadcast their verification keys, all honest parties *unanimously abort*. Otherwise, all honest parties *unanimously continue*. Assuming the honest parties have not aborted, we note that if the corrupt parties do not provide valid inputs to the input phase of even one of the instances of $\mathcal{F}_{\mathsf{SyX}}$ that they are involved in along with an honest party, say P_i for some $i \in [n]$, by the correctness of the ideal functionality $\mathcal{F}_{\mathsf{SyX}}$ and the binding property for the honestly generated commitments, that particular instance of $\mathcal{F}_{\mathsf{SyX}}$ will not complete its input phase successfully. In this case P_i will force all honest parties to *unanimously abort*, since no party (not even the corrupt ones) can obtain their output. We thus consider the case where all instances of $\mathcal{F}_{\mathsf{SyX}}$ have completed their input phases successfully. Let $i \in [n]$ be the smallest value such that P_i is honest. Suppose $i \leq t$. If a corrupt party triggers any instance of $\mathcal{F}_{\mathsf{SyX}}$ involving P_i with a valid witness in round $i-1$, then the honest party obtains a valid witness to trigger all the instances of $\mathcal{F}_{\mathsf{SyX}}$ that it is involved in in rounds i through $n-1$. In this case, all honest parties obtain the output of the computation at the end of round n. Suppose $i > t$. In this case, $i = t+1$ and $P_1, \ldots, P_t$ are corrupt. Suppose that in round $n-1$, no corrupt party triggered an instance of $\mathcal{F}_{\mathsf{SyX}}$ that involved P_n. This, by the definition, means that no party triggered an instance of $\mathcal{F}_{\mathsf{SyX}}$ that involved P_n. This would imply that no party other than P_n learns y_n at the end of the protocol. We argue this as follows. Rounds 1 through $n-2$ do not involve y_n at all and hence no party other than P_n learns y_n at the end of round $n-2$. Suppose that in round $n-1$, no party triggered an instance of $\mathcal{F}_{\mathsf{SyX}}$ that involved P_n. Then, no party other than P_n learns y_n at the end of round $n-1$. Furthermore, party P_n does not learn $y_1, \ldots, y_{n-1}$ at the end of round $n-1$. In round n, no party, including P_n has obtained all shares of the output and hence no party broadcasts anything. This in particular implies that no party other than P_n learns y_n at the end of the protocol. That is, the adversary does not learn the output of the computation at the end of the protocol, and neither do any of the honest parties. If a corrupt party triggers any instance of $\mathcal{F}_{\mathsf{SyX}}$ involving P_n with a valid witness in round $n-1$, then P_n obtains all shares of the output which it broadcasts to all parties in round n. In this case, all honest parties obtain the output of the computation at the end of round n. This completes the proof of correctness.

6.5 Security

We now prove the following lemma.

Lemma 11. *If* $\left(\mathsf{Com}, \mathsf{Open}, \widetilde{\mathsf{Com}}, \widetilde{\mathsf{Open}}\right)$ *is an honest-binding commitment scheme and* $\mathcal{V}$ *is a signature scheme, then the protocol* Π_{FMPC} *securely computes* $\mathcal{F}_{\mathsf{MPC}}$ *with fairness in the* $(\mathcal{F}_{\mathsf{bc}}, \mathcal{F}_{\mathsf{MPC}}, \mathcal{F}_{\mathsf{SyX}})$*-hybrid model.*

Proof. Let $\mathcal{A}$ be an adversary attaching the execution of the protocol described in Section 6.3 in the $(\mathcal{F}_{\mathsf{bc}}, \mathcal{F}_{\mathsf{MPC}}, \mathcal{F}_{\mathsf{SyX}})$-hybrid model. We construct an ideal-model adversary $\mathcal{S}$ in the ideal model of type fair. Let F be the n-input n-output functionality to be computed. Let $\mathcal{I}$ be the set of corrupted parties. If $\mathcal{I}$ is empty, then there is nothing to simulate. $\mathcal{S}$ begins by simulating the first step of the protocol, namely, the invocation of the ideal functionality $\mathcal{F}_{\mathsf{MPC}}$. Here, $\mathcal{S}$ behaves as the ideal functionality $\mathcal{F}_{\mathsf{MPC}}$. Recall that the type of $\mathcal{F}_{\mathsf{MPC}}$ is abort. $\mathcal{S}$ obtains the inputs $\{(x_i, f_i)\}_{i\in\mathcal{I}}$ of the corrupted parties from $\mathcal{A}$. If $(x_i, f_i) = \mathsf{abort}$ for any $i \in \mathcal{I}$, $\mathcal{S}$ forwards $\{(x_i, f_i)\}_{i\in\mathcal{I}}$ to the trusted party computing $\mathcal{F}_{\mathsf{MPC}}$ with fairness, receives $\perp$ as the output of all parties, which it forwards $\mathcal{A}$. Suppose $(x_i, f_i) \neq \mathsf{abort}$ for all $i \in \mathcal{I}$. If there exists a $j \in \mathcal{I}$ such that $f_j \neq F'$ as defined in protocol Π_{FMPC}, $\mathcal{S}$ forwards $\{(x_i, f_i)\}_{i\in\mathcal{I}}$ to the trusted party computing $\mathcal{F}_{\mathsf{MPC}}$ with fairness, which aborts, and then aborts itself. If there exists a $j \in \mathcal{I}$ such that (x_j, f_j) is not of the specified format, $\mathcal{S}$ replaces (x_j, f_j) with a default value. Going forward, we assume that for all $i \in \mathcal{I}$, (x_i, f_i) is well-formed and that $f_i = F'$ as defined in Π_{FMPC}.

