Fair and Robust Multi-Party Computation using a Global Transaction Ledger

Aggelos Kiayias[1], Hong-Sheng Zhou[2], and Vassilis Zikas[3]

[1] National and Kapodistrian University of Athens, aggelos@di.uoa.gr
[2] Virginia Commonwealth University, hszhou@vcu.edu
[3] Rensselaer Polytechnic Institute, vzikas@cs.rpi.edu

Abstract. Classical results on secure multi-party computation (MPC) imply that fully secure computation, including fairness (either all parties get output or none) and robustness (output delivery is guaranteed), is impossible unless a majority of the parties is honest. Recently, cryptocurrencies like Bitcoin where utilized to leverage the fairness loss in MPC against a dishonest majority. The idea is that when the protocol aborts in an unfair manner (i.e., after the adversary receives output) then honest parties get compensated by the adversarially controlled parties.
Our contribution is three-fold. First, we put forth a new formal model of secure MPC with compensation and show how the introduction of suitable ledger and synchronization functionalities makes it possible to describe such protocols using standard interactive Turing machines (ITM) circumventing the need for the use of extra features that are outside the standard model as in previous works. Second, our model, is expressed in the universal composition setting with global setup and is equipped with a composition theorem that enables the design of protocols that compose safely with each other and within larger environments where other protocols with compensation take place; a composition theorem for MPC protocols with compensation was not known before. Third, we introduce the first robust MPC protocol with compensation, i.e., an MPC protocol where not only fairness is guaranteed (via compensation) but additionally the protocol is guaranteed to deliver output to the parties that get engaged and therefore the adversary, after an initial round of deposits, is not even able to mount a denial of service attack without having to suffer a monetary penalty. Importantly, our robust MPC protocol requires only a *constant* number of (coin-transfer and communication) rounds.

1 Introduction

Secure multiparty computation (MPC) enables a set of parties to evaluate the output of a known function $f(\cdot)$ on inputs they privately contribute to the protocol execution. The design of secure MPC protocols, initiated with the seminal works of Yao [31] and Goldreich et al. [21] has evolved to a major effort in computer security engineering. Beyond privacy, a secure MPC protocol is highly desirable to be *fair* (either all parties learn the output or none) and *robust* (the delivery of the output is guaranteed and the adversary cannot mount a "denial of service" against the protocol). Achieving fairness and robustness in a setting where there is an arbitrary number of corruptions, as desirable as it may appear, is prohibited by strong impossibility results stemming from the work of Cleve [16] who showed that coin-flipping is infeasible in any setting where there is no honest majority among parties that execute the protocol. These impossibility results, combined with the importance of the properties that they prevent, strongly motivate the exploration of alternate – yet still realistic – models that would enable fair and robust MPC protocols.

With the advent of Bitcoin [28] and other decentralized cryptocurrencies, the works of [1, 2, 7, 27] showed a new direction for circumvention of the impossibility results regarding the fairness property: enforcing fairness could be achieved through imposing monetary penalties. In this setting a breach of fairness by the adversary is still possible but it results in the honest parties collecting a compensation in a way that is determined by the protocol execution. At the same time, in case fairness is not breached, it is guaranteed that no party loses any money (despite the fact that currency transfers may have taken place between the parties). The rationale here is that a suitable monetary penalty suffices in most practical scenarios to force the adversary to operate in the protocol fairly.

While the main idea of fairness with penalties sounds simple enough, its implementation proves to be quite challenging. The main reason is that the way a crypto-currency operates does not readily provide a trusted party that will collect money from all participants and then either return it or redistribute it according to the pre-agreed penalty structure. This is because crypto-currencies are decentralized and hence no single party is ever in control of a money transfer beyond the owner of a set of coins. The mechanism used in [1, 2, 7, 27] to circumvent the above problem is the capability[4] of the Bitcoin network to issue transactions that are "time-locked", i.e., become valid only after a specific time and prior to that time may be superseded by other transactions that are posted in the public ledger. Superseded time-locked transactions become invalid and remain in the ledger without ever being redeemed.

While the above works are an important step for the design of MPC protocols with properties that circumvent the classical impossibility results, several critical open questions remain to be tackled; those we address herein are as follows.

Our Results. Our contribution is three-fold. First, we put forth a new formal model of secure MPC with compensation and we show how the introduction of suitable ledger and synchronization functionalities makes it possible to express completely such protocols using standard interactive Turing machines (ITM) circumventing the need for the use of extra features that are outside the standard model (in comparison, the only previous model [7] resorted to specialized ITM's that utilize resources outside the computational model[5]). Second, our model is equipped with a composition theorem that enables the design of protocols that

[4] Note that this feature is currently not fully supported.

[5] An ITM with the special features of "wallet" and "safe" was introduced in [7] to express the ability of ITM's to store and transfer "coins." Such coins were treated as physical quantities that were moved between players but also locked in safes in a

compose safely with each other and within larger environments where other protocols with compensation take place; a composition theorem for this class of protocols was not known before and requires a new framework for synchronization in the global UC setting that can be of independent interest. Third, we introduce the first robust MPC protocol with compensation, i.e., an MPC protocol where not only fairness is guaranteed (via compensation) but additionally the protocol is guaranteed to deliver output to the parties that get engaged and therefore the adversary is not even able to mount a denial of service attack without having to suffer a monetary penalty. In more details we have the following.

- We put forth a new model that utilizes two ideal functionalities and expresses the ledger of transactions and a clock in the sense of [24] that is connected to the ledger and enables parties to synchronize their protocol interactions. Our ledger functionality enable us to abstract all the necessary features of the underlying cryptocurrency. Contrary to the only previous formalization approach [7,27], our modeling allows the entities that participate in an MPC execution to be regular interactive Turing machines (ITM) and there is no need to equip them with additional physical features such as "safes" and "locks." Furthermore the explicit inclusion of the clock functionality (which is only alluded to in [7,27]) and a synchronous framework for protocol design given such clock reveal the exact dependencies between the ledger and the clock functionality that are necessary in order for MPC with compensation protocols to be properly described. We express our model within a general framework that we call Q-fairness and robustness and may be of independent interest as it can express meaningful relaxations of fairness and robustness in the presence of a global ideal functionality.
- We prove a composition theorem that establishes that protocols in our framework are secure in a universally composable fashion. Our composition proof treats the clock and ledger functionalities as global setups in the sense of [11,13]. We emphasize that this is a critical design choice: the fact that the ledger is a global functionality ensures that any penalties that are incurred to the adversary that result to credits towards the honest parties will be globally recognized. This should be contrasted to an approach that utilizes regular ideal functionalities which may be only accessible within the scope of a single protocol instance and hence any penalty bookkeeping they account may vanish with the completion of the protocol. Providing a composition theorem for MPC protocols with compensation was left as an open question in [7].
- We finally present a new protocol for fair and robust secure MPC with compensation. Our robustness property guarantees that once the protocol passes an initial round of deposits, parties are guaranteed to obtain output or be compensated. This is in contrast to fair MPC with compensation [1,2,7,27] where the guarantee is that compensation takes place only in case the adversary obtains output while an honest party does not. To put it differently,

way that parties were then prevented to use them in certain ways (in other words such safes were not local but were affected from external events).

