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 an $n$-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.
Contents

1 Introduction 1

2 Overview 4
  2.1 Implementing Synchronizable Exchange 5
  2.2 Fair Protocols in the Multiblockchain Setting 6

3 Preliminaries 7
  3.1 Notation and definitions 7
  3.2 Secure Computation 8
    3.2.1 Functionalities 8
    3.2.2 Adversaries 8
    3.2.3 Model 8
    3.2.4 Protocol 9
    3.2.5 Security with Guaranteed Output Delivery 9
    3.2.6 Security with Fairness 10
    3.2.7 Security with Fairness and Identifiable Abort 11
    3.2.8 Security with Abort 12
    3.2.9 Security with Identifiable Abort 13
  3.3 The Hybrid Model 14
  3.4 Fairness versus Guaranteed Output Delivery 15
  3.5 Computing with an Honest Majority 15
  3.6 Oblivious Transfer 16
  3.7 Broadcast 16
  3.8 Authentication Scheme with Public Verification 17
  3.9 Honest-Binding Commitment Schemes 17
  3.10 Digital Signatures 18
  3.11 Authenticated Encryption 19
  3.12 Receiver Non-Committing Encryption 20
  3.13 Non-interactive Non-Committing Encryption 21
  3.14 Synchronizable Exchange 22
  3.15 Attested Execution Secure Processors 22
  3.16 Witness Indistinguishable Proof Systems 24
  3.17 Augmented Global CRS 25
  3.18 Bulletin Board 25

4 Synchronizable Exchange in the \((G_{att}, F_{BB})\)-Hybrid Model 26

5 Synchronizable Exchange with One-Sided Triggers 33
  5.1 Synchronizable Exchange with One-Sided Triggers in the \((G_{att}, G_{acrs}, F_{BB})\)-Hybrid Model 34
6 Fair Secure Computation using $t$ instances of secure hardware
   6.1 The case $n = 3$ ................................................................. 34
   6.2 Extending to arbitrary $n$ ......................................................... 40
   6.3 Protocol ................................................................. 41
   6.4 Correctness ................................................................. 45
   6.5 Security ................................................................. 45
   6.6 Getting to the $\mathcal{F}_{\text{SyX}}$-hybrid model ................................. 51

7 Preprocessing ................................................... 51
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. 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 “∆-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...
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. 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 $\pi_v$ of posting a value $v$ is unforgeable. That is, it is (computationally) infeasible for a party to compute $\pi_v$ 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

---

3Consortium 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 \( P_i \) 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 \( A \) that controls and co-ordinates all the corrupt parties across consortiums. Further, assume \( A \) 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 \( A \) may corrupt more than \( t_i \) within some consortium, or even when \( A \) 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” \( F_{SW} \) abstraction \cite{KRS20}. \( F_{SyX} \) is a 2-party primitive, and is shown in \cite{KRS20} to be complete for secure multiparty computation with fairness where all parties are pairwise connected by independent instances of \( F_{SyX} \). 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 \( F_{SyX} \), then the result of \cite{KRS20} yields a 3-party fair secure computation protocol. Given the above, it is natural to try and apply \cite{KRS20} to our multiblockchain setting to solve the fair computation problem. Unfortunately, \cite{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 \cite{CGJ+17,SGK19} yield fair protocols even with a public blockchain.
not provide an implementation of \( \mathcal{F}_{\text{SyX}} \). One of the main contributions of our work is to provide the first concrete implementation of \( \mathcal{F}_{\text{SyX}} \). Our implementation of \( \mathcal{F}_{\text{SyX}} \) 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 blockchain. 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 \( \mathcal{F}_{\text{SyX}} \) between them. To derive results in this setting, we work with a restricted variant of \( \mathcal{F}_{\text{SyX}} \) 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 \( \mathcal{F}_{\text{SyX}} \) between them. In this setting, we design new fair multiparty protocols in a \( \mathcal{F}_{\text{SyX}} \)-hybrid model where not all pairs of parties are connected by an \( \mathcal{F}_{\text{SyX}} \) 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 \( \mathcal{F}_{\text{SyX}} \) 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 \( \mathcal{F}_{\text{SyX}} \) 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 \( 1 < b < \binom{n}{2} \) blockchains where each pair of parties both have read/write access on one of the \( b \) blockchains.
2.1 Implementing Synchronizable Exchange

Synchronizable exchange $\mathcal{F}_{\text{SyX}}$ [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 $\phi_1, \phi_2$ are boolean predicates. Upon receiving these inputs, $\mathcal{F}_{\text{SyX}}$ 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 $\mathcal{F}_{\text{SyX}}$ following which $\mathcal{F}_{\text{SyX}}$ checks if $\phi_i(w_i) = 1$. If that is indeed the case, then $\mathcal{F}_{\text{SyX}}$ 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 $\mathcal{F}_{\text{SyX}}$ by giving only one designated party, say $P_i$, the ability to trigger $\mathcal{F}_{\text{SyX}}$. This is done easily by setting $\phi_j \equiv 0$, thereby ensuring that $P_j$ can never trigger $\mathcal{F}_{\text{SyX}}$. Note that $P_i$ will still need to provide a valid witness that satisfies $\phi_i$. Next, we show how to implement this variant when only $P_i$ possesses a TEE.

Implementing $\mathcal{F}_{\text{SyX}}$ with one-sided trigger. We now sketch the $\mathcal{F}_{\text{SyX}}$ 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 $e_1 = \text{Enc}(pk_j, y_1)$ 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 $\pi_1$. Then $P_i$ feeds $\pi_1$ to $T_i$ which then releases the output $y_1$ to $P_i$. Note that $P_i$ 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 $e_2 = \text{Enc}(pk_j, y_2)$ to $P_i$. As before, $P_i$ posts $e_2$ on $B$, gets the proof of publication $\pi_2$, then feeds $\pi_2$ 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 $P_i$ 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 $\pi_1$ (resp. $\pi_2$) 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 $P_i$ can prevent $P_i$ from learning the output of $f_1$ or $f_2$. Next, we sketch how to extend these ideas to implement $\mathcal{F}_{\text{SyX}}$ where either party can trigger.

Implementing $\mathcal{F}_{\text{SyX}}$. 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 \( e = \text{Enc}(ek, w_i\|f_2(x_1, x_2, w_i)) \) to \( P_i \). \( P_i \) then posts \( e \) on the blockchain to obtain a proof of publication \( \pi \), which it then sends to \( T_i \). Upon receiving the proof \( \pi \), \( T_i \) 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 \( e, \pi \) to \( T_j \), which then first checks if \( \pi \) 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 \( \pi \) 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 \( \mathcal{F}_{\text{SyX}} \) (and its variant with one-sided trigger) will be described in the \((\mathcal{G}_{\text{att}}, \mathcal{F}_{\text{BB}}, \mathcal{G}_{\text{acs}})\)-hybrid model, where (1) \( \mathcal{G}_{\text{att}} \) is a global ideal functionality described in [PST17] that captures attested executions, and (2) \( \mathcal{F}_{\text{BB}} \) 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) \( \mathcal{G}_{\text{acs}} \) 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 \( \mathcal{F}_{\text{SyX}} \) 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 \( \mathcal{F}_{\text{SyX}} \) 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 \( \mathcal{F}_{\text{SyX}} \) instances. For the sake of simplicity, we represent this setting with an “\( \mathcal{F}_{\text{SyX}}\)-digraph” \( G \), where the vertices represent the \( n \) parties, and edges represent \( \mathcal{F}_{\text{SyX}} \) instances. More concretely, if \( t \) parties possess a TEE, then \( G \) consists of \( O(nt) \) 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 “\( \mathcal{F}_{\text{SyX}}\)-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 \( c = (c_1, \ldots, c_n) \). The MPC protocol outputs to party \( P_i \) the values \( y_i, c \). 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 \( \mathcal{F}_{ SyX } \) 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 \( i \leq t \), we have that \((i,j) \in G\) for all \( j \in [n] \) (and these are the only edges in \( G \)). Consider the \( \mathcal{F}_{ SyX } \) instance associated with \((i,j) \in G \) with \( i \leq t \) 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 \( \phi_i \) 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 \( y_j \), in round \( j+1 \) for \( 2 \leq j \leq n \), each party \( i \) with \( i \leq t \) and \( i < j \), triggers (if possible) the \( \mathcal{F}_{ SyX } \) 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 \( \mathcal{F}_{ SyX } \) instance associated with \( j \). This consequently means that \( P_j \) would learn all values \( y_1, \ldots, y_j \). If \( j \leq t \), then for all \( k > j \), in round \( k+1 \), party \( P_j \) can trigger the \((j,k) \) \( \mathcal{F}_{ SyX } \) 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 \( i \leq t \) to trigger the \( \mathcal{F}_{ SyX } \) instance associated with \((i,n) \in G \). If \( P_i \) triggers this \( \mathcal{F}_{ SyX } \) 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 \( \vec{a} \) denote vectors. By \( a_i \) we denote the \( i \)-th element of the vector \( \vec{a} \). For a vector \( \vec{a} \) of length \( n \in \mathbb{N} \) and an index set \( I \subseteq [n] \), we denote by \( \vec{a} \mid I \) the vector consisting of (ordered) elements from the set \( \{a_i\}_{i \in I} \). By \( \text{poly}(\cdot) \), 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 \( \lambda \). Informally, we say that a function is negligible, denoted by \( \text{negl} \), if it vanishes faster than the inverse of any polynomial. If \( S \) is a set, then \( x \overset{\$}{\leftarrow} S \) indicates the process of selecting \( x \) uniformly at random over \( S \) (which in particular assumes that \( S \) can be sampled efficiently). Similarly, \( x \overset{\$}{\leftarrow} \mathcal{A}(\cdot) \) 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 \textit{statistical distance} is

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

We say that \( \mathcal{X} \) and \( \mathcal{Y} \) are \( \epsilon \)-close if \( \text{SD} (\mathcal{X}, \mathcal{Y}) \leq \epsilon \) and this is denoted by \( \mathcal{X} \approx \epsilon \mathcal{Y} \). We say that \( \mathcal{X} \) and \( \mathcal{Y} \) are identical if \( \text{SD} (\mathcal{X}, \mathcal{Y}) = 0 \) 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 \( \vec{x} = (x_1, \ldots, x_n) \in \mathcal{X}^n \).

3.2.1 Functionalities

A functionality \( f \) is a randomized process that maps \( n \)-tuples of inputs to \( n \)-tuples of outputs, that is, \( f : \mathcal{X}^n \to \mathcal{Y}^n \). 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 \( 0 \leq t < n \). 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 point-to-point model. We sometimes assume that the parties are given access to a physical broadcast channel (defined in Section 3.7) in addition to the point-to-point network; we refer to this model as the 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) in addition to a fully connected point-to-point network; we refer to this model as the OT-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 OT-broadcast model.

\(^7\)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 \( \mathcal{F}_{bc} \)-hybrid model. See Section 3.3.

\(^11\)This can also be viewed as working in the \( \mathcal{F}_{OT} \)-hybrid model. See Section 3.3.

\(^12\)This can also be viewed as working in the \( (\mathcal{F}_{bc}, \mathcal{F}_{OT}) \)-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 \( (f^1(\vec{x}), \ldots, f^n(\vec{x})) \). 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 \( A \) who has corrupted at most \( t \) parties, where \( 0 \leq t < n \). Denote by \( I \subseteq [n] \) the set of indices of the parties corrupted by \( A \). 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 \( A \) 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 \( A \) 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 \( x'_i \not\in \mathcal{X} \) 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 \( A \) outputs an arbitrary (probabilistic polynomial-time computable) function of its view.

We let \( \text{Ideal}^{g,d}_{f,L,S(z)}(\vec{x}, \lambda) \) 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 \( \pi \) is executed by \( P_1, \ldots, P_n \) (and there is no trusted party). In this case, the adversary \( A \) 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 \( \pi \).
Let \( f \) be as above and let \( \pi \) be an \( n \)-party protocol computing \( f \). Let \( A \) be a non-uniform probabilistic polynomial-time machine with auxiliary input \( z \). We let \( \text{REAL}_{\pi,I,A}(z)(x_1, \ldots, x_n, \lambda) \) be the random variable consisting of the view of the adversary and the output of the honest parties following an execution of \( \pi \) 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 \( \pi \) is said to securely compute \( f \) with guaranteed output delivery if for every non-uniform probabilistic polynomial-time adversary \( A \) in the real model, there exists a non-uniform probabilistic polynomial-time adversary \( S \) in the ideal model such that for every \( I \subseteq [n] \) with \( |I| \leq t \),

\[
\{ \text{Ideal}^{g.d.}_{f,I,S(z)}(\vec{x}, \lambda) \}_{\vec{x} \in \mathcal{X}^n, z \in \{0,1\}^*} \equiv \{ \text{REAL}_{\pi,I,A}(z)(\vec{x}, \lambda) \}_{\vec{x} \in \mathcal{X}^n, z \in \{0,1\}^*}
\]

We will sometimes relax security to statistical or computational definitions. A protocol is statistically secure if the random variables \( \text{Ideal}^{g.d.}_{f,I,S(z)}(\vec{x}, \lambda) \) and \( \text{REAL}_{\pi,I,A}(z)(\vec{x}, \lambda) \) 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 \( A \) who has corrupted at most \( t \) parties, where \( 0 \leq t < n \). Denote by \( I \subseteq [n] \) the set of indices of the parties corrupted by \( A \). 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 \( A \) 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 \( A \) may send any values of their choice. In addition, there exists a special \text{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 \( x'_i = \text{abort} \), the trusted party sends \( \bot \) to all the parties. Otherwise, the trusted party chooses \( r \) uniformly at random, computes \( z_i = f^*(x'_1, \ldots, x'_n; r) \) 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 \( A \) outputs an arbitrary (probabilistic polynomial-time computable) function of its view.
We let $\text{Ideal}^{\text{fair}}_{f,I,S(z)}(\overrightarrow{x},\lambda)$ 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 $\pi$ is said to securely compute $f$ with fairness if for every non-uniform probabilistic polynomial-time adversary $A$ in the real model, there exists a non-uniform probabilistic polynomial-time adversary $S$ in the ideal model such that for every $I \subseteq [n]$ with $\lvert I \rvert \leq t$,

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

We will sometimes relax security to statistical or computational definitions. A protocol is statistically secure if for every non-uniform probabilistic polynomial-time adversary $A$, the random variables $\text{Ideal}^{\text{fair}}_{f,I,S(z)}(\overrightarrow{x},\lambda)$ and $\text{Real}_{\pi,I,A}(z)(\overrightarrow{x},\lambda)$ 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 $A$ who has corrupted at most $t$ parties, where $0 \leq t < n$. Denote by $I \subseteq [n]$ the set of indices of the parties corrupted by $A$. 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 $A$ 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 $A$ 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 $i^* \in I$ and sets $x'_i = (\text{abort}, i^*)$. Denote the inputs sent to the trusted party by $x'_1, \ldots, x'_n$.