$\mathcal{S}$ now needs to simulate the outputs received by the corrupted parties from the ideal functionality $\mathcal{F}_{\mathsf{MPC}}$. For each $i \in [n]$, $\mathcal{S}$ samples a random string $\alpha_i \xleftarrow{\$} \{0,1\}^*$ of length equal to the length of the ith output of F. Let

$$\alpha = \alpha_1 \| \ldots \| \alpha_n$$

Let $h \in [n]$ denote the maximum value such that P_h is honest. We note that if $\mathcal{I} = [t]$, then $h = n$. For each $i \in [n] \setminus \{h\}$, $\mathcal{S}$ samples a random string $z_i \xleftarrow{\$} \{0,1\}^*$ of length equal to the sum of the lengths of all the outputs of F. It then computes commitments along with their openings $(c_i^z, \omega_i^z) \xleftarrow{\$} \mathsf{Com}(z_i)$ to each of the shares z_i for each $i \in \mathcal{I}$. For $i = h$, it samples a equivocable commitment $(c_i^z, \mathsf{state}_i) \xleftarrow{\$} \widetilde{\mathsf{Com}}(1^\lambda)$. Let

$$\overrightarrow{c^z} = (c_1^z, \ldots, c_n^z)$$

Thus, the simulator constructs the output $\left(\alpha_i, \overrightarrow{c^z}, \omega_i^z, z_i\right)$ for each $i \in \mathcal{I}$ and forwards it to $\mathcal{A}$. If $\mathcal{A}$ then sends abort, $\mathcal{S}$ forwards $\{(x_i, f_i)\}_{i\in\mathcal{I}}$ to the trusted party computing $\mathcal{F}_{\mathsf{MPC}}$ with fairness, with (x_j, f_j) replaced with abort for some $j \in \mathcal{I}$, receives $\perp$ as the output of all parties, which it forwards $\mathcal{A}$. Otherwise, $\mathcal{A}$ responds with continue. At this point, $\mathcal{S}$ has completed simulating the invocation of the ideal functionality $\mathcal{F}_{\mathsf{MPC}}$.

For each $i \in [n] \setminus \mathcal{I}$, $\mathcal{S}$ picks a random $\beta_i \in \{0,1\}^*$ and uses this randomness to pick a signing and verification key pair $(\mathsf{sk}_i, \mathsf{vk}_i) = \mathcal{V}.\mathsf{Gen}(1^\lambda; \beta_i)$. Now, $\mathcal{S}$ must simulate the invocations of the ideal functionality $\mathcal{F}_{\mathsf{bc}}$ by the corrupt parties. Here, $\mathcal{S}$ behaves as the ideal functionality $\mathcal{F}_{\mathsf{bc}}$. Recall that the type of $\mathcal{F}_{\mathsf{bc}}$ is g.d.. For all $i \in [n] \setminus \mathcal{I}$, $\mathcal{S}$ "broadcasts" vk_i to all the corrupt parties. For any $i \in \mathcal{I}$, if $\mathcal{A}$ instructs P_i to invoke $\mathcal{F}_{\mathsf{bc}}$ with input vk_i, $\mathcal{S}$ "broadcasts" vk_i to all the corrupt parties and stores vk_i. At the end of this round, if $\mathcal{A}$ did not instruct some corrupt party to invoke $\mathcal{F}_{\mathsf{bc}}$, $\mathcal{S}$ forwards $\{(x_i, f_i)\}_{i\in\mathcal{I}}$ to the trusted party computing $\mathcal{F}_{\mathsf{MPC}}$ with fairness, with (x_j, f_j) replaced with abort for some $j \in \mathcal{I}$, receives $\perp$ as the output of all parties, and aborts itself. Otherwise, $\mathcal{S}$ successfully constructs

$$\overrightarrow{\mathsf{vk}} = (\mathsf{vk}_1, \ldots, \mathsf{vk}_n)$$

At this point, $\mathcal{S}$ has completed simulating the invocations of the ideal functionality $\mathcal{F}_{\mathsf{bc}}$ used to broadcast the verification keys of all the parties.

$\mathcal{S}$ maintains a virtual round counter and initializes it to zero. Now, $\mathcal{S}$ has to simulate the invocations of the inputs phases of the instances of the ideal functionality $\mathcal{F}_{\mathsf{SyX}}$ that involve corrupt parties. Here, $\mathcal{S}$ behaves as the ideal functionality $\mathcal{F}_{\mathsf{SyX}}$. Recall that the type of $\mathcal{F}_{\mathsf{SyX}}$ is g.d.. For any $a, b \in [n]$ with $a < b$ and $a \in \mathcal{I}$ and $b \in [n] \setminus \mathcal{I}$, if $\mathcal{A}$ instructs P_a to invoke the input phase of $\mathcal{F}_{\mathsf{SyX}}^{a,b}$ with inputs

$$\gamma = \left(\overrightarrow{\mathsf{vk}}', \mathsf{sk}_a, \beta_a, \overrightarrow{c^z}', \omega^z, z\right)$$

$\mathcal{S}$ computes $f_1^{a,b}(\gamma, \gamma')$ as defined in Π_{FMPC}, where

$$\gamma' = \left(\overrightarrow{\mathsf{vk}}, \mathsf{sk}_b, \beta_b, \overrightarrow{c^z}, \omega_b^z, z_b\right)$$

Note that since $b \in [n] \setminus \mathcal{I}$, $\mathcal{S}$ does in fact have sk_b, β_b. The only values it does not have are ω_h^z, z_h. In the execution of $f_1^{a,b}$, ω_b^z, z_b are needed to check that

$$\mathsf{Open}(c_b^z, \omega_b^z, z_b) = 1$$

Note that since P_b is an honest party, it would always supply inputs such that this check passes. Furthermore, the outcome of this check does not depend on any input that the adversary sends. Thus, in simulating the computation of $f_1^{a,b}$, $\mathcal{S}$ performs all the checks that $f_1^{a,b}$, except this one. If all the checks pass, $\mathcal{S}$ computes

$$\sigma_{a,b} = (\mathcal{V}.\mathsf{Sign}((a,b); \mathsf{sk}_a), \mathcal{V}.\mathsf{Sign}((a,b); \mathsf{sk}_b))$$

and forwards $\sigma_{a,b}$ to the adversary. $\mathcal{S}$ also stores sk_a, β_a. If any of the checks do not pass, $\mathcal{S}$ simply aborts simulating the input phase of this particular instance $\mathcal{F}_{\mathsf{SyX}}^{a,b}$. $\mathcal{S}$ behaves symmetrically if for any $a, b \in [n]$ with $a < b$ and $b \in \mathcal{I}$ and $a \in [n] \setminus \mathcal{I}$, if $\mathcal{A}$ instructs P_b to invoke the input phase of $\mathcal{F}_{\mathsf{SyX}}^{a,b}$. The final case to consider is if for any $a, b \in [n]$ with $a < b$ and $a, b \in \mathcal{I}$, if $\mathcal{A}$ instructs P_a, P_b to invoke the input phase of $\mathcal{F}_{\mathsf{SyX}}^{a,b}$ with inputs

$$\gamma = \left(\overrightarrow{\mathsf{vk}}', \mathsf{sk}_a, \beta_a, \overrightarrow{c^z}', \omega^z, z\right)$$

and

$$\gamma' = \left(\overrightarrow{\mathsf{vk}}'', \mathsf{sk}_b, \beta_b, \overrightarrow{c^z}'', \omega^{z\prime}, z'\right)$$