it is feasible for the adversary to lead the protocol to a deadlock where no party receives output however the honest parties have wasted resources by introducing transactions in the ledger. We remark that it is in principle possible to upgrade the protocols of [1, 2, 7, 27] to the robust MPC setting by having them perform an MPC with identifiable abort, cf. [21, 23], (in such protocol the party that causes the abort can be identified and excluded from future executions). However even using such protocol the resulting robust MPC with compensation will need in the worst case a *linear* number of deposit/communication rounds in the number of malicious parties. Contrary to that, our robust protocol can be instantiated so that it requires a constant number of deposit/communication rounds independently of the number of parties that are running the protocol. Our construction uses time-locked transactions in a novel way to ensure that parties do progress in the MPC protocol or otherwise transactions are suitably revertible to a compensation for the remaining parties. The structure of our transactions is quite more complex than what can be presently supported by bitcoin; we describe in high-level how our protocol can be implemented via Ethereum[6] contracts.

Related work. In addition to the previous works [1, 2, 7, 27] in fair MPC with compensation, very recently, Ruffing et al. [30] address equivocation issues via penalty mechanism, and design decentralized "non-equivocation" contracts.

There are a number of other works that attempted to circumvent the impossibility results for fairness in the setting of dishonest majority by considering alternate models. Contrary to the approach based on cryptocurrencies these works give an advantage to the protocol designer with respect to the adversarial strategy for corruption. For instance, in [18] a rational adversary is proposed and the protocol designer is privy to the utility function of the adversary. In [3] a reputation system is used and the protocol designer has the availability of the reputation information of the parties that will be engaged in the protocol. Finally in [17] a two tiered model is proposed where the protocol designer is capable of distinguishing two distinct sets of servers at the onset of the computation that differ in terms of their corruptibility.

Global setups were first put forth in [11] motivated by notion of deniability in cryptographic protocols. In our work we utilize global functionalities for universal composition (without the deniability aspect) as in [13] where a similar approach was taken for the case of the use of the random oracle as a global setup functionality for MPC.

Fairness was considered from the resource perspective, cf. [8, 19, 29], where it is guaranteed due to the investment of proportional resources between the parties running the protocol, and the optimistic perspective, cf. [4, 5, 9], where a trusted mediator can be invoked in the case of an abort. We finally note that without any additional assumptions, due to the impossibility results mentioned above, one can provide fairness only with certain high probability that will be affecting the complexity of the resulting protocol, see, e.g., [22] and references therein.

[6] `http://www.ethereum.org`.

In concurrent and independent work, Kosba et al [26] propose a framework for composable protocols based on a ledger. and explore a notion of fairness with compensation. Our work goes beyond fairness and provides a treatment of robustness. Furthermore we provide a synchronous framework with a global clock (of independent interest) that uses the ledger as a *global* setup to achieve fairness and robustness and we prove a composition theorem for our framework.

Organization. We start with preliminaries in Section 2. Then in Sections 3 and 4, we lay down a formal framework for designing composable fair protocols in the presence of globally available trusted resources. In Section 3, we introduce two shared functionalities $\bar{\mathcal{G}}_{\text{CLOCK}}$ and $\bar{\mathcal{G}}_{\text{LEDGER}}$ respectively to formulate the trust resources that are provided by Bitcoin-like systems. Subsequently, in Section 4, we put forth a new formal framework for secure MPC with compensation: we introduce the notions of Q-fairness, and Q-robustness via wrapper functionalities; we then consider the realization of such wrapper functionalities, and further provide a composition theorem. In Section 5, we present a protocol in our new framework to achieve our new notions of fairness and robustness. We refer the reader to the full version of our work [25] for a discussion about implementing our protocol within Ethereum, supplementary material to for Sections 2 and 3, and for the formal proofs of our theorems.

2 Preliminaries

Throughout the paper we assume an (often implicit) security parameter denoted as κ. For a number $n \in \mathbb{N}$ we denote by $[n]$ the set $[n] = \{1, \ldots, n\}$ and denote by 0^n (resp. 1^n) the all-zero (resp. all-ones) string of length n. For a randomized algorithm Alg we denote by $\text{Alg}(x; r)$ the output of Alg on input x and random coins r. To avoid always explicitly writing the coins r, we shall denote by $y \overset{\$}{\leftarrow} \text{Alg}(x)$ the operation of running Alg on input x (and uniformly random coins) and storing the output on variable y. We write $f : X \overset{\$}{\rightarrow} Y$ to denote a probabilistic function with domain X and range Y. We use the standard definition of *negligible* and *overwhelming* (e.g., see [20]).

For a multiparty function $f : (\{0,1\}^* \cup \{\lambda\})^n \rightarrow (\{0,1\}^* \cup \{\bot\})^n$ for parties in $\mathcal{P} = \{p_1, \ldots, p_n\}$ and for a set $\mathcal{P} \subseteq \mathcal{P}$, we denote by $f|_{|\mathcal{P}'|}$ the restriction of f to the parties in $\mathcal{P}'$, namely, if each $p_i \in \mathcal{P}'$ has input x_i, then the output of $f|_{|\mathcal{P}'|}$ is the output of f evaluated on inputs x_i for each $p_i \in \mathcal{P}'$ and $x_j = \lambda$ for each $p_j \in \mathcal{P} \setminus \mathcal{P}'$.