- **Trusted party sends outputs:** If $x'_i \not\in \mathcal{X}$ 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 $x'_i = (\text{abort}, i^*)$ and $i^* \in I$, the trusted party sends $(\bot, i^*)$ 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 $A$ outputs an arbitrary (probabilistic polynomial-time computable) function of its view.

We let $\text{Ideal}^{\text{id-fair}}_{f,I,S(z)}(\overrightarrow{x},\lambda)$ 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 $\pi$ is said to securely compute $f$ with fairness and identifiable abort if for every non-uniform probabilistic polynomial-time adversary $A$ in the real model, there exists a non-uniform probabilistic polynomial-time adversary $S$ in the ideal model such that for every $I \subseteq [n]$ with $\lvert I \rvert \leq t$,

$$\left\{ \text{Ideal}^{\text{id-fair}}_{f,I,S(z)}(\overrightarrow{x},\lambda) \right\}_{\overrightarrow{x} \in \chi^n,z \in \{0,1\}^*} \equiv \left\{ \text{Real}_{\pi,I,A}(z)(\overrightarrow{x},\lambda) \right\}_{\overrightarrow{x} \in \chi^n,z \in \{0,1\}^*}$$
We will sometimes relax security to statistical or computational definitions. A protocol is statistically secure if the random variables $\text{Ideal}_{f,I,S}(\vec{x},\lambda)$ and $\text{Real}_{\pi,I,A}(\vec{x},\lambda)$ 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 $A$ who has corrupted at most $t$ parties, where $0 \leq t < n$. Denote by $I \subseteq [n]$ the set of indices of the parties corrupted by $A$. 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 $A$ 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 $A$ 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 $x'_i = \text{abort}$, the trusted party sends $\bot$ 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 $i \in I$ (that is, to the adversary $A$).

- **Trusted party sends outputs to the honest parties:** After receiving its output (as described above), the adversary either sends abort or continue to the trusted party. In the former case the trusted party sends $\bot$ to the honest parties, and in the latter case the trusted party send $z_j$ to $P_j$ for every $j \in [n] \setminus I$.

- **Outputs:** The honest parties output whatever was sent by the trusted party. The corrupted parties output nothing and $A$ outputs an arbitrary (probabilistic polynomial-time computable) function of its view.

We let $\text{Ideal}_{f,I,S}(\vec{x},\lambda)$ 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 $\pi$ is said to securely compute $f$ with abort if for every non-uniform probabilistic polynomial-time adversary $A$ in the real model, there exists a non-uniform probabilistic polynomial-time adversary $S$ in the ideal model such that for every $I \subseteq [n]$ with $|I| \leq t$,

$$\left\{ \text{Ideal}_{f,I,S}(\vec{x},\lambda) \right\} \equiv \left\{ \text{Real}_{\pi,I,A}(\vec{x},\lambda) \right\}$$

We will sometimes relax security to statistical or computational definitions. A protocol is statistically secure if the random variables $\text{Ideal}_{f,I,S}(\vec{x},\lambda)$ and $\text{Real}_{\pi,I,A}(\vec{x},\lambda)$ 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 $A$ who has corrupted at most $t$ parties, where $0 \leq t < n$. Denote by $I \subseteq [n]$ the set of indices of the parties corrupted by $A$. 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 $A$ 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 $A$ 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 $i^* \in I$ and sets $x'_i = (\text{abort}, i^*)$. Denote the inputs sent to the trusted party by $x'_1, \ldots, x'_n$.

- **Trusted party sends outputs to the adversary:** If $x'_i \not\in \mathcal{X}$ 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 $x'_i = (\text{abort}, i^*)$ and $i^* \in I$, the trusted party sends $(\bot, i^*)$ 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 $i \in I$ (that is, to the adversary $A$).

- **Trusted party sends outputs to the honest parties:** After receiving its output (as described above), the adversary either sends $(\text{abort}, i^*)$ where $i^* \in I$, or continue to the trusted party. In the former case the trusted party sends $(\bot, i^*)$ to the honest parties, and in the latter case the trusted party send $z_j$ to $P_j$ for every $j \in [n] \setminus I$.

- **Outputs:** The honest parties output whatever was sent by the trusted party. The corrupted parties output nothing and $A$ outputs an arbitrary (probabilistic polynomial-time computable) function of its view.

We let $\text{IDEAL}_{f,I,S}^{\text{id-abort}}(x, \lambda)$ 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 $\pi$ is said to securely compute $f$ with identifiable abort if for every non-uniform probabilistic polynomial-time adversary $A$ in the real model, there exists a non-uniform probabilistic polynomial-time adversary $S$ in the ideal model such that for every $I \subseteq [n]$ with $|I| \leq t$,

\[ \{ \text{IDEAL}_{f,I,S}^{\text{id-abort}}(x, \lambda) \}_{x \in \mathcal{X}, z \in \{0,1\}^*} \equiv \{ \text{REAL}_{\pi,I,A}(x, \lambda) \}_{x \in \mathcal{X}, z \in \{0,1\}^*} \]

We will sometimes relax security to statistical or computational definitions. A protocol is statistically secure if the random variables $\text{IDEAL}_{f,I,S}^{\text{id-abort}}(x, \lambda)$ and $\text{REAL}_{\pi,I,A}(x, \lambda)$ 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 $\pi$ in the $G$-hybrid model, for some functionality $G$, involves parties sending normal messages to each other (as in the real model) and, in addition, having access to a trusted party computing $G$. 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 $G$ is computed according to the ideal model with abort. In all our protocols in the $G$-hybrid model there will only be sequential calls to $G$, that is, there is at most a single call to $G$ per round, and no other messages are sent during any round in which $G$ 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 $\text{type} \in \{\text{g.d.}, \text{fair}, \text{id-fair}, \text{abort}, \text{id-abort}\}$. Let $G$ be a functionality and let $\pi$ be an $n$-party protocol for computing some functionality $f$, where $\pi$ includes real messages between the parties as well as calls to $G$. Let $A$ be a non-uniform probabilistic polynomial-time machine with auxiliary input $z$. $A$ corrupts at most $t$ parties, where $0 \leq t < n$. Denote by $I \subseteq [n]$ the set of indices of the parties corrupted by $A$. Let $H_{\pi,A(z)}^{G,\text{type}}(\mathcal{F}, \lambda)$ be the random variable consisting of the view of the adversary and the output of the honest parties, following an execution of $\pi$ with ideal calls to a trusted party computing $G$ 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 $(G, \text{type})$-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 $G$-hybrid model. Letting $\pi$ denote the protocol thus constructed (in the $G$-hybrid model), we denote by $\pi^\rho$ the real-world protocol in which calls to $G$ are replaced by sequential execution of a real-world protocol $\rho$ that computes $G$ in the ideal model “type”. “Sequential” here implies that only one execution of $\rho$ is carried out at any time, and no other $\pi$-protocol messages are sent during the execution of $\rho$. The results of [Can00] then imply that if $\pi$ securely computes $f$ in the $(G, \text{type})$-hybrid model, and $\rho$ securely computes $G$, then the composed protocol $\pi^\rho$ 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 $\text{type}_1, \text{type}_2 \in \{\text{g.d.}, \text{fair}, \text{id-fair}, \text{abort}, \text{id-abort}\}$. Let $G$ be an $n$-party functionality. Let $\rho$ be a protocol that securely computes $G$ with $\text{type}_1$, and let $\pi$ be a protocol that securely computes $f$ with $\text{type}_2$ in the $(G, \text{type}_1)$-hybrid model. Then protocol $\pi^\rho$ securely computes $f$ with $\text{type}_2$ in the real model.

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

- When $G = F_{bc}^{13}$, $\text{type} = \text{g.d.}$.
- When $G = F_{OT}^{14}$, $\text{type} = \text{abort}$.

---

13 See Section 3.7.
14 See Section 3.6.
• When $G = \mathcal{F}_{2\text{PC}}$, type = abort.

• When $G = \mathcal{F}_{\text{MPC}}$, type = abort.

• When $G = \mathcal{F}_{\text{SyX}}$, type = g.d..

When working in a hybrid model that uses multiple ideal functionalities, $\mathcal{G}_1, \ldots, \mathcal{G}_k$ with associated types $\text{type}_1, \ldots, \text{type}_k$ for some $k \in \mathbb{N}$, we call it the $(\mathcal{G}_1, \text{type}_1, \ldots, \mathcal{G}_k, \text{type}_k)$-hybrid model. Furthermore, we will suppress $\text{type}_j$ when $\text{type}_j$ 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 $\pi$ which securely computes $f$ with fairness, then there exists a protocol $\pi'$ 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 $\pi$ which securely computes $f$ with fairness, then there exists a protocol $\pi'$ 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 $\pi$ which securely computes $\mathcal{F}_{\text{MPC}}$ 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 $\pi$ which securely computes $\mathcal{F}_{\text{MPC}}$ 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 $\pi$ which securely computes $\mathcal{F}_{\text{MPC}}$ with guaranteed output delivery in the presence of $t$-threshold adversaries for any $0 \leq t < n/2$. 

15
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 $\mathcal{F}_{\text{OT}}$.

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 $\mathcal{F}_{\text{MPC}}$.

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 $\mathcal{F}_{\text{OT}}$, 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 $\mathcal{F}_{\text{OT}}$, 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 $\pi$ which securely computes $\mathcal{F}_{\text{MPC}}$ with abort in the presence of $t$-threshold adversaries for any $0 \leq t < n$.

3.7 Broadcast

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

16
these results hold irrespective of the model we are working in so long as we do not have explicit access to $F_{bc}$.

3.8 Authentication Scheme with Public Verification

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

$$\Pr \left[ \text{sk} \overset{\$}{\leftarrow} T.\text{Gen}(1^\lambda) \quad \sigma = T.\text{Tag}(m; \text{sk}) : T.\text{Verify}(\sigma, m) = 1 \right] = 1$$

An authentication scheme for message space $M_\lambda$ is existentially unforgeable if for any PPT adversary $A$, the following probability is negligible in $\lambda$:

$$\Pr \left[ \begin{cases} \text{sk} \overset{\$}{\leftarrow} T.\text{Gen}(1^\lambda) \\ Q = \emptyset \\ \left\{ m_i \overset{\$}{\leftarrow} A(1^\lambda, Q) \\ \sigma_i = T.\text{Tag}(m_i; \text{sk}) \\ Q = Q \cup \{(m_i, \sigma_i)\}_i \right\} \\ (m, \sigma) \overset{\$}{\leftarrow} A(1^\lambda, Q) \end{cases} : T.\text{Verify}(\sigma, m) = 1 \land (m, \sigma) \notin Q \right]$$

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 $\omega$ that it used when generating the commitment.

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

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

A commitment scheme for message space $M_\lambda$ is honest-binding if it satisfies the following:
Binding (for an honest sender). For all PPT algorithms $A$ (that maintain state throughout their execution), the following probability is negligible in $\lambda$:

$$
\Pr \left[ \begin{array}{l}
\begin{array}{l}
m \overset{\$}{\leftarrow} A(1^k) ; \omega \overset{\$}{\leftarrow} \{0,1\}^* \\
c = \text{Com}(1^\lambda, m ; \omega) \\
(m', \omega') \overset{\$}{\leftarrow} A(c, \omega)
\end{array}
\end{array} : \text{Open}(c, m', \omega') = 1 \land m \neq m'
\right]
$$

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

$$
\left| \Pr \left[ m \overset{\$}{\leftarrow} A(1^\lambda) ; \omega \overset{\$}{\leftarrow} \{0,1\}^* ; c = \text{Com}(1^\lambda, m ; \omega) : A(1^\lambda, c, \omega) = 1 \right] 
\right| 
- \left| \Pr \left[ (c, \text{state}) \overset{\$}{\leftarrow} \widetilde{\text{Com}}(1^\lambda), m \overset{\$}{\leftarrow} A(1^\lambda) ; \omega \overset{\$}{\leftarrow} \text{Open}(\text{state}, m) : A(1^\lambda, c, \omega) = 1 \right] \right|
$$

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

$$
\left| \Pr \left[ (m_0, m_1) \overset{\$}{\leftarrow} A(1^\lambda) ; b \overset{\$}{\leftarrow} \{0,1\}^* ; c \overset{\$}{\leftarrow} \text{Com}(1^\lambda, m) : A(c) = b \right] \right|
$$

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

We will sometimes use the notation $(c, \omega) \overset{\$}{\leftarrow} \text{Com}(m)$ to mean $c = \text{Com}(1^\lambda, m ; \omega)$, suppressing $\lambda$ when it is clear from the context and having the committing algorithm $\text{Com}$ 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 $V = (\text{Gen}, \text{Sign}, \text{Verify})$ such that for all $\lambda \in \mathbb{N}$ and all messages $m \in \mathcal{M}_\lambda$,

$$
\Pr \left[ \begin{array}{l}
\begin{array}{l}
(vk, sk) \overset{\$}{\leftarrow} V.\text{Gen}(1^\lambda) \\
\sigma \overset{\$}{\leftarrow} V.\text{Sign}(m; sk)
\end{array}
\end{array} : V.\text{Verify}(\sigma, m; vk) = 1 \right] = 1
$$

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

$$
\Pr \left[ \begin{array}{l}
\begin{array}{l}
(vk, sk) \overset{\$}{\leftarrow} V.\text{Gen}(1^\lambda) \\
Q = \emptyset \\
\begin{array}{l}
m_i \overset{\$}{\leftarrow} A(1^\lambda, Q) \\
\sigma_i \overset{\$}{\leftarrow} V.\text{Sign}(m_i; sk)
\end{array}
\end{array}
\end{array} : V.\text{Verify}(\sigma, m; vk) = 1 \land (m, \sigma) \notin Q
\right]
$$

[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:

- $\mathcal{E}.\text{Gen}(1^\lambda)$: Given the security parameter, $\lambda$, the key generation algorithm outputs a secret key. This is denoted by: $\text{sk} \xleftarrow{\$} \mathcal{E}.\text{Gen}(1^\lambda)$. This implicitly defines a message space $\mathcal{M}_\lambda$.