$\mathcal{S}$ computes $f_1^{a,b}(\gamma, \gamma')$ as defined in Π_{FMPC}. If all the checks pass, $\mathcal{S}$ computes

$$\sigma_{a,b} = (\mathcal{V}.\mathsf{Sign}((a,b); \mathsf{sk}_a), \mathcal{V}.\mathsf{Sign}((a,b); \mathsf{sk}_b))$$

and forwards $\sigma_{a,b}$ to the adversary. $\mathcal{S}$ also stores $\mathsf{sk}_a, \beta_a, \mathsf{sk}_b, \beta_b$. If any of the checks do not pass, $\mathcal{S}$ simply aborts simulating the input phase of this particular instance $\mathcal{F}_{\mathsf{SyX}}^{a,b}$. At the end of this round, let $\mathsf{LoadFailed}$ denote the set of all i such that P_i is an honest party and $\mathcal{A}$ did not instruct some corrupt party to invoke the input phase of an instance of $\mathcal{F}_{\mathsf{SyX}}$ that it was involved in with P_i. If $\mathsf{LoadFailed}$ is not empty, for each $i \in \mathsf{LoadFailed}$, $\mathcal{S}$ must simulate the invocations of the ideal functionality $\mathcal{F}_{\mathsf{bc}}$ by party P_i to broadcast abort. For each $i \in \mathsf{LoadFailed}$, $\mathcal{S}$ "broadcasts" abort to all the corrupt parties. $\mathcal{S}$ then forwards $\{(x_i, f_i)\}_{i \in \mathcal{I}}$ to the trusted party computing $\mathcal{F}_{\mathsf{MPC}}$ with fairness, with (x_j, f_j) replaced with abort for some $j \in \mathcal{I}$, receives $\perp$ as the output of all parties, and aborts itself. Otherwise, $\mathcal{S}$ successfully constructs

$$\overrightarrow{\mathsf{sk}} = (\mathsf{sk}_1, \ldots, \mathsf{sk}_n)$$

and

$$\overrightarrow{\beta} = (\beta_1, \ldots, \beta_n)$$

$\mathcal{S}$ computes

$$\sigma_{a,b} = (\mathcal{V}.\mathsf{Sign}((a,b); \mathsf{sk}_a), \mathcal{V}.\mathsf{Sign}((a,b); \mathsf{sk}_b))$$

for every $a < b \in [n]$ and defines

$$\overrightarrow{\sigma} = \{\sigma_{a,b}\}_{a<b, a,b \in [n]}$$

Now, $\mathcal{S}$ must simulate the invocations of the ideal functionality $\mathcal{F}_{\mathsf{bc}}$ by the corrupt parties. For all $a \in [n] \setminus \mathcal{I}$, $\mathcal{S}$ "broadcasts"

$$\overrightarrow{\sigma_i} = \left\{\sigma_{a',b'}\right\}_{a'=i \ \vee \ b'=i}$$

to all the corrupt parties. For any $i \in \mathcal{I}$, if $\mathcal{A}$ instructs P_i to invoke $\mathcal{F}_{\mathsf{bc}}$ with input $\overrightarrow{\sigma_i}$, $\mathcal{S}$ "broadcasts" $\overrightarrow{\sigma_i}$ to all the corrupt parties.

Once round 0 is completed, $\mathcal{S}$ has completed simulating the invocations of the input phase of all the instances of the ideal functionality $\mathcal{F}_{\mathsf{SyX}}$ and the ideal functionality $\mathcal{F}_{\mathsf{bc}}$. What remains is to determine whether the adversary wishes to obtain its output and to simulate the invocations of the trigger phases of the instances of the ideal functionality $\mathcal{F}_{\mathsf{SyX}}$ that the adversary instructs corrupt parties to trigger. We consider two cases. First, we make the following definition: a witness w is *valid for round* r if

$$w = \left(\overrightarrow{\sigma}, \overrightarrow{z}, \overrightarrow{\omega^z}\right)$$

such that

- For all $a, b \in [n]$ with $a < b$,

  $$\mathcal{V}.\mathsf{Verify}\left(\sigma_{a,b,1}, (a,b); \mathsf{vk}_a\right) = 1$$

  and

  $$\mathcal{V}.\mathsf{Verify}\left(\sigma_{a,b,2}, (a,b); \mathsf{vk}_b\right) = 1$$

- $|\overrightarrow{z}| = \left|\overrightarrow{\omega^z}\right| = r$

- $\mathsf{Open}\left(c_j^z, \omega_j^z, z_j\right) = 1$ for every $j \in [r]$.

Case A. $\mathcal{I} \neq [t]$**.** $\mathcal{S}$ determines the smallest value $i \in [t]$ such that $i \notin \mathcal{I}$. Since $\mathcal{I} \neq [t]$, i is well-defined. We first discuss how $\mathcal{S}$ simulates certain invocations of the trigger phases of the instances of the ideal functionality $\mathcal{F}_{\mathsf{SyX}}$ that the adversary instructs corrupt parties to trigger.

- Suppose the adversary instructs a corrupt party, say P_j for $j \in \mathcal{I} \cap [t]$, to trigger an instance of $\mathcal{F}_{\mathsf{SyX}}$ involving another corrupt party, say P_k for $k \in \mathcal{I}$, with a *valid* witness w in round $k-1$ with $j < k < i$. $\mathcal{S}$ sends $(w, (\omega_j^z, z_j, \omega_k^z, z_k))$ to parties P_j and P_k.

- Suppose the adversary instructs a corrupt party to trigger an instance of $\mathcal{F}_{\mathsf{SyX}}$ with an *invalid* witness. $\mathcal{S}$ simply sends no response.