We describe our results in the extension of Canetti's UC framework [10] to allow for global setups, known as GUC [11]. As argued above, this is the natural model to consider execution in the present of a globally synchronized clock and a ledger/bulletin board. Consistently with the (G)UC notation, we denote local (UC) functionalities by calligraphic letters, as in $\mathcal{F}$, and add a bar to denote global functionalities, as in $\bar{\mathcal{G}}$. Furthermore, we denote by ϕ, the dummy protocol. Note that in GUC ϕ might receive inputs for its (UC) hybrids and/or for the global setup, where an implicit mechanism is assumed to allow the environment to define the intended recipient of each submitted input to ϕ. For a protocol π, a (local) UC functionality $\mathcal{F}$ and a global setup $\bar{\mathcal{G}}$ we denote by $\textsc{Exec}^{\bar{\mathcal{G}},\mathcal{F}}_{\pi,\mathcal{A},\mathcal{Z}}$ the output of the environment $\mathcal{Z}$ in an execution of π having hybrid access to $\bar{\mathcal{G}}$ and $\mathcal{F}$ in the presence of adversary $\mathcal{A}$. We assume some familiarity with the UC and/or the GUC framework.

Correlated Randomness as a Sampling Functionality Our protocols are in the *correlated randomness* model, i.e., they assume that the parties initially, before receiving their inputs, receive appropriately correlated random strings. In particular, the parties jointly hold a vector $\boldsymbol{R} = (R_1, \dots, R_n) \in (\{0,1\}^*)^n$, where P_i holds R_i, drawn from a given efficiently samplable distribution $\mathcal{D}$. This is, as usual, captured by giving the parties initial access to an ideal functionality $\mathcal{F}^{\mathcal{D}}_{\text{CORR}}$, known as a *sampling functionality*, which, upon receiving a default input from any party, samples $\boldsymbol{R}$ from $\mathcal{D}$ and distributes it to the parties (see [25] for details). Hence, a protocol in the correlated randomness model is formally an $\mathcal{F}^{\mathcal{D}}_{\text{CORR}}$-hybrid protocol. Formally, a sampling functionality $\mathcal{F}^{\mathcal{D}}_{\text{CORR}}$ is parameterized by an efficiently computable sampling distribution $\mathcal{D}$ and the (ID's of the parties in) the player set $\mathcal{P}$.

3 Model

In this section and next section, we lay down a formal framework for designing composable fair protocols in the presence of globally available trusted resources. we introduce in the current section, shared (in the sense of the GUC model [11]) functionalities $\bar{\mathcal{G}}_{\text{CLOCK}}$ and $\bar{\mathcal{G}}_{\text{LEDGER}}$ respectively to formulate the trust resources that are provided by Bitcoin-like systems. We stress that these two functionalities can be thought of as a single global functionality and in our description are allowed to communicate. Nonetheless, we choose to describe then as two separate functionalities, because as we argue, the clock $\bar{\mathcal{G}}_{\text{CLOCK}}$ can also be used alone (without $\bar{\mathcal{G}}_{\text{LEDGER}}$) to naturally model synchronous computation with a global notion of time.

3.1 Global Clock Functionality and Synchronous Protocol Executions

In this section we describe how to model execution of synchronous protocols that can access a global-clock setup. This is an adaptation of the original idea by Katz et al. [24], where a clock was modelled as UC functionality that is local to the calling protocol, and is of independent interest as a model for the design of synchronous protocols. In addition to being a more realistic model for capturing time in UC, the notion of the global clock allows for synchronous execution of any protocols that choose to use it.

Before defining our clock, we recall the reader the clock and model of synchronous execution from [24] and then highlight the main differences. The clock in [24] is a UC functionality that keeps an indicator bit b originally set to 0. The parties can send to the clock special "update" messages, and once the clock sees that all honest parties agree to update the state it sets $b := b \oplus 1$. The clock then continues to receive "update" messages, and again, once it sees that all honest parties have requested to update after the last switch of the bit b it switches it again. To make sure that the adversary is given enough activations, whenever the clock receives an "update" message from the honest party it notifies the adversary. In addition to "update" messages, the parties can send the clock a "read" message which the clock replies with the current value of b.

The use of such a clock to keep a round structure is as follows: Whenever a party observes a switch of the bit b, it interprets it as a round advance. Thus, a synchronous protocol with access to such a clock is executed as follows. In each round, every party performs all its protocol instructions for the current round, and at the end sends an "update" message to the clock; from the point where the party updates (its round has finished) it queries ("reads") the clock with each following activation to detect when all parties have also finished their rounds (i.e., when the value of b switches). Once this happens, the party starts its next protocol round.

An issue with the above clock is that in order to execute two protocols using the same clock we need to make use of the joint-state UC theorem [15]. Instead, in this work we take an alternative modelling approach and define a shared clock functionality $\bar{\mathcal{G}}_{\text{CLOCK}}$. This functionality can be viewed as a shared version of the clock functionality which was defined by Katz et al. [24]. The main intuition behind our clock functionality is that all honest parties can use it to ensure that they proceed with their rounds at the same pace. On a hight level, the clock operates as follows: any party that wishes to be synchronised with the global clock can send $(\text{REGISTER}, \mathsf{sid})$ to the clock and subsequently it can send it $(\text{CLOCK-UPDATE}, \mathsf{sid})$ commands, where sid is $\bar{\mathcal{G}}_{\text{CLOCK}}$'s identifier. The clock stores a global-time counter τ (initially set to 0), and as soon it is instructed by all currently honest parties and by associated shared functionalities[7] to advance the time (i.e., receives $(\text{CLOCK-UPDATE}, \mathsf{sid})$ it increases its state-counter τ by 1.

The main difference between our formulation and that by Katz et al. [24] is that in [24] the clock is a UC functionality which is local to a single protocol and waits for an "update" message by every honest party to advance its state; however, here we intend to have the clock to be accessed *globally* and used by arbitrary protocols. Therefore we give the power to the environment to define the clock's speed. Indeed, if there are no associated shared functionalities, the environment can instruct dummy parties to send inputs $(\text{CLOCK-UPDATE}, \mathsf{sid})$ to $\bar{\mathcal{G}}_{\text{CLOCK}}$ and advance the clock whenever it wishes. An additional difference is that in [24], the clock state is binary while here, in our formulation, the state τ is a positive integer which indicates the time that has passed from point zero (i.e., from the beginning of time).