- $\mathcal{E}.\text{Enc}(m; \text{sk})$: Given the secret key $\text{sk}$ and a message $m \in \mathcal{M}_\lambda$, the encryption algorithm returns a ciphertext $\text{ct} \xleftarrow{\$} \mathcal{E}.\text{Enc}(m; \text{sk})$.

- $\mathcal{E}.\text{Dec}(\text{ct}; \text{sk})$: Given the secret key $\text{sk}$ and a ciphertext $\text{ct}$, the decryption algorithm returns a message $m \xleftarrow{\$} \mathcal{E}.\text{Dec}(\text{ct}; \text{sk})$, where $m \in \mathcal{M}_\lambda \cup \{\bot\}$.

We make the standard correctness requirement; namely, for any $\text{sk}$ output by $\mathcal{E}.\text{Gen}$ and any $m \in \mathcal{M}_\lambda$, we have $\mathcal{E}.\text{Dec}(\mathcal{E}.\text{Enc}(m; \text{sk}); \text{sk}) = m$. 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 $A$ in the game below is negligible in $\lambda$:

1. The key generation algorithm $\mathcal{E}.\text{Gen}(1^\lambda)$ is run to get $\text{sk}$. The algorithm $A$ is given $1^\lambda$ as input.

2. $A$ 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 $\text{ct} \xleftarrow{\$} \mathcal{E}.\text{Enc}(m_b; \text{sk})$ is computed. $A$ receives $\text{ct}$.

4. $A$ outputs a bit $b' \in \{0, 1\}$.

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

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-$\bot$ 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 $A$ in the game below is negligible in $\lambda$:

1. The key generation algorithm $\mathcal{E}.\text{Gen}(1^\lambda)$ is run to get $\text{sk}$ and $S = \emptyset$ is initialized. The algorithm $A$ is given $1^\lambda$ as input.

2. $A$ may request (repeatedly) for the encryptions of messages of its choice. If $A$ supplies a message $m \in \mathcal{M}_\lambda$, a ciphertext $\text{ct} \xleftarrow{\$} \mathcal{E}.\text{Enc}(m; \text{sk})$ is computed, $S = S \cup \{\text{ct}\}$ is updated and $A$ receives $\text{ct}$.

3. $A$ outputs a challenge ciphertext $\text{ct}^*$. The output of the decryption $m^* \xleftarrow{\$} \mathcal{E}.\text{Dec}(\text{ct}^*; \text{sk})$ is computed. If $m^* \neq \bot$, the value $\text{res} = 1$ is output. Otherwise, the value $\text{res} = 0$ is output.

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

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 $E$ consists of the following five PPT algorithms:

- $E.Gen(1^\lambda)$: Given the security parameter, $\lambda$, the key generation algorithm outputs a key-pair and some auxiliary information. This is denoted by: $(pk, sk, z) \xleftarrow{\$} E.Gen(1^\lambda)$. The public key $pk$ defines a message space $M_\lambda$.

- $E.Enc(m; pk)$: Given the public key $pk$ and a message $m \in M_\lambda$, the encryption algorithm returns a ciphertext $ct \xleftarrow{\$} E.Enc(m; pk)$.

- $E.Dec(ct; sk)$: Given the secret key $sk$ and a ciphertext $ct$, the decryption algorithm returns a message $m \xleftarrow{\$} E.Dec(ct; sk)$, where $m \in M_\lambda \cup \{\bot\}$.

- $E.Enc(pk, sk, z)$: Given the triple $(pk, sk, z)$ output by $E.Gen$, the fake encryption algorithm outputs a “fake ciphertext” $\tilde{ct} \xleftarrow{\$} E.\tilde{Enc}(pk, sk, z)$.

- $E.\tilde{Dec}(pk, sk, z, \tilde{ct}, m)$: Given the triple $(pk, sk, z)$ output by $E.Gen$, a “fake ciphertext” $\tilde{ct}$ output by $E.\tilde{Enc}$ and a message $m \in M_\lambda$, the “fake decryption” algorithm outputs a “fake secret key” $\tilde{sk} \xleftarrow{\$} E.\tilde{Dec}(pk, sk, z, \tilde{ct}, m)$. (Intuitively, $\tilde{sk}$ is a valid-looking secret key for which $\tilde{ct}$ decrypts to $m$.)

We make the standard correctness requirement; namely, for any $(pk, sk, z)$ output by $E.Gen$ and any $m \in M_\lambda$, we have $E.Dec(E.Enc(m; pk); sk) = m$. 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 $E$ be a receiver non-committing encryption scheme. We say that $E$ is secure if the advantage of any PPT algorithm $A$ in the game below is negligible in $\lambda$:

1. The key generation algorithm $E.Gen(1^\lambda)$ is run to get $(pk, sk, z)$.

2. The algorithm $A$ is given $1^\lambda$ and $pk$ as input, and is also given access to a decryption oracle $E.Dec(\cdot; sk)$. It then outputs a challenge message $m \in M_\lambda$.

3. A bit $b$ is chosen at random. If $b = 1$ then a ciphertext $ct \xleftarrow{\$} E.Enc(m; pk)$ is computed, and $A$ receives $(ct, sk)$. Otherwise, a “fake” ciphertext $\tilde{ct} \xleftarrow{\$} E.\tilde{Enc}(pk, sk, z)$ and a “fake” secret key $\tilde{sk} \xleftarrow{\$} E.\tilde{Dec}(pk, sk, z, \tilde{ct}, m)$ are computed, and $A$ receives $(\tilde{ct}, \tilde{sk})$. (After this point, $A$ can no longer query its decryption oracle.) $A$ outputs a bit $b' \in \{0, 1\}$.

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

20
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 \textit{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, $\mathcal{F}_{\text{PKE}}$. 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 $\mathcal{F} = (K, F)$ be a collection of trapdoor permutations, where $K$ denotes an index set and $F = \{f_k\}_{k \in K}$ is a set of permutations with efficiently samplable domains. For every $k \in K$, 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 $\lambda$ outputs a key-trapdoor pair $(k, t_k)$ uniformly at random. Let $H : \{0, 1\}^* \rightarrow \{0, 1\}^{\ell(\lambda)}$ be a random oracle (instantiated by an appropriate hash function). The non-interactive non-committing encryption scheme $\mathcal{E}$ consists of the following algorithms:

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

- $\mathcal{E}.\text{Enc}(m; pk)$: 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 $ct = (f_k(x), H(x) \oplus m)$.

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

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 $y_1, \ldots, y_n \overset{\$}{\leftarrow} \{0, 1\}^{\ell(\lambda)}$. The $m$ ciphertexts are defined to be $\{ct_i\}_{i \in [n]}$ where $ct_i = (f_k(x_i), y_i)$. Then, in order to open the $n$ ciphertexts to a message vector $\vec{m} = (m_1, \ldots, m_n) \in \mathcal{M}_\lambda^n$, $\mathcal{S}$ 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 $\vec{m}$. 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; $\phi_1, \phi_2$ 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 $f_1(x_1, x_2) = \bot$, 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 $P_2$.

*aWe crucially require that $\bot$ is a special symbol different from the empty string. We use $\bot$ as a means of signalling that the input phase of $F_{SyX}$ 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 $F_{SyX}$.

### 3.14 Synchronizable Exchange

Synchronizable exchange is defined as in Figure 4. In order to guarantee termination, 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 $F_{SyX}$ 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 $F_{SyX}$ 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 $F_{SyX}$ 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 $\pi$ which securely computes $F_{MPC}$ with fairness in the presence of $t$-threshold adversaries for any $0 \leq t < n$ in the $F_{SyX}$-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 $\pi$ in the programmable random oracle model which securely preprocesses for and computes an arbitrary (polynomial) number of instances of $F_{MPC}$ with fairness in the presence of $t$-threshold adversaries for any $0 \leq t < n$ in the $F_{SyX}$-hybrid model.

### 3.15 Attested Execution Secure Processors

In this section, we recall the $G_{att}$ 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; \( \phi_1, \phi_2 \) are boolean predicates; \( r \) denotes the current round number; \( \text{INPUT\_TIMEOUT} < \text{TRIGGER\_TIMEOUT} \) are round numbers representing time outs. The functionality proceeds as follows:

- **Load phase.** If \( r > \text{INPUT\_TIMEOUT} \), 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 \( f_i(x_1, x_2, r) \) to \( P_i \) for \( i \in \{1, 2\} \), and go to next phase.

- **Trigger phase.** If \( r > \text{TRIGGER\_TIMEOUT} \), abort. Otherwise, upon receiving input \( w \) from party \( P_i \), check if \( \phi_i(w, r) = 1 \). If yes, then send \((w, f_j(x_1, x_2, w, r))\) to both parties \( P_j \) for \( j \in \{1, 2\} \).

Figure 5: The clock-aware ideal functionality \( \mathcal{F}_{\text{SyX}} \).

---

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

\[ \text{// initialization:} \]
\[ \text{On initialize:} \quad (v_k_{\text{att}}, s_k_{\text{att}}) := \mathcal{V}.\text{Gen}(1^\lambda), \ T = \emptyset \]

\[ \text{// public query interface:} \]
\[ \text{On receive* getpk() from some } P: \text{ send } v_k_{\text{att}} \text{ to } P \]

**Enclave operations**

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

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

Figure 6: The ideal functionality \( \mathcal{G}_{\text{att}} – \text{a global functionality modeling an SGX-like secure processor.} \) 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 \( \sigma \). For honest parties, the functionality verifies that installed enclaves are parametrized by the session ID, \( sid \) of the current protocol instance.
1. Registry. First, $G_{\text{att}}$ is parametrized with a static registry $\text{reg}$ 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 $\text{reg}$ may invoke enclave operations, including:
   
   - **install**: installing a new enclave with a program $\text{prog}$, henceforth referred to as the enclave program. Upon installation, $G_{\text{att}}$ simply generates a fresh enclave identifier $\text{eid}$ and returns the $\text{eid}$. This enclave identifier may now be used to uniquely identify the enclave instance.
   
   - **resume**: resuming the execution of an existing enclave with inputs $\text{inp}$. Upon a resume call, $G_{\text{att}}$ execute the $\text{prog}$ over the inputs $\text{inp}$, an outputs an output $\text{outp}$. $G_{\text{att}}$ would then sign the $\text{prog}$ together with $\text{outp}$ as well as additional metadata, and return both $\text{outp}$ 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 $G_{\text{att}}$ functionality has a manufacturer public key and secret key pair denoted by $(\text{mpk}, \text{msk})$, and is parametrized by a signature scheme $\mathcal{V}$. When an enclave resume operation is invoked, $G_{\text{att}}$ signs any output to be attested with $\text{msk}$ using the signature scheme $\mathcal{V}$. $G_{\text{att}}$ provides the manufacturer public key $\text{mpk}$ to any party upon query, using which, any party can verify an anonymous attestation signed by $G_{\text{att}}$.

The enclave program $\text{prog}$ and all inputs $\text{inp}$ 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 $\text{NIWI}$, for an NP language $\mathcal{L}$ consists of the following algorithms:

- $\text{crs} \leftarrow \text{Gen}(1^\lambda)$: The generation algorithm takes as input the security parameter $\lambda$ and generates a common reference string $\text{crs}$.