Suppose the adversary does not instruct a corrupt party, say P_j for some $j \in \mathcal{I} \cap [t]$, to trigger an instance of $\mathcal{F}_{\mathsf{SyX}}$ involving P_i with a *valid* witness with $j < i$ and the round counter exceeds $i-1$, $\mathcal{S}$ forwards $\{(x_i, f_i)\}_{i \in \mathcal{I}}$ to the trusted party computing $\mathcal{F}_{\mathsf{MPC}}$ with fairness, with (x_j, f_j) replaced with abort for some $j \in \mathcal{I}$, receives $\perp$ as the output of all parties, and aborts itself. Otherwise, as soon as the adversary instructs a corrupt party to trigger an instance of $\mathcal{F}_{\mathsf{SyX}}$ involving P_i with a *valid* witness with $j < i$ in round $i-1$, $\mathcal{S}$ forwards $\{(x_i, f_i)\}_{i \in \mathcal{I}}$ to the trusted party computing $\mathcal{F}_{\mathsf{MPC}}$ with fairness. It receives the corrupt parties outputs, namely, $\{y_i\}_{i \in \mathcal{I}}$. $\mathcal{S}$ chooses the outputs of the honest party completely at random, that is, it samples random strings $y_i \xleftarrow{\$} \{0,1\}^*$ of length equal to the length of the ith output of F, for $i \in [n] \setminus \mathcal{I}$. $\mathcal{S}$ then constructs

$$y = y_1 \| \ldots \| y_n$$

It then defines

$$z = y \oplus \alpha$$

$\mathcal{S}$ then computes

$$z_h = z \oplus \bigoplus_{i \in [n] \setminus \{h\}} z_i$$

and constructs

$$\overrightarrow{z} = (z_1, \ldots, z_n)$$

$\mathcal{S}$ computes $\omega_h^z \xleftarrow{\$} \widetilde{\mathsf{Open}}(\mathsf{state}_h, z_h)$ and constructs

$$\overrightarrow{\omega^z} = (\omega_1^z, \ldots, \omega_n^z)$$

Note that, at this point, $\mathcal{S}$ has every value ever used in the protocol. $\mathcal{S}$ sends $(w, (\omega_i^z, z_i, \omega_j^z, z_j))$ to P_j. Going forward, $\mathcal{S}$ simulates invocations of the trigger phases of the instances of the ideal functionality $\mathcal{F}_{\mathsf{SyX}}$ that involve corrupt parties as follows.

- Suppose the adversary instructs a corrupt party, say P_j for $j \in \mathcal{I} \cap [t]$, to trigger an instance of $\mathcal{F}_{\mathsf{SyX}}$ involving another corrupt party, say P_k for $k \in \mathcal{I}$, with a *valid* witness w in round $k-1$ with $j < k$, $\mathcal{S}$ sends $(w, (\omega_j^z, z_j, \omega_k^z, z_k))$ to parties P_j and P_k.
- Suppose the adversary instructs a corrupt party, say P_j for $j \in \mathcal{I} \cap [t]$, to trigger an instance of $\mathcal{F}_{\mathsf{SyX}}$ involving an honest party, say P_k for $k \in [n] \setminus \mathcal{I}$, with a *valid* witness w in round $k-1$ with $j < k$, $\mathcal{S}$ sends $(w, (\omega_j^z, z_j, \omega_k^z, z_k))$ to P_j.
- Suppose the adversary instructs a corrupt party to trigger an instance of $\mathcal{F}_{\mathsf{SyX}}$ with an *invalid* witness. $\mathcal{S}$ simply sends no response.
- Suppose an honest party, say P_k for $k \in [n] \setminus \mathcal{I}$, triggers an instance of $\mathcal{F}_{\mathsf{SyX}}$ involving a corrupt party, say P_j for j, $\mathcal{S}$ sends $(w, (\omega_k^z, z_k, \omega_j^z, z_j))$ to P_j.

Case B. $\mathcal{I} = [t]$**.** We first discuss how $\mathcal{S}$ simulates certain invocations of the trigger phases of the instances of the ideal functionality $\mathcal{F}_{\mathsf{SyX}}$ that the adversary instructs corrupt parties to trigger.

- Suppose the adversary instructs a corrupt party, say P_j for $j \in \mathcal{I}$, to trigger an instance of $\mathcal{F}_{\mathsf{SyX}}$ involving another corrupt party, say P_k for $k \in \mathcal{I}$, with a *valid* witness w in round $k-1$ with $j < k$. $\mathcal{S}$ sends $(w, (\omega_j^z, z_j, \omega_k^z, z_k))$ to parties P_j and P_k.

- Suppose the adversary instructs a corrupt party, say P_j for $i \in \mathcal{I}$, to trigger an instance of $\mathcal{F}_{\mathsf{SyX}}$ involving an honest party, say P_k for $k \in [n] \setminus \mathcal{I}$, with a *valid* witness w in round $k-1$ with $j < k < n$, $\mathcal{S}$ sends $(w, (\omega_j^z, z_j, \omega_k^z, z_k))$ to P_j.
- Suppose the adversary instructs a corrupt party to trigger an instance of $\mathcal{F}_{\mathsf{SyX}}$ with an *invalid* witness. $\mathcal{S}$ simply sends no response.

Suppose the adversary does not instruct a corrupt party, say P_j for some $j \in \mathcal{I}$, to trigger an instance of $\mathcal{F}_{\mathsf{SyX}}$ involving P_n with a *valid* witness and the round counter exceeds $n-1$, $\mathcal{S}$ forwards $\{(x_i, f_i)\}_{i \in \mathcal{I}}$ to the trusted party computing $\mathcal{F}_{\mathsf{MPC}}$ with fairness, with (x_j, f_j) replaced with abort for some $j \in \mathcal{I}$, receives $\perp$ as the output of all parties, and aborts itself. Otherwise, as soon as the adversary instructs a corrupt party to trigger an instance of $\mathcal{F}_{\mathsf{SyX}}$ involving P_n with a *valid* witness in round $n-1$, $\mathcal{S}$ forwards $\{(x_i, f_i)\}_{i \in \mathcal{I}}$ to the trusted party computing $\mathcal{F}_{\mathsf{MPC}}$ with fairness. It receives the corrupt parties outputs, namely, $\{y_i\}_{i \in \mathcal{I}}$. $\mathcal{S}$ chooses the outputs of the honest party completely at random, that is, it samples random strings $y_i \xleftarrow{\$} \{0,1\}^*$ of length equal to the length of the ith output of F, for $i \in [n] \setminus \mathcal{I}$. $\mathcal{S}$ then constructs