Next, we elaborate and explain how to use the global clock to design synchronous protocols. We remark that the model of synchronous protocol execution

[7] Certain global functionalities, such as the ledger defined in the following section, might depend on time and, therefore, need to be synchronized with the clock.

of [24] cannot be used in our setting as the environment can make the clock advance before honest parties have time to take actions in any round. Indeed, in the ideal setting the environment can keep sending (CLOCK-UPDATE, sid) to the dummy parties, which will forward it to the clock making its state to advance; to make sure that the protocol is indistinguishable, honest parties would have to do the same, thereby giving away the activations that they need for executing their protocol instructions such as send and receive operations.[8] This might, at first, seem like a bug but it is in fact a feature. It captures the fact that since time is a quantity that should be in the control of the environment, if the environment chooses to advance time too fast then some protocol might not have enough time to perform their operations for each round, and might therefore need to give up.

To make sure that the environment cannot exploit such fast-forwarding of the clock we use the following idea: We allow the clock to receive from honest parties or (non-shared) ideal functionalities a special (CLOCK-FAST) message, which makes it set an internal indicator from 0 to 1. This indicator will be added onto the response of the clock to CLOCK-READ queries, and will make any synchronous protocol or corresponding functionality that reads the clock and observes this indicator being set to one to immediately terminate with a default value. This way we ensure that an environment that tries such a fast-forward distinguishing attack will be forced to make any synchronous protocol behave in a default way, a behavior which, as we see, is easily imitated in the ideal world. The detailed description of the clock functionality can be found in Figure 1.

We stress that having a global $\bar{\mathcal{G}}_{\text{CLOCK}}$-hybrid model makes the mode of execution of synchronous protocols more intuitive compared to [24]. Here is how synchronous protocols are executed in this setting. First, as is the case in real-life synchronous protocols, we assume that the protocol participants have agreed on the starting time τ_0 of their protocol and also on the duration of each round.[9] We abstract this knowledge by assuming the parties know a function $\mathtt{Round2Time} : \mathbb{Z} \to \mathbb{Z}$ which maps protocol rounds to time (according to the global clock) in which the round should be completed. For $\rho \in \mathbb{Z}$, $\mathtt{Round2Time}(\rho)$ is the time in which the ρth round of the protocol should be completed. To make sure that no party proceeds to round $\rho + 1$ of the protocol before all honest parties have completed round ρ , we require that any two protocol rounds are at least two clock-ticks apart (see [24] for a discussion); formally, for all $\rho \geq 0$, it holds that $\mathtt{Round2Time}(\rho + 1) \geq \mathtt{Round2Time}(\rho) + 2$.

A synchronous protocol in the above setting proceeds as follows where the parties keep locally track of the current round ρ in the protocol they are in:

- Upon receiving a (CLOCK-UPDATE, sid) input (from its environment) where sid is the ID of $\bar{\mathcal{G}}_{\text{CLOCK}}$, party P_i forwards it to $\bar{\mathcal{G}}_{\text{CLOCK}}$.
- Upon receiving a (CLOCK-READ, sid) input (from its environment), party P_i forwards it to $\bar{\mathcal{G}}_{\text{CLOCK}}$ and outputs the response to the environment.

[8] The communication channels we are using are fetch-type bounded delivery channels as in [24]. In such channels, the receiver needs to issue "fetch"-requests which are answered only if a message is ready for delivery. We refer to [24] for details.

[9] Different protocols might proceed at a different pace.

Functionality $\bar{\mathcal{G}}_{\text{CLOCK}}$

Shared functionality $\bar{\mathcal{G}}_{\text{CLOCK}}$ is globally available to all participants. The shared functionality is parameterized with variables τ, a bit $d_{\bar{\mathcal{G}}_{\text{LEDGER}}}$ a set $\mathcal{P}'$ and a bit $\texttt{fast}$ and is associated with a ledger shared functionality $\bar{\mathcal{G}}_{\text{LEDGER}}$.

Initially, $\tau := 0$, $d_{\bar{\mathcal{G}}_{\text{LEDGER}}} := 0$, $\texttt{fast} := 0$ and $\mathcal{P}' := \emptyset$.

- Upon receiving $(\text{REGISTER}, \mathsf{sid})$ from some party P, set $\mathcal{P}' := \mathcal{P}' \cup \{P\}$ and if P was not registered before, set $d_P := 0$; subsequently, forward $(\text{REGISTER}, \mathsf{sid}, P)$ to $\mathcal{A}$.
- Upon receiving $(\text{CLOCK-UPDATE}, \mathsf{sid})$ from $\bar{\mathcal{G}}_{\text{LEDGER}}$ set $d_{\bar{\mathcal{G}}_{\text{LEDGER}}} := 1$ and forward $(\text{CLOCK-UPDATE}, \mathsf{sid}, \bar{\mathcal{G}}_{\text{LEDGER}})$ to $\mathcal{A}$
- Upon receiving $(\text{CLOCK-UPDATE}, \mathsf{sid})$ from some honest party $P \in \mathcal{P}'$ set $d_i := 1$; then if $d_{\bar{\mathcal{G}}_{\text{LEDGER}}} := 1$ and $d_P = 1$ for all honest parties in $\mathcal{P}'$, then set $\tau := \tau + 1$ and reset $d_{\bar{\mathcal{G}}_{\text{LEDGER}}} := 0$ and $d_P := 0$ for all parties in $\mathcal{P}'$. Forward $(\text{CLOCK-UPDATE}, \mathsf{sid}, P)$ to $\mathcal{A}$.
- Upon receiving $(\text{CLOCK-READ}, \mathsf{sid})$ from any participant (including the environment, the adversary, or any ideal—shared or local—functionality) return $(\text{CLOCK-READ}, \mathsf{sid}, \tau, \texttt{fast})$ to the requestor.
- Upon receiving (Clock-Fast) from any honest party or ideal functionality, set $\texttt{fast} := 1$.

Fig. 1. The clock functionality.