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

- $b \leftarrow \text{Verify}(\text{crs}, \text{stmt}, \text{π})$: The proof verification algorithm takes as input the common reference string $\text{crs}$, a statement $\text{stmt}$ and a proof $\text{π}$, 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 $(\text{stmt}, \text{w}) \in R_{\mathcal{L}}$, we have that

$$\Pr[\text{crs} \leftarrow \text{Gen}(1^\lambda), \text{π} \leftarrow \text{Prove}(\text{crs}, \text{stmt}, \text{w}) : \text{Verify}(\text{crs}, \text{stmt}, \text{π}) = 1] = 1$$

\footnote{We assume that a description of the language $\mathcal{L}$ is also provided implicitly as an input.}
On initialize: $(pk_{G_{acs}}, sk_{G_{acs}}) \leftarrow \mathcal{E}.\text{Gen}(1^\lambda)$, $(vk_{V_{acs}}, sk_{V_{acs}}) \leftarrow \mathcal{V}.\text{Gen}(1^\lambda)$, $crs \leftarrow \text{NIWI}$.\text{Gen}(1^\lambda)$

On receive* “crs” from $P$: return $G_{acs}$.mpk := $(pk_{E_{acs}}, acrs, vk_{V_{acs}}, acrs, crs)\leftarrow V.\text{Gen}(1^\lambda)$

On receive* “idk” from $P$: assert $P$ is corrupt, and then return $(V.\text{Sign}(P; sk_{V_{acs}}))$

Figure 7: The ideal functionality $G_{acs}$ – 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 $A$, the following probability is negligible in $\lambda$:

$$\Pr[crs \leftarrow G_{acs}.\text{Gen}(1^\lambda), (\text{stmt}, \pi) \leftarrow A(crs): \text{Verify}(crs, \text{stmt}, \pi) = 1 \land \text{stmt} \notin \mathcal{L}]$$

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

$$\left| \Pr\left[ \begin{array}{c} \text{crs} \leftarrow \mathcal{E}.\text{Gen}(1^\lambda), \\
\text{(stmt}, w_0, w_1) \leftarrow \mathcal{A}(crs) \\
\pi \leftarrow \text{Prove}(crs, \text{stmt}, w_0) \\
\text{stmt} \in \mathcal{R}_L \land (\text{stmt}, w_1) \in \mathcal{R}_L \\
\text{A}(\pi) = 1 \\
\end{array} \right] \right| - \left| \Pr\left[ \begin{array}{c} \text{crs} \leftarrow \mathcal{E}.\text{Gen}(1^\lambda), \\
\text{(stmt}, w_0, w_1) \leftarrow \mathcal{A}(crs) \\
\pi \leftarrow \text{Prove}(crs, \text{stmt}, w_1) \\
\text{stmt} \in \mathcal{R}_L \land (\text{stmt}, w_1) \in \mathcal{R}_L \\
\text{A}(\pi) = 1 \\
\end{array} \right] \right|$$

### 3.17 Augmented Global CRS

The augmented global CRS functionality denoted by $G_{acs}$ is described in Figure 7. We recall the need for this functionality in the context of [PST17]. $G_{acs}$ was first proposed by [CDPW07]. $G_{acs}$ provides a common reference string that is honestly generated. Honest parties never have to query $G_{acs}$ for any additional information. On the other hand, $G_{acs}$ leaves a backdoor for the adversary, such that the adversary can obtain identity keys pertaining to their party identifiers. In practice, the $G_{acs}$ 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
\[
\mathcal{F}_{BB}
\]

// initialization:
On initialize: \( sk_{BB} \leftarrow T.Gen(1^\lambda), t = 0, \text{Ledger} = \emptyset \)

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

// posting to the bulletin board:
On receive* post(\( x \)) from \( P \):
update \( t \leftarrow t + 1 \)
let \( \sigma = T.Tag(t \parallel x; sk_{BB}) \)
update Ledger = Ledger \cup \{(t, x, \sigma)\}
send \((t, \sigma)\) to \( P \)

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

Figure 8: The ideal functionality \( \mathcal{F}_{BB} \) – 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 it as an explicit query. We model the bulletin board as an ideal functionality \( \mathcal{F}_{BB} \) 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 \((\mathcal{G}_{att}, \mathcal{F}_{BB})\)-Hybrid Model

In this section, we show how the ideal functionality for synchronizable exchange \( \mathcal{F}_{SyX} \) can be realized in the \((\mathcal{G}_{att}, \mathcal{F}_{BB})\)-hybrid model. The idea for the construction is the following. Each party loads their inputs to \( \mathcal{F}_{SyX} \) 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 \(F_{\text{SyX}}\). The party that obtains this ciphertext then publishes it using the ideal functionality \(F_{\text{BB}}\) 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 \(F_{\text{BB}}\) 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 \(F_{\text{BB}}\) 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 \(F_{\text{SyX}}\), 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 \(F_{\text{SyX}}\). The party then publishes the ciphertext using the ideal functionality \(F_{\text{BB}}\) 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 \(F_{\text{BB}}\) 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, \(E\) is perfectly correct, and satisfies semantic security and INT-CTXT security, and that \(T\) and \(V\) are existentially unforgeable, it holds that the protocol described in Figures 9, 10, 11 and 12 securely realizes \(F_{\text{SyX}}\) 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 \(S\) as described below.

- Unless otherwise noted later, \(S\) passes through messages between \(P_2\) and \(G_{\text{att}}\).
- \(S\) calls \(eid_1 := G_{\text{att}}.\text{install}(\text{sid}, \text{prog}_{\text{SyX},1}[P_1])\), and \((g^{\nu_1}, \sigma_1) := G_{\text{att}}.\text{resume}(eid_1, \text{“keyex”})\) and sends \((eid_1, g^{\nu_1}, \sigma_1)\) to \(P_2\). \(S\) waits to receive the first message \((eid_2, g^{\nu_2}, \sigma_2)\) from \(P_2\) – if this tuple was not the answer to a previous \(G_{\text{att}}\) query, jump to the exception handler denoted except. At this point, \(eid_2\) is called the challenge \(eid\).
- \(S\) checks that \(V.\text{Verify}(\sigma_2, (\text{sid}, eid_2, \text{prog}_{\text{SyX},2}[P_2], g^{\nu_2}); \text{vk}_{\text{att}})\) succeeds. If not, jump to the exception handler denoted except.
- \(S\) calls \((\text{ct}_1, \_):= G_{\text{att}}.\text{resume}(eid_1, \text{“send”}, g^{\nu_2}, \vec{0})\) and sends \(\text{ct}_1\) to \(P_2\).
- The first time \(P_2\) calls \(G_{\text{att}}.\text{resume}(eid_2, \text{“loadstart”}, \text{inp}_2, \text{ct}_1)\) for some input \(\text{inp}_2\) where \(eid_2\) is the challenge \( eid\), and \(\text{ct}_1\) is what \(S\) has sent, the simulator \(S\) extracts and saves \(\text{inp}_2\).
- \(S\) waits to receive the message \(\text{ct}_2\) from \(P_2\). If \((\text{ct}_2, \_):=\) the not the result of the first \(G_{\text{att}}.\text{resume}(eid_2, \text{“send”}, g^{\nu_1})\) call where \(eid_2\) is the challenge \(eid\), and \(g^{\nu_1}\) was what \(S\) previously sent to \(P_2\), or if no such call has taken place, then jump to the exception handler denoted except.

27
On input ("loadfinish"): \( y_1 \overset{\$}{\leftarrow} \mathbb{Z}_p \), return \( g^{y_1} \)

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

On input ("loadstart", \( ct_2 \)):

assert "send" has been called, assert \( ct_2 \) not seen
\( inp_2 := \mathcal{E}.\text{Dec}(ct_2; sk) \), assert that decryption succeeds
parse \( inp_1 := (sid, INPUT\_TIMEOUT, TRIGGER\_TIMEOUT, x_1, f) \)
parse \( inp_2 := (sid', INPUT\_TIMEOUT', TRIGGER\_TIMEOUT', x_2, f') \)
assert that: \( sid = sid', f = f', INPUT\_TIMEOUT = INPUT\_TIMEOUT' \), and
\( TRIGGER\_TIMEOUT = TRIGGER\_TIMEOUT' \)
assert \( r \leq INPUT\_TIMEOUT \), parse \( f = (f_1, f_2, \phi_1, \phi_2) \)
\( z_1^t = f_1(x_1, x_2, r) \), \( z_2^t = f_2(x_1, x_2, r) \), \( ct_{load} := \mathcal{E}.\text{Enc}(z_2^t; sk) \), return \( (ct_{load}, \text{"ok"}) \)

On input ("loadfinish", \( \sigma_3, t_{load}, \sigma_4, v \)):

if \( v \neq \bot \), return \( (v, \text{"ok"}) \)
assert "loadstart" has been called and returned (\( \cdot, \text{"ok"} \))
assert \( \mathcal{T}.\text{Verify}(\sigma_3, (sid, eid_1, prog_{SYX,1}[P_1], (ct_{load}, \text{"ok"})); \mathcal{V}.\text{att}) \)
assert \( \mathcal{T}.\text{Verify}(\sigma_4, t_{load} \parallel \text{"load"} \parallel \text{"sid"} || ct_{load} || \sigma_3) \)
assert \( t_{load} \leq INPUT\_TIMEOUT \), set \( i_1 := 0, i_2 := 0 \), return \( (z_1^t, \text{"ok"}) \)

On input ("triggerstart", \( w_1 \)):

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

On input ("triggerfinish", \( \sigma_5, i_1, t_{trigger1,i_1}, \sigma_6, i_1 \)):

assert "triggerstart" has been called and returned (\( \cdot, i_1, \text{"ok"} \))
assert "triggerfinish" has never previously returned (\( \cdot, i_1, \text{"ok"} \))
assert \( \mathcal{V}.\text{Verify}(\sigma_5, i_1, (sid, eid_1, prog_{SYX,1}[P_1], (ct_{trigger1,i_1}, i_1, \text{"ok"})); \mathcal{V}.\text{att}) \)
assert \( \mathcal{T}.\text{Verify}(\sigma_6, t_{trigger1,i_1} \parallel \text{"trigger1"} \parallel \text{"sid"} || ct_{trigger1,i_1} || \sigma_5) \)
assert \( t_{trigger1,i_1} \leq TRIGGER\_TIMEOUT \), return \( (z_{2,i_1}, i_1, \text{"ok"}) \)

On input ("triggerbyother", \( ct_{trigger2,i_2+1}, \sigma_7, i_2+1, \sigma_8, i_2+1, v \)):

if \( v \neq \bot \), \( i_2 := i_2 + 1 \), return \( (v, i_2, \text{"ok"}) \)
assert "loadfinish" has been called and returned (\( \cdot, \text{"ok"} \))
assert \( i_2 = 0 \) or "triggerbyother" has been called and returned (\( \cdot, i_2 - 1, \text{"ok"} \))
and "triggerbyother" has never previously returned (\( \cdot, i_2, \text{"ok"} \))
assert \( \mathcal{V}.\text{Verify}(\sigma_7, i_2+1, (sid, eid_2, prog_{SYX,2}[P_2], (ct_{trigger2,i_2+1}, i_2 + 1, \text{"ok"})); \mathcal{V}.\text{att}) \), and
\( \mathcal{T}.\text{Verify}(\sigma_8, i_2+1, t_{trigger2,i_2+1} \parallel \text{"trigger2"} || ct_{trigger2,i_2+1} || \sigma_7, i_2+1) \)
assert \( t_{trigger2,i_2+1} \leq TRIGGER\_TIMEOUT \)
\( z_{3,i_2+1} := \mathcal{E}.\text{Dec}(ct_{trigger2,i_2+1}; sk) \), assert that decryption succeeds, \( i_2 := i_2 + 1 \), return \( (z_{3,i_2}, i_2, \text{"ok"}) \)

\[ \text{prog}_{SYX,1}[P_1] \]

Figure 9: Program installed in the secure hardware of Party \( P_1 \).
On input* ("keyex"): $y_2 \leftarrow \mathbb{Z}_p$, return $g^{y_2}$

On input ("loadstart"), inp$_2$, ct$_1$:
assert "keyex" has been called, assert ct$_1$ not seen
inp$_1 := E$.Dec(ct$_1$; sk), assert that decryption succeeds
parse inp$_1 := (sid$, INPUT_TIMEOUT, TRIGGER_TIMEOUT, $x_1$, f)
parse inp$_2 := (sid'$, INPUT_TIMEOUT', TRIGGER_TIMEOUT', $x_2$, f')
assert that: sid = sid', f = f', INPUT_TIMEOUT = INPUT_TIMEOUT', and
TRIGGER_TIMEOUT = TRIGGER_TIMEOUT'
assert $r \leq$ 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", sk := ($g^{y_1}$)$^{y_2}$, ct := $E$.Enc(inp$_2$; sk), return ct

On input ("loadfinish"), ct$_{load}$, $\sigma_3$, $t_{load}$, $\sigma_4$, v):
if $v \neq \bot$, return (v, "ok")
assert "loadstart" has been called and returned "ok", assert ct$_{load}$ not seen
assert $\forall$.Verify($\sigma_3$, (sid, eid$_1$, prog$_{SYX,1}$[$P_1$], (ct$_{load}$, "ok")); $\varphi_{att}$)
assert $T$.Verify($\sigma_4$, $t_{load}$ $\parallel$ "load" $\parallel$ sid $\parallel$ ct$_{load}$ $\parallel$ $\sigma_3$)
assert $t_{load} \leq$ INPUT_TIMEOUT
$z_1^2 :=$ $E$.Dec(ct$_{load}$; 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 $r \leq$ TRIGGER_TIMEOUT
$i_2 := i_2 + 1$, $z_3$, $i_2 := f_2(x_1, x_2, w_2, r)$, ct$_{trigger2,i_2} := E$.Enc($z_3$, $i_2$; sk), return (ct$_{trigger2,i_2}$, $i_2$, "ok")

On input* ("triggerfinish", $\sigma_7$, $\sigma_8$, $\sigma_9$, $\sigma_{i_2}$):
assert "triggerstart" has been called and returned (\(\cdot\), $i_2$, "ok")
assert "triggerfinish" has never previously returned (\(\cdot\), $i_2$, "ok")
assert $\forall$.Verify($\sigma_7$, $i_2$, (sid, eid$_2$, prog$_{SYX,2}$[$P_2$], (ct$_{trigger2,i_2}$, $i_2$, "ok")); $\varphi_{att}$)
assert $T$.Verify($\sigma_8$, $i_2$, $\parallel$ "trigger2" $\parallel$ sid $\parallel$ $i_2$ $\parallel$ ct$_{trigger2,i_2}$ $\parallel$ $\sigma_7$, $i_2$)
assert $t_{trigger2,i_2} \leq$ TRIGGER_TIMEOUT, return ($z_3$, $i_2$, $i_2$, "ok")