$$y = y_1 \| \dots \| y_n$$

It then defines

$$z = y \oplus \alpha$$

$\mathcal{S}$ then computes

$$z_n = z \oplus \bigoplus_{i \in [n-1]} z_i$$

and constructs

$$\overrightarrow{z} = (z_1, \dots, z_n)$$

$\mathcal{S}$ computes $\omega_n^z \xleftarrow{\$} \widetilde{\mathsf{Open}}(\mathsf{state}_n, z_n)$ and constructs

$$\overrightarrow{\omega^z} = (\omega_1^z, \dots, \omega_n^z)$$

Note that, at this point, $\mathcal{S}$ has every value ever used in the protocol. $\mathcal{S}$ sends $(w, (\omega_j^z, z_j, \omega_n^z, z_n))$ to P_j. Going forward, $\mathcal{S}$ simulates invocations of the trigger phases of the instances of the ideal functionality $\mathcal{F}_{\mathsf{SyX}}$ that the adversary instructs corrupt parties to trigger as follows.

- Suppose the adversary instructs a corrupt party, say P_j for $j \in \mathcal{I}$, to trigger an instance of $\mathcal{F}_{\mathsf{SyX}}$ involving P_n with a *valid* witness w in round $n-1$, $\mathcal{S}$ sends $(w, (\omega_j^z, z_j, \omega_n^z, z_n))$ to P_j.
- Suppose the adversary instructs a corrupt party to trigger an instance of $\mathcal{F}_{\mathsf{SyX}}$ with an *invalid* witness. $\mathcal{S}$ simply sends no response.

Finally, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs. It is easy to see that the view of $\mathcal{A}$ is indistinguishable in the execution of the protocol Π_{FMPC} and the simulation with $\mathcal{S}$, if $\left(\mathsf{Com}, \mathsf{Open}, \widetilde{\mathsf{Com}}, \widetilde{\mathsf{Open}}\right)$ is an honest-binding commitment scheme and $\mathcal{V}$ is a signature scheme. We therefore conclude that the protocol Π_{FMPC} securely computes $\mathcal{F}_{\mathsf{MPC}}$ with fairness in the $(\mathcal{F}_{\mathsf{bc}}, \mathcal{F}_{\mathsf{MPC}}, \mathcal{F}_{\mathsf{SyX}})$-hybrid model, as required. $\square$

Remark. In the proof of Lemma 11, we ignore some annoying technicalities. For instance, the adversary may cause the honest parties to abort, will be unable to obtain its output but still pointlessly interact with some of the ideal functionalities. In the proof, however, the simulator would have aborted. We note that these details are not particularly enlightening and are of no consequence. One can deal with these sorts of attacks by asking the simulator to wait in these scenarios until the adversary says that it is done and then finally abort if it has to. Thus, we assume, for the purpose of the proof, that if the adversary forces the honest parties to abort in a situation where it will be unable to obtain its output, without loss of generality, it halts. Other examples of such technicalities are when the adversary sends "unexpected" messages, "incomplete" messages, etc. Note that such messages can be easily detected and ignored, and do not affect the protocol in any way.

6.6 Getting to the $\mathcal{F}_{\mathsf{SyX}}$-hybrid model

Combining Lemmas 1, 4, 8 and 11, we obtain the following theorem.

Theorem 4. *Consider n parties $P_1, \ldots, P_n$ in the* point-to-point *model. Then, assuming the existence of one-way functions, there exists a protocol π which securely computes $\mathcal{F}_{\mathsf{MPC}}$ with fairness in the presence of t-threshold adversaries for any $0 \leq t < n$ in the $(\mathcal{F}_{\mathsf{OT}}, \mathcal{F}_{\mathsf{SyX}})$-hybrid model where only parties $P_1, \ldots, P_t$ can trigger their instances of $\mathcal{F}_{\mathsf{SyX}}$.*

As discussed in Section 3.14, $\mathcal{F}_{\mathsf{2PC}}$, and hence $\mathcal{F}_{\mathsf{OT}}$, can be realized in the $\mathcal{F}_{\mathsf{SyX}}$-hybrid model. We thus have the following theorem.

Theorem 5. *Consider n parties $P_1, \ldots, P_n$ in the* point-to-point *model. Then, assuming the existence of one-way functions, there exists a protocol π which securely computes $\mathcal{F}_{\mathsf{MPC}}$ with fairness in the presence of t-threshold adversaries for any $0 \leq t < n$ in the $\mathcal{F}_{\mathsf{SyX}}$-hybrid model where only parties $P_1, \ldots, P_t$ can trigger their instances of $\mathcal{F}_{\mathsf{SyX}}$.*

It is important to note that via this transformation, we have not introduced a need for the parties to have access to multiple instances of the ideal functionality $\mathcal{F}_{\mathsf{SyX}}$ as opposed to one. This is because, in the protocol Π_{FMPC}, the ideal functionality $\mathcal{F}_{\mathsf{OT}}$ will only be used to emulate the ideal functionality $\mathcal{F}_{\mathsf{MPC}}$. During this stage, we do not make any use of the ideal functionality $\mathcal{F}_{\mathsf{SyX}}$. Once we are done with the signle invocation of $\mathcal{F}_{\mathsf{MPC}}$, we only invoke the ideal functionality $\mathcal{F}_{\mathsf{SyX}}$. As a consequence, parties can reuse the same instance of $\mathcal{F}_{\mathsf{SyX}}$ to first emulate $\mathcal{F}_{\mathsf{OT}}$ and then as a complete $\mathcal{F}_{\mathsf{SyX}}$ functionality. We note that this however does increase the number of times the functionality is invoked.

Combining this with Theorem 3, we obtain a protocol for fair secure computation against t-threshold adversaries when only t of the parties posses secure attestation processers and each pair of parties where at least one of them has a secure attestation processor, share common bulletin board.