- Upon receiving a (Clock-Fast) input (from its environment), party P_i forwards it to $\bar{\mathcal{G}}_{\text{CLOCK}}$.
- Upon receiving any message $(\text{Input}, \mathsf{sid}')$ where sid' is the session ID of a protocol P_i is involved in, do the following: Send $(\text{CLOCK-READ}, \mathsf{sid})$ to $\bar{\mathcal{G}}_{\text{CLOCK}}$ and denote the response by $(\text{CLOCK-READ}, \mathsf{sid}, \tau, \texttt{fast})$; if $\texttt{fast} = 1$ then output Clock-Fast to the environment. Otherwise do:
  - if $\tau \leq \texttt{Round2Time}(\rho - 1)$ halt;
  - else, if $\texttt{Round2Time}(\rho - 1) < \tau \leq \texttt{Round2Time}(\rho)$ execute the next pending round$-\rho$ instruction (if all the instructions for round ρ are finished halt.)
  - else, if $\tau > \texttt{Round2Time}(\rho)$ and there are still pending instructions for the current round, send (Clock-Fast) to $\bar{\mathcal{G}}_{\text{CLOCK}}$.
  - else, i.e., if $\tau > \texttt{Round2Time}(\rho)$ and P_i has completed all round-ρ instruction, then set $\rho := \rho + 1$ and halt.

It is easy to verify that the above mode of operation will guarantee that the parties are never out-of-sync, since as soon as the first party issues a Clock-Fast message for the clock, all synchronous protocols will enter the mode of outputting Clock-Fast for every input that the environment hands them (that is not intended for the clock). However, there is one more thing that needs to be taken care of. Since in the real-world the parties go to a default mode (where they output Clock-Fast to every query) when the environment does not give them sufficient time, this should also be the case in the ideal world. To achieve this we use another idea inspired by the guaranteed termination functionality from [24]: Let π be a synchronous protocol with round-to-time function $\texttt{Round2Time} : \mathbb{Z} \to \mathbb{Z}$, where in each round, each party needs exactly m activations to perform its instructions[10]. We introduce a wrapper $\tilde{\mathcal{W}}$ which, at a high level, forwards messages to and from its wrapped functionality but stores a round-index and checks, as the protocol would, that every party issues to the wrapped functionality, at least m activations for each round ρ in the intended interval. If this is not the case the wrapper sends (Clock-Fast) to $\bar{\mathcal{G}}_{\text{CLOCK}}$ and responds with Clock-Fast form that point on. The detailed description can be found in the full version [25].

3.2 Global Ledger Functionality

Functionality $\bar{\mathcal{G}}_{\text{LEDGER}}$ provides the abstraction of a public ledger in Bitcoin-like systems (e.g., Bitcoin, Litecoin, Namecoin, Ethereum, etc). Intuitively, the public ledger could be accessed globally by protocol parties or other entities including the environment $\mathcal{Z}$. Protocol parties or the environment can generate transactions; and these valid transactions will be gathered by a set of ledger maintainers (e.g., miners in Bitcoin-like systems) in a certain order as the state of the ledger. More concretely, whenever the ledger maintainers receive a vector of transactions $\texttt{tx}$, they first add the transactions in a buffer, assuming they are valid with respect to the existing transactions and the state of the ledger; thus, in this way a vector of transactions is formed in the buffer. After a certain amount of time, denoted by $\texttt{T}$, which will be also referred to as a *ledger round*, all transactions in the buffer will be "glued" into the ledger state in the form of a block. The adversary is allowed to permute the buffer prior to its addition to the ledger. In Bitcoin, $\texttt{T}$ is 10 minutes (approximately); thus in about every 10 minutes, a new block of transactions will be included into the ledger, and the ledger state will be updated correspondingly.

To enable the ledger to be aware of time, the ledger maintainers are allowed to "read" the state of another publicly available functionality $\bar{\mathcal{G}}_{\text{CLOCK}}$ defined above. Furthermore, to ensure that the ledger is activated at least once in each time-tick[11] (i.e., each advance of the $\bar{\mathcal{G}}_{\text{CLOCK}}$ state) we have the ledger, with every message it gets from a party other than the adversary, send a $(\text{CLOCK-UPDATE}, \mathsf{sid})$ message to $\bar{\mathcal{G}}_{\text{CLOCK}}$. (Recall that, as defined, $\bar{\mathcal{G}}_{\text{CLOCK}}$ always waits for at least one such message from the ledger before advancing its time counter.)

We remark that all gathered transactions should be "valid" which is defined by a predicate $\mathsf{Validate}$. In different systems, predicate $\mathsf{Validate}$ will take different forms. For example, in the Bitcoin system, the predicate $\mathsf{Validate}$ should make sure that for each newly received transaction that transfers v coins from the original wallet address $\texttt{address}_o$ to the destination wallet address $\texttt{address}_d$,

[10] One can make any synchronous protocol have this form by introducing dummy instructions.

[11] This is essential to ensure that updates are done in a time-consistent manner.

Functionality $\bar{\mathcal{G}}_{\text{LEDGER}}$

Shared functionality $\bar{\mathcal{G}}_{\text{LEDGER}}$ is globally available to all participants. The shared functionality is parameterized with a predicate $\mathsf{Validate}$, a constant $\mathtt{T}$, and variables state, buffer and $\mathsf{counter}$.

Initially, $\mathsf{state} := \varepsilon$, $\mathsf{buffer} := \varepsilon$, and $\mathsf{counter} := 0$.

- Upon receiving $(\text{SUBMIT}, \mathsf{sid}, \mathtt{tx})$ from some participant, If $\mathsf{Validate}(\mathsf{state}, (\mathsf{buffer}, \mathtt{tx})) = 1$, then set $\mathsf{buffer} := \mathsf{buffer}||\mathtt{tx}$. Go to *State Extend.*
- Upon receiving $(\text{READ}, \mathsf{sid})$ from a party P or $\mathcal{A}$, if P is honest set $b = \mathsf{state}$ else set $b = (\mathsf{state}, \mathsf{buffer})$.
  1. Execute *State Extend.*
  2. Return $(\text{READ}, \mathsf{sid}, b)$ to the requestor.
- Upon receiving $(\text{PERMUTE}, \mathsf{sid}, \pi)$ from $\mathcal{A}$ apply permutation π on the elements of buffer.

State Extend: Send $(\text{CLOCK-READ}, \mathsf{sid})$ to $\bar{\mathcal{G}}_{\text{CLOCK}}$ and receive $(\text{CLOCK-READ}, \mathsf{sid}, \tau)$ from $\bar{\mathcal{G}}_{\text{CLOCK}}$. If $|\tau - \mathtt{T} \cdot \mathsf{counter}| > \mathtt{T}$, then set $\mathsf{state} := \mathsf{state}||\mathsf{Blockify}(\tau, \mathsf{buffer})$ and $\mathsf{buffer} := \varepsilon$ and $\mathsf{counter} := \mathsf{counter} + 1$. Subsequently, send $(\text{CLOCK-UPDATE}, \mathsf{sid})$ to $\bar{\mathcal{G}}_{\text{CLOCK}}$ where sid is the ID of $\bar{\mathcal{G}}_{\text{CLOCK}}$.