On input* ("triggerbyother", ct$_{trigger1,i_1 + 1}$, $\sigma_5$, $i_1 + 1$, $\sigma_6$, $i_1 + 1$, v):
if $v \neq \bot$, $i_1 := i_1 + 1$, 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 $\forall$.Verify($\sigma_5$, $i_1 + 1$, (sid, eid$_1$, prog$_{SYX,1}$[$P_1$], (ct$_{trigger1,i_1 + 1}$, $i_1 + 1$, "ok")); $\varphi_{att}$), and
$T$.Verify($\sigma_6$, $i_1 + 1$, $\parallel$ "trigger1" $\parallel$ sid $\parallel$ $(i_1 + 1)$ $\parallel$ ct$_{trigger1,i_1 + 1}$ $\parallel$ $\sigma_5$, $i_1 + 1$)
assert $t_{trigger1,i_1 + 1} \leq$ TRIGGER_TIMEOUT
$z_{2,i_1 + 1} := E$.Dec(ct$_{trigger1,i_1 + 1}$; 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 $P_2$.  

probsyx$[P_2]$
Prot_{SyX,1}[sid, P_1]

inp₁ := (sid, INPUT_TIMEOUT, TRIGGER_TIMEOUT, x₁, f = (f₁, f₂, φ₁, φ₂))

Initialization

eid₁ := G_{att}.install(sid, prog_{SyX,1}[P₁])

henceforth denote G_{att}.resume(·) := G_{att}.resume(eid₁, ·)

(g₁, σ₁) := G_{att}.resume("keyex")

send (eid₁, g₁, σ₁) to P₂, await (eid₂, g₂, σ₂)

assert \text{Verify}(σ₂, (sid, eid₂, prog_{SyX,2}[P₂], g₂); \text{vk}_{att})

Load Phase

(ct₁, ·) := G_{att}.resume("send", g₂, inp₁), send ct₁ to P₂, await ct₂

(ctload, "ok", σ₃) := G_{att}.resume("loadstart", ct₂)

(tload, σ₄) := F_{BB}.post("load" || sid || ctload || σ₃)

(z₁, "ok", ·) := G_{att}.resume("loadfinish", σ₃, tload, σ₄, ⊥), set i₁ := 0, i₂ := 0

Trigger Phase

Triggers by P₁

\begin{align*}
i₁ := i₁ + 1 & \quad \{ \text{cttrigger}_1, i₁, "ok", σ_{5,i₁} := G_{att}.resume("triggerstart", w₁) \} \\
(ttrigger₁, i₁, σ₅₁) := F_{BB}.post("trigger₁" || sid || cttrigger₁ || σ₅₁) & \quad \{ \text{z₂}, i₁, "ok", · := G_{att}.resume("triggerfinish", σ₅₁, ttrigger₁ || σ₆₁) \}
\end{align*}

Triggers by P₂

\begin{align*}
\{ \text{obtain a } t_{trigger₂, i₂+1} ≤ \text{TRIGGER_TIMEOUT such that} \} & \quad \{ "trigger₂" || sid || (i₂+1) || cttrigger₂, i₂+1 || σ₇,i₂+1 || σ₈,i₂+1 := F_{BB}.get-content(t_{trigger₂, i₂+1}) \} \\
(z₃, i₂+1, i₂ + 1, "ok", · := G_{att}.resume("triggerbyother", cttrigger₂, i₂+1, σ₇,i₂+1, t_{trigger₂, i₂+1}, σ₈,i₂+1) & \quad i₂ := i₂ + 1
\end{align*}

Figure 11: Protocol executed by Party P₁ in realizing F_{SyX}.
\[
\text{Prot}_{\text{SyX,}2}[\text{sid}, \text{P}_2]
\]

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

Initialization

\[
\text{eid}_2 := \text{G}_{\text{att}}.\text{install}(\text{sid}, \text{progs}_{\text{SyX,}2}[\text{P}_2])
\]

henceforth denote \(\text{G}_{\text{att}}.\text{resume}(\cdot) := \text{G}_{\text{att}}.\text{resume}(\text{eid}_2, \cdot)\)

(\(g^{y_2}, \sigma_2\) := \(\text{G}_{\text{att}}.\text{resume}(\text{keyex})\))

send (\(\text{eid}_2, g^{y_2}, \sigma_2\)) to \(\text{P}_1\), await (\(\text{eid}_1, g^{y_1}, \sigma_1\))

assert \(\forall.\text{Verify}(\sigma_1, (\text{sid}, \text{eid}_1, \text{progs}_{\text{SyX,}1}[\text{P}_1], g^{y_1}); \text{vk}_{\text{att}})\)

Load Phase

Await \(\text{ct}_1\)

(\(\text{ct}_2, \cdot\) := \(\text{G}_{\text{att}}.\text{resume}(\text{loadstart}, \text{inp}_2, \text{ct}_1)\))

obtain a \(t_{\text{load}} \leq \text{INPUT\_TIMEOUT}\) such that

(\(\text{ct}_{\text{load}}, \cdot\) := \(\text{F}_{\text{BB}}.\text{getContent}(t_{\text{load}})\))

(\(z^1_2, \cdot\) := \(\text{G}_{\text{att}}.\text{resume}(\text{loadfinish}, \text{ct}_{\text{load}}, \sigma_3, t_{\text{load}}, \sigma_4, \perp)\), set \(i_1 := 0, i_2 := 0\)

Trigger Phase

Triggers by \(\text{P}_2\)

\[
\begin{align*}
\begin{cases}
  i_2 &:= i_2 + 1 \\
  (\text{ct}_{\text{trigger}2, i_2}, i_2, \cdot) &:= \text{G}_{\text{att}}.\text{resume}(\text{triggerstart}, w_2) \\
  (\text{t}_{\text{trigger}2, i_2}, i_2) &:= \text{F}_{\text{BB}}.\text{post}(\text{trigger2} \parallel \text{sid} \parallel \text{ct}_{\text{trigger}2, i_2} \parallel \sigma_7, i_2) \\
  (z_{3, i_2}, i_2, \cdot) &:= \text{G}_{\text{att}}.\text{resume}(\text{triggerfinish}, \sigma_7, i_2, \text{t}_{\text{trigger}2, i_2}, \sigma_8, i_2)
  \end{cases}
\end{align*}
\]

Triggers by \(\text{P}_1\)

\[
\begin{align*}
\begin{cases}
  (\text{t}_{\text{trigger}1, i_1}, i_1, \cdot) &:= \text{F}_{\text{BB}}.\text{getContent}(t_{\text{trigger}1, i_1}) \\
  (z_{2, i_1}, i_1 + 1, \cdot) &:= \text{G}_{\text{att}}.\text{resume}(\text{triggerbyother}, \text{ct}_{\text{trigger}1, i_1}, \sigma_5, i_1, t_{\text{trigger}1, i_1}, \sigma_6, i_1, \perp) \\
  i_1 &:= i_1 + 1
  \end{cases}
\end{align*}
\]

Figure 12: Protocol executed by Party \(\text{P}_2\) in realizing \(\mathcal{F}_{\text{SyX}}\).
• At this point, the load phase of $\mathcal{F}_{SyX}$ has been completed. $S$ sends $\text{inp}_2^{19}$ to the trusted party computing $\mathcal{F}_{SyX}$ with guaranteed output delivery. It receives the corrupt party’s output for the load phase, namely, $z_1^2$.

• $S$ calls $(\text{ct}_{\text{load}}, \text{"ok"}, \sigma_3) := \mathcal{G}_{\text{att}}.\text{resume}(\text{eid}_1, (\text{"loadstart"}, \text{ct}_2))$ and $S$ then calls $(\text{t}_{\text{load}}, \sigma_4) := \mathcal{F}_{\text{BB}}.\text{post}(\text{"load"} || \text{sid} || \text{ct}_{\text{load}} || \sigma_3)$.

• When $S$ receives $\mathcal{G}_{\text{att}}.\text{resume}(\text{eid}_2, (\text{"loadfinish"}, \text{ct}_{\text{load}}, \sigma_3, \text{t}_{\text{load}}, \sigma_4, v))$ from $P_2$, if $v \neq \bot$, pass through the call. Else, when $S$ first receives such a call with $v = \bot$, $S$ calls $(z_1^2, \text{"ok"}, \sigma_{\text{sim},1}) := \mathcal{G}_{\text{att}}.\text{resume}(\text{eid}_1, (\text{"loadfinish"}, \sigma_3, \text{t}_{\text{load}}, \sigma_4, z_1^2))$ and returns $(z_1^2, \text{"ok"}, \sigma_{\text{sim},1})$.

Hybrid 0. Identical to the simulation, except that every occurrence of the challenge $sk = g^{y_1y_2}$ 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 $\mathcal{E}$.

---

\textsuperscript{19}We note that the format of $\text{inp}_2 := (\text{sid}', \text{INPUT\_TIMEOUT}', \text{TRIGGER\_TIMEOUT}', x_2, f')$ is different from the input format that the ideal functionality $\mathcal{F}_{SyX}$ described in Figure 5 expects to receive. In particular, the differences are that there is a new variable representing the session identifier $\text{sid}$ and that the round numbers corresponding to the time outs $\text{INPUT\_TIMEOUT}', \text{TRIGGER\_TIMEOUT}'$ 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 $\text{inp}_2 := (\text{sid}', \text{INPUT\_TIMEOUT}', \text{TRIGGER\_TIMEOUT}', x_2, f')$. We omit the details for simplicity here.
Hybrid 3. Identical to Hybrid 2, except that the challenge \(sk\) is now replaced with the true \(g^{y_1y_2}\) 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 \(\pi\) which securely computes \(F_{\text{MPC}}\) with fairness in the presence of \(t\)-threshold adversaries for any \(0 \leq t < n\) in the \((G_{\text{att}}, F_{\text{BB}})\)-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 \((G_{\text{att}}, F_{\text{BB}})\)-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 \(\phi_1 \equiv \phi_2 \equiv 0\), then, the functionality is equivalent to the ideal functionality \(F_{\text{2PC}}\) (\(F_{\text{MPC}}\) 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 \(F_{\text{SyX}}\) 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 \(F_{\text{SyX}}\). We can thus model the ability to trigger an instance of the ideal functionality \(F_{\text{SyX}}\) 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 \((G_{\text{att}}, F_{\text{BB}})\)-hybrid model.
5.1 Synchronizable Exchange with One-Sided Triggers in the \((G_{\text{att}}, G_{\text{acrs}}, F_{BB})\)-Hybrid Model

We show how the ideal functionality for synchronizable exchange \(F_{\text{SyX}}\) with one-sided trigger can be realized in the \((G_{\text{att}}, G_{\text{acrs}}, F_{BB})\)-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 \(F_{\text{SyX}}\). The party that obtains this ciphertext then publishes it using the ideal functionality \(F_{BB}\) 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 \(F_{BB}\) 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 \(F_{BB}\) and decrypts to obtain the output of the load phase.

When the party with the processor wishes to trigger \(F_{\text{SyX}}\), 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 \(F_{\text{SyX}}\). The party then publishes the ciphertext using the ideal functionality \(F_{BB}\) 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 \(F_{BB}\) 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, \(E_1\) is perfect correct, and satisfies semantic security, \(E_2\) is perfectly correct, and satisfies semantic security and INT-CTXT security, the proof system satisfies computational soundness and witness indistinguishability, and that \(T\) and \(V\) are existentially unforgeable, it holds that the protocol described in Figures 13, 14 and 15 securely realizes \(F_{\text{SyX}}\) 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 \(F_{\text{MPC}}\) can be realized in the \((G_{\text{att}}, F_{BB})\)-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 \(F_{\text{MPC}}\) in the \(F_{\text{SyX}}\)-hybrid model where only \(t\) of the parties can trigger an instance of the ideal functionality \(F_{\text{SyX}}\). 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
prog\text{one-sided-SyX}.1[G_{acrs}.mpk, P_1]

On input ("init"): \((pk, sk) \xleftarrow{\$} E_1.\text{Gen}(1^\lambda), \text{return } pk\)

On input ("extract", idk):
if check\((G_{acrs}.mpk, P_2, idk) = 1, v := sk, \text{else } v := \bot, \text{return } v\)

On input ("loadstart", ct_2): \((\text{inp}_2, k) := E_1.\text{Dec}(ct_2; sk), \text{return } ct_2\)

On input ("loadcontinue", ct_continue, inp_1):
assert "loadstart" has been called
parse \(\text{inp}_1 := (\text{sid}, \text{INPUT\_TIMEOUT}, \text{TRIGGER\_TIMEOUT}, x_1, f)\)
parse \(\text{inp}_2 := (\text{sid}', \text{INPUT\_TIMEOUT}', \text{TRIGGER\_TIMEOUT}', x_2, f')\)
assert that:
\(\text{sid} = \text{sid}', f = f'\)
\(\text{INPUT\_TIMEOUT} = \text{INPUT\_TIMEOUT}'\)
\(\text{TRIGGER\_TIMEOUT} = \text{TRIGGER\_TIMEOUT}'\)
assert \(r \leq \text{INPUT\_TIMEOUT}\), parse \(f = (f_1, f_2, \phi_1, \phi_2)\)
\(z_1 = f_1(x_1, x_2, r), \text{ct}_{\text{load}} := E_2.\text{Enc}(z_1; k), \text{return } (\text{ct}_{\text{load}}, \text{"ok"})\)

On input ("loadfinish", \(\sigma_3, t_{\text{load}}, \sigma_4\)):
assert "loadcontinue" has been called and returned (\(\bot, \text{"ok"}\))
assert \(\mathcal{V}.\text{Verify}(\sigma_3, (\text{sid}, \text{eid}_1, \text{prog}\text{one-sided-SyX}.1[P_1], (\text{ct}_{\text{load}}, \text{"ok"})); \text{vk}_{\text{att}})\)
assert \(\mathcal{T}.\text{Verify}(\sigma_4, t_{\text{load}} \parallel \text{"load"} \parallel \text{sid} \parallel \text{ct}_{\text{load}} \parallel \sigma_3)\)
assert \(t_{\text{load}} \leq \text{INPUT\_TIMEOUT}\), set \(i_1 := 0, \text{return } (z_1, \text{"ok"})\)