7 Preprocessing

As described, our protocol in the $\mathcal{F}_{\mathsf{SyX}}$-hybrid model runs in $\mathcal{O}(n)$ rounds. Since our $\mathcal{F}_{\mathsf{SyX}}$ implementation in the $(\mathcal{G}_{\mathsf{att}}, \mathcal{F}_{\mathsf{BB}}, \mathcal{G}_{\mathsf{acrs}})$-hybrid model requires two write queries to $\mathcal{F}_{\mathsf{BB}}$ (i.e., two writes on the blockchain), it follows that the real-world implementation of our protocol will require $\mathcal{O}(n)$

on-chain rounds. Fortunately, following ideas from [KRS20], we can preprocess our use of $\mathcal{F}_{\mathsf{SyX}}$, and minimize the use of blockchain. Specifically, in an optimistic setting where parties do not deviate from the protocol instructions, our (preprocessed) protocol runs entirely off-chain. In fact, we preprocess an $\mathcal{F}_{\mathsf{SyX}}$ instance between parties P_i and P_j such that P_i and P_j can reuse this across many different protocols possibly involving different sets of parties.

The main idea is to let the $\mathcal{F}_{\mathsf{SyX}}$ instance between P_i and P_j give both parties shares of a "master key" $K^{i,j}$ along with commitments on both these shares in the load phase. Concretely, $\mathcal{F}_{\mathsf{SyX}}$ provides $K_i^{i,j}, c_i^{i,j}, c_j^{i,j}$ to P_i and $K_j^{i,j}, c_i^{i,j}, c_j^{i,j}$ to P_j, where $K_i^{i,j} \oplus K_j^{i,j} = K^{i,j}$ and $c_i^{i,j}, c_j^{i,j}$ are respectively commitments on $K_i^{i,j}, K_j^{i,j}$. Note that the load phase is independent of the function that will be computed fairly, and is also independent of the parties involved in the computation.

Next, we will show how to use this setup to emulate the trigger phase of $\mathcal{F}_{\mathsf{SyX}}$ in the "unpreprocessed" fair protocol. Recall that in the "unpreprocessed" version of our fair protocol, the $\mathcal{F}_{\mathsf{SyX}}$ instance between (i, j) was loaded with the commitments $c_1, \ldots, c_j$ along with P_j's secret share y_j. Then to trigger the $\mathcal{F}_{\mathsf{SyX}}$ instance between (i, j), party P_i needed to provide openings to the first $j - 1$ commitments $c_1, \ldots, c_{j-1}$. In the preprocessed case, note that the $\mathcal{F}_{\mathsf{SyX}}$ instance is not loaded with the set of commitments $c_1, \ldots, c_j$. Our main strategy will be to let the triggering party provide the set of all commitments $c_1, \ldots, c_n$, along with the openings of the first j commitments to $\mathcal{F}_{\mathsf{SyX}}$ in the trigger phase.

We will also let the triggering party provide the protocol specific identifier id along with the start time T of the protocol. (We assume that all parties begin by first agreeing on the values id, T for that protocol instance. In particular, honest parties would reject id values that were used in a previous protocol.) Summarizing, to trigger $\mathcal{F}_{\mathsf{SyX}}$, party P_i will have to provide a tuple $(\mathsf{id}, T, (c_1, \ldots, c_n), (w_1, \ldots, w_j))$. Upon receiving this tuple, $\mathcal{F}_{\mathsf{SyX}}$ performs the following checks: (1) the current time must be $\leq T + j$, and (2) for all $1 \leq k \leq j$, it holds that w_k is a valid opening of c_k. If all checks pass, then $\mathcal{F}_{\mathsf{SyX}}$ outputs a derived key $K_{\mathsf{id}}^{i,j} = \mathsf{Hash}(K^{i,j}, \mathsf{id}, T, c_1, \ldots, c_n)$ and outputs this to both P_i and P_j in a fair manner.

Now, to emulate the trigger phase of $\mathcal{F}_{\mathsf{SyX}}$ in the "unpreprocessed" fair protocol we will need the unfair MPC protocol π_{MPC} to provide an encryption of the trigger output y_j under the derived key $K_{\mathsf{id}}^{i,j}$. For this to work, each party P_i will need to provide $\{K_i^{i,j}, c_i^{i,j}, c_j^{i,j}\}_{j \in [n]}$ as input to π_{MPC} (in addition to agreed upon values id, T, and the input to the function evaluation). As before, π_{MPC} computes the function output, secret shares it, and then computes commitments $c_1, \ldots, c_n$ on the output shares. In addition, π_{MPC} uses the key shares to reconstruct $\{K^{i,j}\}_{i<j}$, and then derives the protocol specific derived keys $\{K_{\mathsf{id}}^{i,j}\}_{i<j}$ from the values $\mathsf{id}, T, c_1, \ldots, c_n$. Then, π_{MPC} computes the ciphertexts $e_{\mathsf{id}}^{i,j} = \mathsf{Enc}(K_{\mathsf{id}}^{i,j}, y_j)$ and outputs these ciphertexts to the parties. This way, upon triggering $\mathcal{F}_{\mathsf{SyX}}$, both parties will obtain the derived key $K_{\mathsf{id}}^{i,j}$, and then decrypt the ciphertext $e_{\mathsf{id}}^{i,j}$ using $K_{\mathsf{id}}^{i,j}$ to learn y_j (i.e., exactly as in the "unpreprocessed" protocol).

Note that the unfair MPC protocol π_{MPC} will also check if the input key shares $K_i^{i,j}$ and $K_j^{i,j}$ are consistent with the openings of the commitments $c_i^{i,j}$ and $c_j^{i,j}$ respectively. This is to ensure that parties do not submit invalid key shares, as this would result in the ciphertexts being computed using invalid derived keys. To see why this is a problem, suppose party P_j is the only honest party, and suppose corrupt P_{j+1} supplied an invalid key share $\widehat{K}_{j+1}^{j,j+1} \neq K_{j+1}^{j,j+1}$ to π_{MPC}. Now, in round j, suppose corrupt P_{j-1} obtained y_j by triggering the $(j-1, j)$ $\mathcal{F}_{\mathsf{SyX}}$ instance. Then in round $j+2$, honest P_j would trigger the $(j, j+1)$ $\mathcal{F}_{\mathsf{SyX}}$ instance in an attempt to learn y_{j+1}. While P_j would indeed learn the correct instance specific derived key $K_{\mathsf{id}}^{j,j+1}$ by triggering $\mathcal{F}_{\mathsf{SyX}}$, this key turns out to be useless for decrypting the ciphertext $e_{\mathsf{id}}^{j,j+1}$ since this ciphertext was encrypted under an invalid key. Therefore, this results in honest P_j not learning the final output. On the other hand, the adversary has already learned (the only honest share) y_j, and thus the final output.