Fig. 2. The public ledger functionality.

the original wallet address $\mathtt{address}_o$ should have v or more than v coins, and the transaction should be generated by the original wallet holder (as shown by the issuance of a digital signature). Furthermore, prior to each vector of transactions becoming block, the vector is passed through a function $\mathsf{Blockify}(\cdot)$ that homogenizes the sequence of transactions in the form of a block. Moreover, in some systems like Bitcoin, it may add a special transaction called a "coinbase" transaction that implements a reward mechanism for the ledger maintainers.

In Figure 2 we provide the details of the ledger functionality.

4 Q-Fairness and Q-Robustness

In this section, we provide a formal framework for secure computation with fair and robust compensation. In the spirit of [19], our main tool is a wrapper functionality. Our wrapper functionality is equipped with a predicate $\mathsf{Q}_{\bar{\mathcal{G}}}$ which is used to make sure that the outcome of the protocol execution is consistent with appropriate conditions on the state of the global setup $\bar{\mathcal{G}}$. Intuitively, the predicate $\mathsf{Q}_{\bar{\mathcal{G}}}$ works as a filter, such that if certain "bad" event occurs (e.g., an abort), then the wrapped functionality will restrict the simulators influence. More concretely, the predicate $\mathsf{Q}_{\bar{\mathcal{G}}}$ has three modes $\mathsf{Q}^{\text{Init}}_{\bar{\mathcal{G}}}$, $\mathsf{Q}^{\text{Dlv}}_{\bar{\mathcal{G}}}$ and, $\mathsf{Q}^{\text{Abt}}_{\bar{\mathcal{G}}}$, where $\mathsf{Q}^{\text{Init}}_{\bar{\mathcal{G}}}$ specifies under which condition (on the global setup's state) the protocol should start executing; $\mathsf{Q}^{\text{Dlv}}_{\bar{\mathcal{G}}}$ specifies under which condition parties should receive their output; and $\mathsf{Q}^{\text{Abt}}_{\bar{\mathcal{G}}}$ specifies under which condition the simulator is allowed to force parties to abort. With foresight $\mathsf{Q}_{\bar{\mathcal{G}}}^{\mathrm{Init}}$ will ensure that the protocol is executed only if all honest participants have enough coins; $\mathsf{Q}_{\bar{\mathcal{G}}}^{\mathrm{Dlv}}$ will ensure that honest parties do not lose coins if they execute the protocol; and $\mathsf{Q}_{\bar{\mathcal{G}}}^{\mathrm{Abt}}$ will ensure that honest parties might be forced to an "unfair" abort (i.e, where the adversary has received his output) only if they are compensated by earning coins (from the corrupted parties). We will call an implementation of a wrapped version of $\mathcal{F}$ a Q-fair implementation of $\mathcal{F}$. [12]

Our definition of $\mathsf{Q}_{\bar{\mathcal{G}}}$-fairness can be instantiated with respect to any global setup that upon receiving a READ symbol (from any protocol participant or functionality) it returns its public state trans. Concretely, let $\bar{\mathcal{G}}$ be global ideal functionality and let $\mathsf{Q}_{\bar{\mathcal{G}}}$ a predicate, as above, with respect to such $\bar{\mathcal{G}}$. Let also $\mathcal{F}$ be a non-reactive functionality[13] which allows for fair evaluation of a given function (SFE) in the sense of [19], i.e., it has two modes of delivering output: (i) delayed delivery: $(\textsc{deliver}, \mathsf{sid}, m, P)$ signifying delayed output delivery[14] of m to party P, (ii) fair delivery: $(\textsc{fair-deliver}, \mathsf{sid}, (m, P_{i_1}), \ldots, (m, P_{i_k}), (m_{\mathcal{S}}, \mathcal{S}))$ that results in simultaneous delivery of outputs $m_{i_1}, \ldots m_{i_k}$ to parties $P_{i_1}, \ldots, P_{i_k}$ and output $m_{\mathcal{S}}$ to $\mathcal{S}$. We note that (G)UC does not have an explicit mechanism for simultaneous delivery of outputs. Thus, when we refer to simultaneous delivery of a vector $(m_{i_1}, \ldots, m_{i_k})$ to parties $P_{i_1}, \ldots, P_{i_k}$, respectively, we imply that the functionality prepares all the output to be delivered in a "fetch mode" as defined in [24]; that is:

- The functionality registers the pairs $(m_{i_1}, P_{i_1}), \ldots, (m, P_{i_k})$ as "ready to fetch" and sends the set $\{(m_{i_j}, P_{i_j}) | P_{i_j} \text{ is corrupted } \}$ to $\mathcal{S}$.
- Upon receiving an input $(\textsc{fetch-output}, P_i)$ from party P_i, if a message (m_i, P_i) has been registered as "ready to fetch" then remove it from the "ready to fetch" set and output it to P_i (if more than one such messages are registered, deliver and remove from the "ready to fetch" set the first, chronologically, registered such pair); otherwise send $(\textsc{fetch-output}, P_i)$ to $\mathcal{S}$.

4.1 $\mathsf{Q}_{\bar{\mathcal{G}}}$-Fairness

The wrapper functionality $\mathcal{W}$ that will be used in the definition of Q-fair (secure) computation is given in Figure 3. The intuition is as follows: Prior to handing inputs to the (wrapped) functionality $\mathcal{F}$, the parties can request the wrapper to generate on their behalf a *resource-setup* (by executing an associated resource-setup generation algorithm Gen) which allows them to update the global setup $\bar{\mathcal{G}}$; this resource setup consists of a public component $RS_{P,\mathsf{sid}}^{\mathsf{pub}}$ and a private

[12] We note that whenever it is clear from the context we may drop the subscript $\bar{\mathcal{G}}$ in $\mathsf{Q}_{\bar{\mathcal{G}}}, \mathsf{Q}_{\bar{\mathcal{G}}}^{\mathrm{Init}}, \mathsf{Q}_{\bar{\mathcal{G}}}^{\mathrm{Dlv}}, \mathsf{Q}_{\bar{\mathcal{G}}}^{\mathrm{Abt}}$.