On input* ("triggerstart", \(w_1\)):
assert "loadfinish" has been called and returned (\(\bot, \text{"ok"}\))
assert \(i_1 = 0\) or "triggerfinish" has been called and returned (\(\bot, i_1 - 1, \text{"ok"}\))
and "triggerstart" has never previously returned (\(\bot, i_1, \text{"ok"}\))
assert \(\phi_1(w_1) = 1\) and that \(r \leq \text{TRIGGER\_TIMEOUT}\)
\(i_1 := i_1 + 1, z_{2,i_1} := f_2(x_1, x_2, w_1, r), \text{ct}_{\text{trigger}1,i_1} := E_2.\text{Enc}(z_{2,i_1}; k)\)
return \((\text{ct}_{\text{trigger}1,i_1}, i_1, \text{"ok"})\)

On input* ("triggerfinish", \(\sigma_{5,i_1}, t_{\text{trigger}1,i_1}, \sigma_{6,i_1}\)):
assert "triggerstart" has been called and returned (\(\bot, i_1, \text{"ok"}\))
assert "triggerfinish" has never previously returned (\(\bot, i_1, \text{"ok"}\))
assert \(\mathcal{V}.\text{Verify}(\sigma_{5,i_1}, (\text{sid}, \text{eid}_1, \text{prog}\text{SyX}.1[P_1], (\text{ct}_{\text{trigger}1,i_1}, i_1, \text{"ok"})); \text{vk}_{\text{att}})\)
assert \(\mathcal{T}.\text{Verify}(\sigma_{6,i_1}, t_{\text{trigger}1,i_1} \parallel \text{"trigger1"} \parallel \text{sid} \parallel i_1 \parallel \text{ct}_{\text{trigger}1,i_1} \parallel \sigma_{5,i_1})\)
assert \(t_{\text{trigger}1,i_1} \leq \text{TRIGGER\_TIMEOUT}\), return \((z_{2,i_1}, i_1, \text{"ok"})\)

Figure 13: Program installed in the secure hardware of Party \(P_1\).
\[
\text{Prot}_{\text{one-sided-SyX}, 1}[\text{sid}, \mathcal{G}_{\text{acs}}{\cdot}\text{mpk}, P_1]
\]

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

**Initialization**

\(\text{eid}_1 := \mathcal{G}_{\text{att}}{\cdot}\text{install}(\text{sid}, \text{prog}_{\text{one-sided-SyX}, 1}[P_1])\)

henceforth denote \(\mathcal{G}_{\text{att}}{\cdot}\text{resume}(\cdot) := \mathcal{G}_{\text{att}}{\cdot}\text{resume}(\text{eid}_1, \cdot)\)

\((pk, \sigma_1) := \mathcal{G}_{\text{att}}{\cdot}\text{resume}(\text{“init”})\)

send \((\text{eid}_1, \psi(P_2, pk, \sigma_1))\) to \(P_2\)

**Load Phase**

await \(ct_2\) from \(P_2\)

\((ct_2, \sigma_2) := \mathcal{G}_{\text{att}}{\cdot}\text{resume}(\text{“loadstart”}, ct_2)\)

send \(\psi(P_2, ct_2, \sigma_2)\) to \(P_2\), await \(ct_{\text{continue}}\) from \(P_2\)

\((ct_{\text{load}}, \text{“ok”}, \sigma_3) := \mathcal{G}_{\text{att}}{\cdot}\text{resume}(\text{“loadcontinue”}, ct_{\text{continue}}, \text{inp}_1)\)

\((t_{\text{load}}, \text{“ok”}, \sigma_4) := \mathcal{F}_{\text{BB}}{\cdot}\text{post}(\text{“load”}, \text{sid}\|ct_{\text{load}}\|\sigma_3)\)

\((z_1, \text{“ok”}, \cdot) := \mathcal{G}_{\text{att}}{\cdot}\text{resume}(\text{“loadfinish”}, \sigma_3, t_{\text{load}}, \sigma_4), \text{set } i_1 := 0\)

**Trigger Phase**

\(\text{Triggers by } P_1\)

\[
\begin{align*}
\{ & i_1 := i_1 + 1 \\
& (ct_{\text{trigger1}}, i_1, \text{“ok”}, \sigma_5, i_1) := \mathcal{G}_{\text{att}}{\cdot}\text{resume}(\text{“triggerstart”}, w_1) \\
& (t_{\text{trigger1}}, \sigma_6, i_1) := \mathcal{F}_{\text{BB}}{\cdot}\text{post}(\text{“trigger1”}, \text{sid}\|i_1\|ct_{\text{trigger1}}, \sigma_5, i_1) \\
& (z_{2, i_1}, i_1, \text{“ok”}, \cdot) := \mathcal{G}_{\text{att}}{\cdot}\text{resume}(\text{“triggerfinish”}, \sigma_5, t_{\text{trigger1}}, \sigma_6, i_1) \}
\end{align*}
\]

Figure 14: Protocol executed by Party \(P_1\) in realizing \(\mathcal{F}_{\text{SyX}}\) with One-Sided Trigger.
\[
\text{Prot}_{\text{one-sided-SyX}, 2}[\text{sid}, P_2]
\]

\[
in_{P_2} := (\text{sid}, \text{INPUT\_TIMEOUT}, \text{TRIGGER\_TIMEOUT}, x_2, f = (f_1, f_2, \phi_1, \phi_2))
\]

**Load Phase**

\[
\text{await (eid}_1, \psi) \text{ from } P_1
\]

// Henceforth for \( \psi := (\text{msg}, C, \pi) \), let \( \text{Ver}(\psi) := \text{Ver}(\text{crs}, (\text{sid}, \text{eid}_1, C, \text{mpk}, G_{\text{accs}.\text{mpk}}, P_2, \text{msg}, \pi)) \)

assert \( \text{Ver}(\psi) \), parse \( \psi := (\text{pk}, \text{msg}, 0) \)

\[
k \leftarrow \{0, 1\} \lambda, \text{ct}_2 = \mathcal{E}_1.\text{Enc}((\text{inp}_2, k); \text{pk}), \text{send ct}_2 \text{ to } P_1
\]

\[
\text{assert ct}_2 \text{ to } P_1, \text{assert \( \text{Ver}(\psi) \), parse \( \psi := (\text{ct}_2, \ldots) \)}
\]

\[
\text{ct}_\text{continue} \leftarrow \mathcal{E}_2.\text{Enc}(\text{"continue"}), \text{send ct}_\text{continue} \text{ to } P_1
\]

\[
\text{obtain a } t_{\text{load}} \leq \text{INPUT\_TIMEOUT} \text{ such that}
\]

\[
\text{assert } V.\text{Verify}(\sigma_3, (\text{sid}, \text{eid}_1, \text{prog}_{\text{one-sided-SyX}, 1}[P_1], (\text{ct}_\text{load}, \text{"ok"}); \text{vk}_\text{att})
\]

\[
\text{assert } T.\text{Verify}(\sigma_4, t_{\text{load}} || \text{"load"}\text{||\text{sid}\text{||ct}_\text{load}\text{||\sigma}_3})
\]

\[
\text{assert } t_{\text{load}} \leq \text{INPUT\_TIMEOUT}
\]

\[
z_1 := \mathcal{E}_2.\text{Dec}(\text{ct}_\text{load}; k), \text{assert that decryption succeeds, set } i_1 := 0
\]

**Trigger Phase**

**Triggers by P₁**

\[
\begin{align*}
\text{obtain a } t_{\text{trigger}, i_1, 1} \leq \text{TRIGGER\_TIMEOUT} \text{ such that} \\
(\text{"trigger1" || \text{sid}||\text{ct}_{\text{trigger}, i_1, 1}||\sigma_{5, i_1, 1}||\sigma_{6, i_1, 1}) := & \\
\mathcal{F}_{\text{BB}, \text{getContent}}(t_{\text{trigger}, i_1, 1}) \\
\text{assert } V.\text{Verify}(\sigma_{5, i_1, 1}, (\text{sid}, \text{eid}_1, \text{prog}_{\text{one-sided-SyX}, 1}[P_1], (\text{ct}_{\text{trigger}, i_1, 1, 1}, i_1 + 1, \text{"ok"}); \text{vk}_\text{att})
\]

\[
\text{assert } T.\text{Verify}(\sigma_{6, i_1, 1}, t_{\text{trigger}, i_1, 1, 1} || \text{"trigger1" || \text{sid}||\text{ct}_{\text{trigger}, i_1, 1, 1}||\sigma_{5, i_1, 1})
\]

\[
\text{assert } t_{\text{trigger}, i_1, 1, 1} \leq \text{TRIGGER\_TIMEOUT}
\]

\[
z_{2, i_1, 1} := \mathcal{E}_2.\text{Dec}(\text{ct}_{\text{trigger}, i_1, 1, 1}; k), \text{assert that decryption succeeds}
\]

\[
i_1 := i_1 + 1
\]

Figure 15: Protocol executed by Party \( P_2 \) in realizing \( F_{\text{SyX}} \) with One-Sided Trigger.
parties $P_i$ and $P_j$ have access to the ideal functionality $F_{\text{SyX}}$. In particular, let $F_{\text{SyX}}^{i,j}$ represent the instantiation of the $F_{\text{SyX}}$ functionality used by parties $P_i, P_j$. Furthermore, only parties $P_1$ and $P_2$ may trigger the instances of $F_{\text{SyX}}$ 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 $(y_1, y_2, y_3) \leftarrow F(x_1, x_2, x_3)$ 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 $|z'_i| = |z_i|$ 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 $z_i$s 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 $F_{\text{SyX}}$.** We assume that the secure function evaluation also provides commitments to all the shares of the output. That is, $P_i$ receives $(y_i, \overline{c})$ for each $i \in [3]$, where $\overline{c}$ is a commitment scheme and

$$\overline{c} = \{\text{Com}(y_1), \text{Com}(y_2), \text{Com}(y_3)\}$$

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 $\text{Sign}$ and a verification algorithm $\text{Verify}$, for each $i \in [3]$. All parties then broadcast their verification keys to all parties. Let

$$\overrightarrow{vk} = \{vk_1, vk_2, vk_3\}$$

---

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 $\mathcal{F}_{SyX}^{i,j}$ with inputs

$$x_i = (v_k, sk_i, y_i, \overline{c})$$

and

$$x_j = (v_k, sk_j, y_j, \overline{c})$$

The function $f_1$ checks if both parties provided the same value for $v_k$, $\overline{c}$ 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 $\mathcal{F}_{SyX}^{i,j}$ computes

$$\sigma_{i,j} = \text{Sign}((i, j); sk_i) \| \text{Sign}((i, j); sk_j)$$

This completes the description of $f_1$.

**Synchronization step.** The output of $f_1$ for each of the $\mathcal{F}_{SyX}^{i,j}$ will provide a way to synchronize all $\mathcal{F}_{SyX}$ instances. By synchronization, we mean that an $\mathcal{F}_{SyX}^{i,j}$ instance cannot be triggered unless every other instance has already completed its input phase successfully. We will have the instance $\mathcal{F}_{SyX}$ 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 $\phi$. The protocol proceeds in two rounds:

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

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

**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 $\mathcal{F}_{SyX}$ instances is triggered. Furthermore, note that this implies that the corrupt parties must successfully complete the input phases of the instances of $\mathcal{F}_{SyX}$ that it shares with all of the honest parties in order to obtain the witness that can be used to trigger the $\mathcal{F}_{SyX}$ instances. Following the input phases of all of the $\mathcal{F}_{SyX}$ instances, we ask each party to broadcast the receipt $\sigma_{i,j}$ obtained from $\mathcal{F}_{SyX}^{i,j}$. 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 $\mathcal{F}_{SyX}$ to compute $\sigma_{i,j}$. Since honest $P_k$ does not know the synchronizing witness $\overline{\sigma}$, it will not be able to trigger any of the $\mathcal{F}_{SyX}$ 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 $\mathcal{F}_{SyX}^{i,k}$ to obtain $P_k$’s share of the key. However, once $P_i$ triggers $\mathcal{F}_{SyX}^{i,k}$, it follows that $P_k$ would obtain the synchronizing witness $\overline{\sigma}$ along with some shares of the output. If $k = 1$ or $k = 2$, then $P_k$ obtains $(\overline{\sigma}, y_1, y_2)$ at the end of round 1 and can successfully trigger $\mathcal{F}_{SyX}^{1,3}$ or $\mathcal{F}_{SyX}^{2,3}$ in round 2 to obtain the final share $y_3$ of the output. If $k = 3$, then $P_3$ obtains $(\overline{\sigma}, y_1, y_2, y_3)$ 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 $F_{SyX}^{i,j}$. In particular, let $F_{SyX}^{i,j}$ represent the instantiation of the $F_{SyX}$ functionality used by parties $P_i, P_j$. Furthermore, only parties $P_1, \ldots, P_i$ may trigger the instances of $F_{SyX}$ 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 $F_{SyX}^{i,j}$ will provide a way to synchronize all $F_{SyX}$ instances. By synchronization, we mean that an $F_{SyX}^{i,j}$ instance cannot be triggered unless every other instance has already completed its input phase successfully. We will have the instance $F_{SyX}^{i,j}$ 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 $\phi$. 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 $F_{SyX}^{i,j,r+1}$ accepts a trigger $(\sigma, y_1, \ldots, y_r)$ from $P_i$. If $P_i$ provides this trigger, then both $P_i$ and $P_{r+1}$ receive $(\sigma, y_1, \ldots, y_{r+1})$.