As in [KRS20], additional care must be taken to ensure that all parties obtain the ciphertexts before any party receives the set of all commitments. Otherwise, we end up in a situation where honest parties do not have all the ciphertexts they need (to decrypt and learn the output), but corrupt parties have the set of all commitments to start triggering some $\mathcal{F}_{\mathsf{SyX}}$ instances and to try and learn the output. Concretely, suppose P_j is the only honest party, and suppose P_j did not receive the ciphertext $e_{\mathsf{id}}^{j,j+1}$ from the unfair MPC protocol. Now, corrupt P_1 can trigger $(1, j)$ $\mathcal{F}_{\mathsf{SyX}}$ using the commitments, and (valid) openings of $c_1, \ldots, c_{j-1}$ to obtain P_j's share y_j. While honest P_j can still trigger $(j, j+1)$ $\mathcal{F}_{\mathsf{SyX}}$ instance to obtain the (correct) derived key $K_{\mathsf{id}}^{j,j+1}$, it does not have the ciphertext $e_{\mathsf{id}}^{j,j+1}$ and thus will not be able to learn P_{j+1}'s share. This leaves us in a situation where the adversary learns the output but honest P_j is unable to learn the same.

To avoid the problem, we let the unfair MPC protocol π_{MPC} output n-out-of-n additive shares of $\{e_{\mathsf{id}}^{i,j}\}_{i<j}$ and $\{c_i\}_{i\in[n]}$. If some party did not receive its shares, then everyone terminates the protocol. Otherwise, in the next round, parties first reconstruct the ciphertexts $\{e_{\mathsf{id}}^{i,j}\}_{i<j}$. If some party cannot reconstruct the ciphertexts (e.g., it did not receive some of the remaining $n-1$ shares), then all parties terminate the protocol. Otherwise, in the next round, parties reconstruct the set of all commitments $\{c_i\}_{i\in[n]}$. It may be the case that some honest parties did not receive the set of all commitments. However, this is not a problem since corrupt parties will need to trigger $\mathcal{F}_{\mathsf{SyX}}$ to learn the output, and when they do this, $\mathcal{F}_{\mathsf{SyX}}$ would release the set of all commitments (as part of the trigger witness) to honest parties. On the other hand, note that without the set of all commitments no party can trigger any $\mathcal{F}_{\mathsf{SyX}}$ instance to obtain derived keys corresponding to this protocol instance. Therefore, the attack described previously cannot be carried out in the modified protocol.

References

[AAB+19] Zachary Amsden, Ramnik Arora, Shehar Bano, Mathieu Baudet, Sam Blackshear, Abhay Bothra, George Cabrera, Christian Catalini, Konstantinos Chalkias, Evan Cheng, Avery Ching, Andrey Chursin, George Danezis, Gerardo Di Giacomo, David L. Dill, Hui Ding, Nick Doudchenko, Victor Gao, Zhenhuan Gao, FranÃ§ois Garillot, Michael Gorven, Philip Hayes, J. Mark Hou, Yuxuan Hu, Kevin Hurley, Kevin Lewi, Chunqi Li, Zekun Li, Dahlia Malkhi, Sonia Margulis, Ben Maurer, Payman Mohassel, Ladi de Naurois, Valeria Nikolaenko, Todd Nowacki, Oleksandr Orlov, Dmitri Perelman, Alistair Pott, Brett Proctor, Shaz Qadeer, Rain, Dario Russi, Bryan Schwab, Stephane Sezer, Alberto Sonnino, Herman Venter, Lei Wei, Nils Wernerfelt, Brandon Williams, Qinfan Wu, Xifan Yan, Tim Zakian, and Runtian Zhou. The libra blockchain. https://developers.libra.org/docs/assets/papers/the-libra-blockchain.pdf, 2019.

[ABB+18] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 30:1–30:15, 2018.

[ABMO15] Gilad Asharov, Amos Beimel, Nikolaos Makriyannis, and Eran Omri. Complete characterization of fairness in secure two-party computation of boolean functions. In *Theory of Cryptography - 12th Theory of Cryptography Conference, TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings, Part I*, pages 199–228, 2015.

[ADMM14] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Fair two-party computations via bitcoin deposits. In *Financial Cryptography and Data Security - FC 2014 Workshops, BITCOIN and WAHC 2014, Christ Church, Barbados, March 7, 2014, Revised Selected Papers*, pages 105–121, 2014.

[ADMM16] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. *Commun. ACM*, 59(4):76–84, 2016.

[Ash14] Gilad Asharov. Towards characterizing complete fairness in secure two-party computation. In *Theory of Cryptography - 11th Theory of Cryptography Conference, TCC 2014, San Diego, CA, USA, February 24-26, 2014. Proceedings*, pages 291–316, 2014.

[ASW97] N. Asokan, Matthias Schunter, and Michael Waidner. Optimistic protocols for fair exchange. In *CCS '97, Proceedings of the 4th ACM Conference on Computer and Communications Security, Zurich, Switzerland, April 1-4, 1997.*, pages 7–17, 1997.

[ASW00] N. Asokan, Victor Shoup, and Michael Waidner. Optimistic fair exchange of digital signatures. *IEEE Journal on Selected Areas in Communications*, 18(4):593–610, 2000.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10, 1988.

[BK14] Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, pages 421–439, 2014.

[BLOO11] Amos Beimel, Yehuda Lindell, Eran Omri, and Ilan Orlov. $1/p$-secure multiparty computation without honest majority and the best of both worlds. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, pages 277–296, 2011.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptology*, 13(1):143–202, 2000.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 11–19, 1988.

[CDPW07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings*, pages 61–85, 2007.

[CGJ+17] Arka Rai Choudhuri, Matthew Green, Abhishek Jain, Gabriel Kaptchuk, and Ian Miers. Fairness in an unfair world: Fair multiparty computation from public bulletin boards. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 719–728, 2017.