[13] A non-reactive functionality does not accept any input from honest parties after generating output.

[14] Delayed output delivery is a standard (G)UC mechanism where the adversary is allowed to schedule the output at a time of its choosing.

component $RS^{\texttt{priv}}_{P,sid}$. [15] Both these values are given to the simulator, and the public component is handed to the party.

From the point when parties receive their inputs the Q predicate is used as a filter to specify the wrapper's behavior and add the fairness guarantees. More concretely, upon receiving an input from a party, the wrapper checks on the global setup to ensure that $\mathsf{Q}^{\mathrm{Init}}$ is true, and if it is not true it aborts (i.e., sets all honest parties' outputs to $\perp$ and blocks any communication between $\mathcal{F}$ and the adversary). This means that if the environment has not set up the experiment properly,[16] then the experiment will not be executed and the wrapped functionality will become useless. This formally resolves the question "What happens if some party does not have sufficient coins to play the protocol?" which leads to some ambiguity in existing bitcoin-based definitions of computation with fair compensation [7].

The predicates $\mathsf{Q}^{\mathrm{Dlv}}$ and $\mathsf{Q}^{\mathrm{Abt}}$ are used to filter out attempts of the simulator to deliver outputs or abort when $\mathsf{Q}^{\mathrm{Dlv}}$ and $\mathsf{Q}^{\mathrm{Abt}}$ are violated.[17] Concretely, any such attempt will be ignored if the corresponding predicate is not satisfied.

Intuitively, by requiring the protocol to implement such a wrapped version of a functionality, we will ensure that the parties might only abort if $\mathsf{Q}^{\mathrm{Abt}}$ is true, and might output a valid (non-$\perp$) value if $\mathsf{Q}^{\mathrm{Dlv}}$. As we shall see in Section 4.2, by a trivial modification of the fairness wrapper, we can capture a stronger property which we will call Q-robustness; the latter, roughly, guarantees that honest parties which start the protocol will either receive their output (and $\mathsf{Q}^{\mathrm{Dlv}}$ being true) or will abort and increase their revenue. (I.e., there is no way for the adversary to make the protocol abort after the first honest party has sent its first input-dependent message).

Definition 1. *We say protocol π realizes functionality $\mathcal{F}$ with $\mathsf{Q}_{\bar{\mathcal{G}}}$-fairness with respect to global functionality $\bar{\mathcal{G}}$, provided the following statement is true. For all adversaries $\mathcal{A}$, there is a simulator $\mathcal{S}$ so that for all environments $\mathcal{Z}$ it holds:*

$$\textsc{Exec}^{\bar{\mathcal{G}}}_{\pi,\mathcal{A},\mathcal{Z}} \approx \textsc{Exec}^{\bar{\mathcal{G}},\mathcal{W}_{\mathsf{Q},\bar{\mathcal{G}}}(\mathcal{F})}_{\mathcal{S},\mathcal{Z}}$$

More generally, the protocol σ realizes $\mathcal{H}$ with $\mathsf{Q}'_{\bar{\mathcal{G}}}$ fairness using a functionality $\mathcal{F}$ with fairness $\mathsf{Q}_{\bar{\mathcal{G}}}$ provided that for all adversaries $\mathcal{A}$, there is a simulator $\mathcal{S}$ so that for all environments $\mathcal{Z}$, it holds:

$$\textsc{Exec}^{\bar{\mathcal{G}},\mathcal{W}_{\mathsf{Q},\bar{\mathcal{G}}}(\mathcal{F})}_{\pi,\mathcal{A},\mathcal{Z}} \approx \textsc{Exec}^{\bar{\mathcal{G}},\mathcal{W}_{\mathsf{Q}',\bar{\mathcal{G}}}(\mathcal{H})}_{\mathcal{S},\mathcal{Z}}$$

[15] In the case of bitcoin-like ledgers these will correspond to a wallet (public-key) and a corresponding secret key.

[16] In the case of a bitcoin-ledger this corresponds to the environment not transferring to some protocol-related wallet sufficient funds to execute the protocol.

[17] As we will see, in bitcoin-like instantiations, $\mathsf{Q}^{\mathrm{Dlv}}$ will be satisfied when no honest party has a negative balance, and $\mathsf{Q}^{\mathrm{Abt}}$ will be satisfied when every honest party has a (strictly) positive balance.

Wrapper Functionality $\mathcal{W}_{\mathsf{Q},\bar{\mathcal{G}}}(\mathcal{F})$

The wrapper $\mathcal{W}_{\mathsf{Q},\bar{\mathcal{G}}}(\mathcal{F})$ interacts with a set of parties $\mathcal{P} = \{P_1, \ldots, P_n\}$, the adversary $\mathcal{S}$ and the environment $\mathcal{Z}$, as well as shared functionality $\bar{\mathcal{G}}$. It is parameterized with a predicate $\mathsf{Q} = (\mathsf{Q}^{\mathrm{Init}}, \mathsf{Q}^{\mathrm{Dlv}}, \mathsf{Q}^{\mathrm{Abt}})$ and a resource-setup generating algorithm $\mathrm{Gen} : 1^* \xrightarrow{\$} (\{0,1\}^*)^2$ and wraps any given non-reactive n-party functionality $\mathcal{F}$ with the two output-delivery modes (delayed and fair) described in Section 4.1. The functionality also keeps an indicator bit b, initially set to 0, indicating whether or not $\mathcal{S}$ is blocked from sending messages to $\mathcal{F}$.