Consider, however, the case of $n = 4$ and $t = 2$. Since $P_3$ does not have the ability to trigger an instance of $F_{SyX}$, 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 $F_{SyX}$ instances is triggered. Furthermore, note that this implies that the corrupt parties must successfully complete the input phases of the instances of $F_{SyX}$ that it shares with all of the honest parties in order to obtain the witness that can be used to trigger the $F_{SyX}$ instances. Following the input phases of all of the $F_{SyX}$ instances, we ask each party to broadcast the receipt $\sigma_{i,j}$ obtained from $F_{SyX}^{i,j}$. 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 $F_{SyX}$ 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 $F_{SyX}$ 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 $F_{SyX}$. 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 $F_{SyX}$. This in particular implies that
The protocol $\Pi$ for fair secure computation in the $(F_{bc}, F_{MPC}, F_{SyX})$-hybrid model.

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

**Protocol.** The protocol $\Pi_{\text{F MPC}}$ proceeds as follows:

- Define $F'$ to 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
    $$y = y_1 \parallel \cdots \parallel y_n$$
  - Sample random strings $\alpha_i \xleftarrow{\$} \{0,1\}^*$ such that $|\alpha_i| = |y_i|$ for each $i \in [n]$. Let
    $$\alpha = \alpha_1 \parallel \cdots \parallel \alpha_n$$
  - Let $z = y \oplus \alpha$. 

6.3 Protocol

We now present the protocol for fair secure computation in the $(F_{bc}, F_{MPC}, F_{SyX})$-hybrid model.

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 $P_\ell$ for some $\ell \in [\min\{t, j - 1\}]$ that triggered the channel $F_{SyX}^{\ell,j}$ that involved $P_j$. Then, at the end of round $j - 1$, party $P_\ell$ learns $(\overrightarrow{\sigma}, y_1, \ldots, y_j)$. If $j \leq t$, 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 $F_{SyX}^{\ell,m}$ in round $m - 1$ for every $j + 1 \leq m \leq n$. In this case, $k = j$ and we are done. Notice that this means that if there exists an honest party $P_j$ with $j \leq t$, 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 $F_{SyX}$ that involved $P_n$. This, by the definition, means that no party triggered an instance of $F_{SyX}$ 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 $F_{SyX}$ 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 $P_j$ learns $y_n$ at the end of the protocol, it must be the case that there exists a (corrupt) party $P_\ell$ for some $\ell \in [t]$ that triggered the channel $F_{SyX}^{\ell,n}$ 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.
– 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 $(c_i^z, \omega_i^z) \overset{\$}{\leftarrow} \text{Com}(z_i)$ to each of the shares $z_i$ for each $i \in [n]$. Let
\[
c^z = (c_1^z, \ldots, c_n^z)
\]

– Party $P_i$ receives output $(\alpha_i, c_i^z, \omega_i^z, z_i)$ for each $i \in [n]$.

• The parties invoke the ideal functionality $F_{\text{MPC}}$ with inputs $((x_1, F'), \ldots, (x_n, F'))$. If the ideal functionality returns $\bot$ to party $P_i$, then $P_i$ aborts for any $i \in [n]$\(^\text{21}\). Otherwise, party $P_i$ receives output $(\alpha_i, c_i^z, \omega_i^z, z_i)$ for each $i \in [n]$.

• Each party $P_i$, for each $i \in [n]$, picks a random $\beta_i \in \{0, 1\}^n$ and uses this randomness to pick a signing and verification key pair $(sk_i, vk_i) = \mathcal{V}.\text{Gen}(1^{\lambda}; \beta_i)$. It then invokes the ideal functionality $F_{\text{bc}}$ 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
\[
vk = (vk_1, \ldots, vk_n)
\]

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

  – Let $f_{a,b}^{\gamma, \gamma'}$ be the function that takes as input $(\gamma, \gamma')$ and parses
\[
\gamma = (\overrightarrow{vk}, sk, \beta, c^z, \omega^z, z)
\]
and
\[
\gamma' = (\overrightarrow{vk'}, sk', \beta', c'^{z'}, \omega'^{z'}, z')
\]

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

  If all of these checks pass, then $f_{a,b}^{\gamma, \gamma'}$ outputs
\[
\sigma_{a,b} = (\mathcal{V}.\text{Sign}((a, b); sk_a), \mathcal{V}.\text{Sign}((a, b); sk_b))
\]
and otherwise it outputs $\bot$.

\(^{21}\)In the $F_{\text{OT}}$-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 $F_{\text{MPC}}$-hybrid model, we know that in the ideal model, this is the case when the honest parties receive $\bot$ as their output. If we assume that in the case when the adversary decides to let the honest parties obtain their outputs, no honest party ever receives $\bot$, 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 $\bot$ 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.

42
Let $\phi_{a,b}^1$ be the function that takes as input a witness $w$, which is of the form $(\vec{a}, \vec{z}, \omega^z)$. 

* 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 $a < b$,
    
    \[
    \forall. \text{Verify} (\sigma_{a,b,1}, (a,b); vk_a) = 1 
    \]

    and
    
    \[
    \forall. \text{Verify} (\sigma_{a,b,2}, (a,b); vk_b) = 1 
    \]

    · $|\vec{z}| = |\vec{\omega}| = r$
    · $\text{Open} (c^j_z, \omega^z_j, z_j) = 1$ for every $j \in [r]$.

Let $\phi_{a,b}^2$ be the function that takes as input a witness $w$, which is of the form $(\vec{a}, \vec{z}, \omega^z)$. 

* 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 $a < b$,
    
    \[
    \forall. \text{Verify} (\sigma_{a,b,1}, (a,b); vk_a) = 1 
    \]

    and
    
    \[
    \forall. \text{Verify} (\sigma_{a,b,2}, (a,b); vk_b) = 1 
    \]

    · $|\vec{z}| = |\vec{\omega}| = r$
    · $\text{Open} (c^j_z, \omega^z_j, z_j) = 1$ for every $j \in [r]$.

Let $f_{a,b}^2$ be the function that takes as input $(\gamma, \gamma')$ where $\gamma, \gamma'$ are as above, and outputs $(\omega^z, z, \omega^z', z')$.

- 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 $F_{SyX}$ that it is involved in. For each pair of parties $P_a, P_b$ with $a \neq b$ for $a,b \in [n]$, let $a' = \min(a,b)$ and $b' = \max(a,b)$. For each such pair of parties $P_a, P_b$, party $P_a$ runs the input phase of $F_{a',b'}^{SyX}$, providing inputs $(x_a, f)$, where

  \[
  x_a = (\vec{vk}, \text{sk}_a, \beta_a, c^z, \omega^z_a, z_a) 
  \]

  and

  \[
  f = (f_{a',b'}^1, f_{a',b'}^a, \phi_{a',b'}^1, \phi_{a',b'}^2) 
  \]

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

---

22This 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 $\mathcal{F}_{\text{SyX}}^{a',b'}$ it is involved in, indicating that the input phase of all such $\mathcal{F}_{\text{SyX}}$ functionalities were completed successfully, and $r = 0$, it invokes the ideal functionality $F_{bc}$ and broadcasts 

$$\overline{\sigma}_a = \{\sigma_{a', b'}\}_{a' = a \lor b = a}$$

to all the parties. Otherwise, it invokes the ideal functionality $F_{bc}$ 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 $(\overrightarrow{\sigma}, \overrightarrow{z}, \overrightarrow{\omega})$ such that

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

$$\forall \text{Verify} (\sigma_{a, b, 1}, (a, b); vk_a) = 1$$

and

$$\forall \text{Verify} (\sigma_{a, b, 2}, (a, b); vk_b) = 1$$

* $|\overrightarrow{z}| = |\overrightarrow{\omega}| = r$

* Open $(c^i_j, \omega^i_j, z_j) = 1$ for every $j \in [r]$.

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

- For $i \in [n]$, if at the end of round $r = n - 1$, party $P_i$ has obtained values $(\overrightarrow{z}, \overrightarrow{\omega})$ such that

* $|\overrightarrow{z}| = |\overrightarrow{\omega}| = n$

* Open $(c^i_j, \omega^i_j, z_j) = 1$ for every $j \in [n]$.

then, $P_i$ invokes the ideal functionality $F_{bc}$ in round $r + 1 = n$ and broadcasts $(\overrightarrow{z}, \overrightarrow{\omega})$ to all the parties.

- For $i \in [n]$, if at the end of round $r = n$, party $P_i$ has obtained values $(\overrightarrow{z}, \overrightarrow{\omega})$ such that

* $|\overrightarrow{z}| = |\overrightarrow{\omega}| = n$

* Open $(c^i_j, \omega^i_j, z_j) = 1$ 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 $O(n^2)$ signatures with $n$ other commitments to $n$ other independent random proof values that can be used to prove that all the instances of $\mathcal{F}_{\text{SyX}}$ 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 $\mathcal{F}_{\text{MPC}}$. Furthermore, we have that at the end of the invocation of the ideal functionality $\mathcal{F}_{\text{MPC}}$, either all honest parties unanimously abort or all honest parties unanimously continue. Thus, assuming that $\mathcal{F}_{\text{MPC}}$ did not abort, every party receives the output of $F'$. For every $i \in [n]$, let $vk_i$ denote the set of verification keys that were obtained by party $P_i$. Note that, by the correctness of the ideal functionality $\mathcal{F}_{\text{bc}},$

$$\overrightarrow{vk} = \overrightarrow{vk_i}$$

for all $i \in [n]$. If $\overrightarrow{vk}$ 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 $\mathcal{F}_{\text{SyX}}$ 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 $\mathcal{F}_{\text{SyX}}$ and the binding property for the honestly generated commitments, that particular instance of $\mathcal{F}_{\text{SyX}}$ 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 $\mathcal{F}_{\text{SyX}}$ 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 $\mathcal{F}_{\text{SyX}}$ 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 $\mathcal{F}_{\text{SyX}}$ 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 $\mathcal{F}_{\text{SyX}}$ that involved $P_n$. This, by the definition, means that no party triggered an instance of $\mathcal{F}_{\text{SyX}}$ 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 $\mathcal{F}_{\text{SyX}}$ 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 $\mathcal{F}_{\text{SyX}}$ 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 $(\text{Com, Open, } \widetilde{\text{Com, Open}})$ is an honest-binding commitment scheme and $\mathcal{V}$ is a signature scheme, then the protocol $\Pi_{\text{F MPC}}$ securely computes $\mathcal{F}_{\text{MPC}}$ with fairness in the $(\mathcal{F}_{\text{bc}}, \mathcal{F}_{\text{MPC}}, \mathcal{F}_{\text{SyX}})$-hybrid model.
Proof. Let \( \mathcal{A} \) be an adversary attaching the execution of the protocol described in Section 6.3 in the \((\mathcal{F}_{bc}, \mathcal{F}_{MPC}, \mathcal{F}_{SyX})\)-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 \( \mathcal{F}_{MPC} \). Here, \( \mathcal{S} \) behaves as the ideal functionality \( \mathcal{F}_{MPC} \). Recall that the type of \( \mathcal{F}_{MPC} \) is abort. \( \mathcal{S} \) obtains the inputs \( \{(x_i, f_i)\}_{i \in \mathcal{I}} \) of the corrupted parties from \( \mathcal{A} \). If \( (x_i, f_i) = \text{abort} \) for any \( i \in \mathcal{I} \), \( \mathcal{S} \) forwards \( \{(x_i, f_i)\}_{i \in \mathcal{I}} \) to the trusted party computing \( \mathcal{F}_{MPC} \) with fairness, receives \( \bot \) as the output of all parties, which it forwards \( \mathcal{A} \). Suppose \( (x_i, f_i) \neq \text{abort} \) for all \( i \in \mathcal{I} \). If there exists a \( j \in \mathcal{I} \) such that \( f_j \neq F' \) as defined in protocol \( \Pi_{FMPC} \), \( \mathcal{S} \) forwards \( \{(x_i, f_i)\}_{i \in \mathcal{I}} \) to the trusted party computing \( \mathcal{F}_{MPC} \) 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 \( \Pi_{FMPC} \).

\( \mathcal{S} \) now needs to simulate the outputs received by the corrupted parties from the ideal functionality \( \mathcal{F}_{MPC} \). For each \( i \in [n] \), \( \mathcal{S} \) samples a random string \( \alpha_i \xleftarrow{\$} \{0, 1\}^* \) of length equal to the length of the \( i \)th 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 \( (c_i^z, \omega_i^z) \xleftarrow{\$} \text{Com}(z_i) \) to each of the shares \( z_i \) for each \( i \in \mathcal{I} \). For \( i = h \), it samples an equivocable commitment \( (c_h^z, \text{state}_h) \xleftarrow{\$} \text{Com}(1^\lambda) \). 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 \text{abort}, \( \mathcal{S} \) forwards \( \{(x_i, f_i)\}_{i \in \mathcal{I}} \) to the trusted party computing \( \mathcal{F}_{MPC} \) with fairness, with \( (x_j, f_j) \) replaced with \text{abort} for some \( j \in \mathcal{I} \), receives \( \bot \) 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 \( \mathcal{F}_{MPC} \).

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

\[
\overrightarrow{\text{vk}} = (\text{vk}_1, \ldots, \text{vk}_n)
\]

At this point, \( \mathcal{S} \) has completed simulating the invocations of the ideal functionality \( \mathcal{F}_{bc} \) 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 \( \mathcal{F}_{SyX} \) that involve corrupt
parties. Here, \( S \) behaves as the ideal functionality \( F_{SyX} \). Recall that the type of \( F_{SyX} \) 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 \( A \) instructs \( P_a \) to invoke the input phase of \( F_{SyX}^{a,b} \) with inputs

\[
\gamma = (\vec{v}', \sk_a, \beta_a, \vec{c}^t$, $\omega^z, z)
\]

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

\[
\gamma' = (\vec{v}k, \sk_b, \beta_b, \vec{c}^t$, $\omega^z_b, z_b)
\]

Note that since \( b \in [n] \setminus \mathcal{I} \), \( S \) does in fact have \( \sk_b, \beta_b \). The only values it does not have are \( \omega^z_h, z_h \).