[CHK05] Ran Canetti, Shai Halevi, and Jonathan Katz. Adaptively-secure, non-interactive public-key encryption. In *Theory of Cryptography, Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005, Proceedings*, pages 150–168, 2005.

[CKN03] Ran Canetti, Hugo Krawczyk, and Jesper Buus Nielsen. Relaxing chosen-ciphertext security. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, pages 565–582, 2003.

[CL17] Ran Cohen and Yehuda Lindell. Fairness versus guaranteed output delivery in secure multiparty computation. *J. Cryptology*, 30(4):1157–1186, 2017.

[Cle86] Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*, pages 364–369, 1986.

[FGH+02] Matthias Fitzi, Daniel Gottesman, Martin Hirt, Thomas Holenstein, and Adam D. Smith. Detectable byzantine agreement secure against faulty majorities. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing, PODC 2002, Monterey, California, USA, July 21-24, 2002*, pages 118–126, 2002.

[FGMvR02] Matthias Fitzi, Nicolas Gisin, Ueli M. Maurer, and Oliver von Rotz. Unconditional byzantine agreement and multi-party computation secure against dishonest minorities from scratch. In *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*, pages 482–501, 2002.

[GHKL11] S. Dov Gordon, Carmit Hazay, Jonathan Katz, and Yehuda Lindell. Complete fairness in secure two-party computation. *J. ACM*, 58(6):24:1–24:37, 2011.

[GIM+10] S. Dov Gordon, Yuval Ishai, Tal Moran, Rafail Ostrovsky, and Amit Sahai. On complete primitives for fairness. In *Theory of Cryptography, 7th Theory of Cryptography Conference, TCC 2010, Zurich, Switzerland, February 9-11, 2010. Proceedings*, pages 91–108, 2010.

[GK09] S. Dov Gordon and Jonathan Katz. Complete fairness in multi-party computation without an honest majority. In *Theory of Cryptography, 6th Theory of Cryptography*

Conference, TCC 2009, San Francisco, CA, USA, March 15-17, 2009. Proceedings, pages 19–35, 2009.

[GK10] S. Dov Gordon and Jonathan Katz. Partial fairness in secure two-party computation. In *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Monaco / French Riviera, May 30 - June 3, 2010. Proceedings*, pages 157–176, 2010.

[GKKZ11] Juan A. Garay, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. Adaptively secure broadcast, revisited. In *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC 2011, San Jose, CA, USA, June 6-8, 2011*, pages 179–186, 2011.

[GMPY11] Juan A. Garay, Philip D. MacKenzie, Manoj Prabhakaran, and Ke Yang. Resource fairness and composability of cryptographic protocols. *J. Cryptology*, 24(4):615–658, 2011.

[GMW87] Oded Goldreich, Silvio Micali, and Avi 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, 1987, New York, New York, USA*, pages 218–229, 1987.

[Gol04] Oded Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.

[GV87] Oded Goldreich and Ronen Vainish. How to solve any protocol problem - an efficiency improvement. In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, pages 73–86, 1987.

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, pages 572–591, 2008.

[KGM19] Gabriel Kaptchuk, Matthew Green, and Ian Miers. Giving state to the stateless: Augmenting trustworthy computation with ledgers. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019.

[Kil88] Joe Kilian. Founding cryptography on oblivious transfer. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 20–31, 1988.

[KL12] Alptekin Küpçü and Anna Lysyanskaya. Usable optimistic fair exchange. *Computer Networks*, 56(1):50–63, 2012.

[KMB15] Ranjit Kumaresan, Tal Moran, and Iddo Bentov. How to use bitcoin to play decentralized poker. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and*

Communications Security, Denver, CO, USA, October 12-16, 2015, pages 195–206, 2015.

[KRS20] Ranjit Kumaresan, Srinivasan Raghuraman, and Adam Sealfon. Synchronizable exchange. *IACR Cryptol. ePrint Arch.*, 2020:976, 2020.

[KVV16] Ranjit Kumaresan, Vinod Vaikuntanathan, and Prashant Nalini Vasudevan. Improvements to secure computation with penalties. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 406–417, 2016.

[LSP82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[Mor16] JP Morgan. Quorum whitepaper. *New York: JP Morgan Chase*, 2016.

[MZ17] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 19–38, 2017.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Working Paper*, 2008.

[Nie02] Jesper Buus Nielsen. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings*, pages 111–126, 2002.

[Pin03] Benny Pinkas. Fair secure two-party computation. In *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4-8, 2003, Proceedings*, pages 87–105, 2003.

[PS19] Souradyuti Paul and Ananya Shrivastava. Efficient fair multiparty protocols using blockchain and trusted hardware. In *Progress in Cryptology - LATINCRYPT 2019 - 6th International Conference on Cryptology and Information Security in Latin America, Santiago de Chile, Chile, October 2-4, 2019, Proceedings*, pages 301–320, 2019.

[PSL80] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[PST17] Rafael Pass, Elaine Shi, and Florian Tramèr. Formal abstractions for attested execution secure processors. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*, pages 260–289, 2017.

[RAA+19] Mark Russinovich, Edward Ashton, Christine Avanessians, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, CÃ©dric Fournet, Matthew Kerner, Sid Krishna, Julien Maffre, Thomas Moscibroda Kartik Nayak, Olya Ohrimenko, Felix

Schuster, Roy Schwartz, Alex Shamis, Olga Vrousgou, and Christoph M. Wintersteiger. Ccf: A framework for building confidential verifiable replicated services. *Microsoft*, April 2019.

[RB89] Tal Rabin and Michael 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, May 14-17, 1989, Seattle, Washigton, USA*, pages 73–85, 1989.

[Rom90] John Rompel. One-way functions are necessary and sufficient for secure signatures. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 387–394, 1990.

[SGK19] Rohit Sinha, Sivanarayana Gaddam, and Ranjit Kumaresan. Luciditee: Policy-based fair computing at scale. *IACR Cryptology ePrint Archive*, 2019:178, 2019.

[WW06] Stefan Wolf and Jürg Wullschleger. Oblivious transfer is symmetric. In *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings*, pages 222–232, 2006.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 162–167, 1986.

Disclaimer