In the execution of \( f_1^{a,b}, \omega^z_b, z_b \) are needed to check that

\[
\text{Open}(\omega^z_h, z_h) = 1
\]

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 \( f_1^{a,b} \), \( S \) performs all the checks that \( f_1^{a,b} \), except this one. If all the checks pass, \( S \) computes

\[
\sigma_{a,b} = (\text{V.Sign}((a, b), \sk_a), \text{V.Sign}((a, b), \sk_b))
\]

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

\[
\gamma = (\vec{v}', \sk_a, \beta_a, \vec{c}^t$, $\omega^z, z)
\]

and

\[
\gamma' = (\vec{v}k', \sk_b, \beta_b, \vec{c}^t$, $\omega^z', z')
\]

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

\[
\sigma_{a,b} = (\text{V.Sign}((a, b), \sk_a), \text{V.Sign}((a, b), \sk_b))
\]

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

\[
\vec{sk} = (sk_1, \ldots, sk_n)
\]
and
\[ \beta = (\beta_1, \ldots, \beta_n) \]

\( \mathcal{S} \) computes
\[ \sigma_{a,b} = (\mathcal{V}.\text{Sign}((a, b); sk_a), \mathcal{V}.\text{Sign}((a, b); sk_b)) \]
for every \( a < b \in [n] \) and defines
\[ \overrightarrow{\sigma} = \{ \sigma_{a,b} \}_{a \neq b, a, b \in [n]} \]

Now, \( \mathcal{S} \) must simulate the invocations of the ideal functionality \( F_{bc} \) by the corrupt parties. For all \( a \in [n] \setminus \mathcal{I} \), \( \mathcal{S} \) “broadcasts”
\[ \overrightarrow{\sigma}_i = \{ \sigma_{a,b'} \}_{a \equiv i \vee b' \equiv i} \]
to all the corrupt parties. For any \( i \in \mathcal{I} \), if \( \mathcal{A} \) instructs \( P_i \) to invoke \( F_{bc} \) 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 \( F_{SyX} \) and the ideal functionality \( F_{bc} \). 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 \( F_{SyX} \) that the adversary instructs corrupt parties to trigger. We consider two cases. First, we make the following definition: a witness \( w \) is \textit{valid for round} \( r \) if
\[ w = (\overrightarrow{\sigma}, \overrightarrow{\tau}, \overrightarrow{\omega}) \]
such that

- For all \( a, b \in [n] \) with \( a < b \),
  \[ \forall \text{Ver} (\sigma_{a,b,1}, (a, b); vk_a) = 1 \]
  and
  \[ \forall \text{Ver} (\sigma_{a,b,2}, (a, b); vk_b) = 1 \]

- \( |\overrightarrow{\tau}| = |\overrightarrow{\omega}| = r \)

- \( \text{Open} (c_{\overrightarrow{\tau}}, \omega_{\overrightarrow{\omega}}, z) = 1 \) for every \( j \in [r] \).

\textbf{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 \( F_{SyX} \) 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 \( F_{SyX} \) involving another corrupt party, say \( P_k \) for \( k \in \mathcal{I} \), with a \textit{valid} witness \( w \) in round \( k - 1 \) with \( j < k < i \). \( \mathcal{S} \) sends \( (w, (\omega_{\overrightarrow{\omega}}, z, \omega_{\overrightarrow{\omega}}, z_k)) \) to parties \( P_j \) and \( P_k \).

- Suppose the adversary instructs a corrupt party to trigger an instance of \( F_{SyX} \) with an \textit{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 $\mathcal{F}_{\text{SyX}}$ involving $P_i$ with a valid witness with $j < i$ and the round counter exceeds $i - 1$, $S$ forwards $\{(x_i, f_i)\}_{i \in \mathcal{I}}$ to the trusted party computing $\mathcal{F}_{\text{MPC}}$ with fairness, with $(x_j, f_j)$ replaced with abort for some $j \in \mathcal{I}$, receives $\bot$ as the output of all parties, and aborts itself. Otherwise, as soon as the adversary instructs a corrupt party to trigger an instance of $\mathcal{F}_{\text{SyX}}$ involving $P_i$ with a valid witness with $j < i$ in round $i - 1$, $S$ forwards $\{(x_i, f_i)\}_{i \in \mathcal{I}}$ to the trusted party computing $\mathcal{F}_{\text{MPC}}$ with fairness. It receives the corrupt parties outputs, namely, $\{y_i\}_{i \in \mathcal{I}}$. $S$ chooses the outputs of the honest party completely at random, that is, it samples random strings $y_i \xleftarrow{\$} \{0, 1\}^* \text{ of length equal to the length of the } i\text{th output of } F$, for $i \in [n] \setminus \mathcal{I}$. $S$ then constructs

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

It then defines

$$z = y \oplus \alpha$$

$S$ then computes

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

and constructs

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

$S$ computes $\omega_h^i \xleftarrow{\$} \overline{\text{Open(state}_h, z_h)}$ and constructs

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

Note that, at this point, $S$ has every value ever used in the protocol. $S$ sends $(w, (\omega_1^i, z_1, \omega_j^i, z_j))$ to $P_j$. Going forward, $S$ simulates invocations of the trigger phases of the instances of the ideal functionality $\mathcal{F}_{\text{SyX}}$ that involve corrupt parties as follows.

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

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

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

- Suppose an honest party, say $P_k$ for some $k \in [n] \setminus \mathcal{I}$, triggers an instance of $\mathcal{F}_{\text{SyX}}$ involving a corrupt party, say $P_j$ for some $j \in \mathcal{I}$, $S$ sends $(w, (\omega_k^i, z_k, \omega_j^i, z_j))$ to $P_j$.

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

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

49
• Suppose the adversary instructs a corrupt party, say \( P_j \) for \( i \in \mathcal{I} \), to trigger an instance of \( F_{SyX} \) involving an honest party, say \( P_k \) for \( k \in [n] \setminus \mathcal{I} \), with a valid witness \( w \) in round \( k - 1 \) with \( j < k < n \), \( S \) 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 \( F_{SyX} \) with an invalid witness. \( S \) 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 \( F_{SyX} \) involving \( P_n \) with a valid witness, and the round counter exceeds \( n - 1 \), \( S \) forwards \( \{(x_i, f_i)\}_{i \in \mathcal{I}} \) to the trusted party computing \( F_{MPC} \) with fairness, with \( (x_j, f_j) \) replaced with \( \text{abort} \) 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 \( F_{SyX} \) involving \( P_n \) with a valid witness in round \( n - 1 \), \( S \) forwards \( \{(x_i, f_i)\}_{i \in \mathcal{I}} \) to the trusted party computing \( F_{MPC} \) with fairness. It receives the corrupt parties outputs, namely, \( \{y_i\}_{i \in \mathcal{I}} \). \( S \) 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 \( i \)th output of \( F \), for \( i \in [n] \setminus \mathcal{I} \). \( S \) then constructs
\[
y = y_1 || \ldots || y_n
\]
It then defines
\[
z = y \oplus \alpha
\]
\( S \) then computes
\[
z_n = z \oplus \bigoplus_{i \in [n-1]} z_i
\]
and constructs
\[
\vec{z} = (z_1, \ldots, z_{n-1})
\]
\( S \) computes \( \omega_n^z \xleftarrow{\$} \overline{\text{Open}}(\text{state}_n, z_n) \) and constructs
\[
\vec{\omega}^z = (\omega_1^z, \ldots, \omega_n^z)
\]
Note that, at this point, \( S \) has every value ever used in the protocol. \( S \) sends \((w, (\omega_j^z, z_j, \omega_n^z, z_n))\) to \( P_j \). Going forward, \( S \) simulates invocations of the trigger phases of the instances of the ideal functionality \( F_{SyX} \) 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 \( F_{SyX} \) involving \( P_n \) with a valid witness \( w \) in round \( n - 1 \), \( S \) 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 \( F_{SyX} \) with an invalid witness. \( S \) simply sends no response.

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

\( \Box \)
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 \(F_{SyX}\)-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 \(\pi\) which securely computes \(F_{MPC}\) with fairness in the presence of \(t\)-threshold adversaries for any \(0 \leq t < n\) in the \((F_{OT}, F_{SyX})\)-hybrid model where only parties \(P_1, \ldots, P_t\) can trigger their instances of \(F_{SyX}\).

As discussed in Section 3.14, \(F_{2PC}\), and hence \(F_{OT}\), can be realized in the \(F_{SyX}\)-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 \(\pi\) which securely computes \(F_{MPC}\) with fairness in the presence of \(t\)-threshold adversaries for any \(0 \leq t < n\) in the \(F_{SyX}\)-hybrid model where only parties \(P_1, \ldots, P_t\) can trigger their instances of \(F_{SyX}\).

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 \(F_{SyX}\) as opposed to one. This is because, in the protocol \(\Pi_{F_{MPC}}\), the ideal functionality \(F_{OT}\) will only be used to emulate the ideal functionality \(F_{MPC}\). During this stage, we do not make any use of the ideal functionality \(F_{SyX}\). Once we are done with the single invocation of \(F_{MPC}\), we only invoke the ideal functionality \(F_{SyX}\). As a consequence, parties can reuse the same instance of \(F_{SyX}\) to first emulate \(F_{OT}\) and then as a complete \(F_{SyX}\) 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 possess secure attestation processors 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 \(F_{SyX}\)-hybrid model runs in \(O(n)\) rounds. Since our \(F_{SyX}\) implementation in the \((G_{att}, F_{BB}, G_{accs})\)-hybrid model requires two write queries to \(F_{BB}\) (i.e., two writes on the blockchain), it follows that the real-world implementation of our protocol will require \(O(n)\)
we preprocess an 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 \( \mathcal{FSyX} \) 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, \( \mathcal{FSyX} \) provides \( K_{i,j}^{1}, c_{i,j}^{1}, c_{j,i}^{1} \) to \( P_i \) and \( K_{j,i}^{1}, c_{j,i}^{1}, c_{i,j}^{1} \) to \( P_j \), where \( K_{i,j}^{1} \oplus K_{j,i}^{1} = K_{i,j}^{1} \) and \( c_{i,j}^{1}, c_{j,i}^{1} \) are respectively commitments on \( K_{i,j}^{1} \), \( K_{j,i}^{1} \). 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 \( \mathcal{FSyX} \) in the “unpreprocessed” fair protocol. Recall that in the “unpreprocessed” version of our fair protocol, the \( \mathcal{FSyX} \) 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 \( \mathcal{FSyX} \) 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 \( \mathcal{FSyX} \) 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 \( \mathcal{FSyX} \) 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 \( \mathcal{FSyX} \), party \( P_i \) will have to provide a tuple \((id, T, (c_1, \ldots, c_n), (w_1, \ldots, w_j))\). Upon receiving this tuple, \( \mathcal{FSyX} \) 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 \( \mathcal{FSyX} \) outputs a derived key \( K_{id}^{i,j} = \text{Hash}(K_{i,j}^{i,j}, id, T, c_1, \ldots, c_n) \) and outputs this to both \( P_i \) and \( P_j \) in a fair manner.

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

Note that the unfair MPC protocol \( \pi_{MPC} \) will also check if the input key shares \( K_{i,j}^{i,j} \) and \( K_{j,i}^{i,j} \) are consistent with the openings of the commitments \( c_{i,j}^{i,j} \) and \( c_{j,i}^{i,j} \) 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 \( K_{j,j+1}^{j,j+1} \neq K_{j,j+1}^{j,j+1} \) to \( \pi_{MPC} \). Now, in round \( j \), suppose corrupt \( P_{j-1} \) obtained \( y_j \) by triggering the \((j-1, j)\) \( \mathcal{FSyX} \) instance. Then in round \( j+2 \), honest \( P_j \) would trigger the \((j, j+1)\) \( \mathcal{FSyX} \) instance in an attempt to learn \( y_{j+1} \). While \( P_j \) would indeed learn the correct instance specific derived key \( K_{id}^{j,j+1} \) by triggering \( \mathcal{FSyX} \), this key turns
out to be useless for decrypting the ciphertext $e_{id}^{j,j+1}$ 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 $F_{SyX}$ 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 $e_{id}^{j,j+1}$ from the unfair MPC protocol. Now, corrupt $P_i$ can trigger $(1,j)$ $F_{SyX}$ 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 $(j,j+1)$ $F_{SyX}$ instance to obtain the (correct) derived $K_{id}^{j,j+1}$, it does not have the ciphertext $e_{id}^{j,j+1}$ 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 $\pi_{\text{MPC}}$ output $n$-out-of-$n$ additive shares of $\{e_{id}^{i,j}\}_{i<j}$ 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 $\{e_{id}^{i,j}\}_{i<j}$. 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 $F_{SyX}$ to learn the output, and when they do this, $F_{SyX}$ 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 $F_{SyX}$ instance to obtain derived keys corresponding to this protocol instance. Therefore, the attack described previously cannot be carried out in the modified protocol.

References


[ABB+18] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Chris tidis, 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,


**Disclaimer**

Case studies, comparisons, statistics, research and recommendations are provided “AS IS” and intended for informational purposes only and should not be relied upon for operational, marketing, legal, technical, tax, financial or other advice. Visa Inc. neither makes any warranty or representation as to the completeness or accuracy of the information within this document, nor assumes any liability or responsibility that may result from reliance on such information. The Information contained herein is not intended as investment or legal advice, and readers are encouraged to seek the advice of a competent professional where such advice is required.

These materials and best practice recommendations are provided for informational purposes only and should not be relied upon for marketing, legal, regulatory or other advice. Recommended marketing materials should be independently evaluated in light of your specific business needs and any applicable laws and regulations. Visa is not responsible for your use of the marketing materials, best practice recommendations, or other information, including errors of any kind, contained in this document.