- *Allocating Resources.* Upon receiving $(\textsc{alocate}, \mathsf{sid})$ from a party P, if a message $(\textsc{alocate}, \mathsf{sid})$ has already been received for P then ignore it; else send $(\textsc{Coins}, \mathsf{sid}, P)$ to $\mathcal{S}$ and upon receiving $(\textsc{Coins}, \mathsf{sid}, P, r)$ from $\mathcal{S}$ compute $(RS^{\mathsf{pub}}_{P,\mathsf{sid}}, RS^{\mathtt{priv}}_{P,\mathsf{sid}}) \leftarrow \mathrm{Gen}(1^\kappa; r)$ and sends a delayed output $(\textsc{deliver}, \mathsf{sid}, RS^{\mathsf{pub}}_{P,\mathsf{sid}}, P)$ to P.
- Upon receiving any message M from $\mathcal{F}$ to be delivered to its simulator, if $b = 0$ forward M to $\mathcal{S}$.
- Upon receiving a message $(\textsc{Forward}, M)$ from $\mathcal{S}$, if $b = 0$ then forward M to $\mathcal{F}$ as a message coming from its simulator.
- *Receiving input for $\mathcal{F}$.* Upon receiving $(\textsc{input}, \mathsf{sid}, x)$ from a party P, send $\textsc{read}$ to $\bar{\mathcal{G}}$, denote the response by trans and if $\neg\mathsf{Q}^{\mathrm{Init}}(RS^{\mathsf{pub}}_{P,\mathsf{sid}}, \mathsf{trans})$ then set $b := 1$ and issue a message $(\textsc{fair-deliver}, \mathsf{sid}, (\perp, P_1), \ldots, (\perp, P_n), (\perp, \mathcal{S}))$ (i.e., simultaneously deliver $\perp$ to all parties and ignore all future messages except $(\textsc{fetch-output}, \cdot)$ messages. Otherwise, forward $(\textsc{input}, \mathsf{sid}, x)$ to $\mathcal{F}$ as input for P.
- *Generating delayed output.* Upon receiving a message from $\mathcal{F}$ marked $(\textsc{deliver}, \mathsf{sid}, m, P)$ forwards m to party P via delayed output.
- *Registering fair output.* Upon receiving a message from $\mathcal{F}$ that is marked for fair delivery $(\textsc{fair-deliver}, \mathsf{sid}, \mathsf{mid}, (m_1, P_{i_1}), \ldots, (m_k, P_{i_k}), (m_{\mathcal{S}}, \mathcal{S}))$, it forwards $(\mathsf{mid}, P_{i_1}, \ldots, P_{i_k}, m_{\mathcal{S}})$ to $\mathcal{S}$.
- Q*-fair delivery.* Upon receiving $(\textsc{Q-deliver}, \mathsf{sid}, \mathsf{mid})$ from $\mathcal{S}$ then provided that a message $(\mathsf{mid}, \ldots)$ has been delivered to $\mathcal{S}$ operate as follows. For each pair of the form (m, P) associated with mid: Let $\mathrm{L} = \{(m, P) |\ P \text{ is uncorrupted}\}$. Send $\{(m, P) | P \text{ is corrupted}\}$ to $\mathcal{S}$. (If some currently honest P becomes corrupted later on, remove (m, P) from sending and send (m, P) to $\mathcal{S}$.) Subsequently perform the following.
  - On input a message $(\textsc{deliver}, \mathsf{sid}, \mathsf{mid}, P)$ from $\mathcal{S}$, provided that the record mid contains the pair $(m, P) \in \mathrm{L}$, send $\textsc{read}$ to $\bar{\mathcal{G}}$, denote the response by trans and if $\neg\mathsf{Q}^{\mathrm{Dlv}}(\mathsf{sid}, P, RS^{\mathsf{pub}}_{P,\mathsf{sid}}, \mathsf{trans})$ then ignore the message. Else, remove (m, P) from L and register (m, P) as "ready to fetch".
  - On input a message $(\textsc{abort}, \mathsf{sid}, \mathsf{mid}, P)$ from $\mathcal{S}$, provided that the record mid contains the pair $(m, P) \in \mathrm{L}$, send $\textsc{read}$ to $\bar{\mathcal{G}}$, denote the response by trans and if $\neg\mathsf{Q}^{\mathrm{Abt}}(\mathsf{sid}, P, RS^{\mathsf{pub}}_{P,\mathsf{sid}}, \mathsf{trans})$ then ignore the message. Else, remove (m, P) from L and register $(\perp, P)$ as "ready to fetch".
- Upon receiving an input $(\textsc{fetch-output}, P)$ from party P, if a message (m, P) has been registered as "ready to fetch" then remove it from the "ready to fetch" set and output it to P_i (if more than one such messages are registered, deliver and remove from the "ready to fetch" set the first, chronologically, registered such pair); otherwise send $(\textsc{fetch-output}, P_i)$ to $\mathcal{S}$.

Fig. 3. The Q-Fairness wrapper functionality.

We note that, both protocol π and the functionality $(\mathcal{W}_{\mathsf{Q},\bar{\mathcal{G}}}(\mathcal{F}), \bar{\mathcal{G}})$ are with respect to the global functionality[18] $\bar{\mathcal{G}}$. By following the very similar proof idea in [11], we can prove the following lemma and theorem:

Lemma 1. *Let $\mathsf{Q}_{\bar{\mathcal{G}}}$ be a predicate with respect to global functionality $\bar{\mathcal{G}}$. Let π be a protocol that realizes the functionality $\mathcal{F}$ with $\mathsf{Q}_{\bar{\mathcal{G}}}$-fairness. Let σ be a protocol in $(\mathcal{W}_{\mathsf{Q},\bar{\mathcal{G}}}(\mathcal{F}), \bar{\mathcal{G}})$-hybrid world. Then for all adversaries $\mathcal{A}$, there is a simulator $\mathcal{S}$ so that for all environments $\mathcal{Z}$, it holds*

$$\text{EXEC}^{\bar{\mathcal{G}}}_{\sigma^{\pi},\mathcal{A},\mathcal{Z}} \approx \text{EXEC}^{\bar{\mathcal{G}},\mathcal{W}_{\mathsf{Q},\bar{\mathcal{G}}}(\mathcal{F})}_{\sigma,\mathcal{S},\mathcal{Z}}$$

Theorem 1. *Let $\mathsf{Q}_{\bar{\mathcal{G}}}$ and $\mathsf{Q}'_{\bar{\mathcal{G}}}$ be predicates with respect to global functionality $\bar{\mathcal{G}}$. Let π be a protocol that realizes the functionality $\mathcal{F}$ with $\mathsf{Q}_{\bar{\mathcal{G}}}$-fairness. Let σ be a protocol in $(\mathcal{W}_{\mathsf{Q},\bar{\mathcal{G}}}(\mathcal{F}), \bar{\mathcal{G}})$-hybrid world that realizes the functionality $\mathcal{H}$ with $\mathsf{Q}'_{\bar{\mathcal{G}}}$-fairness. Then for all adversaries $\mathcal{A}$, there is a simulator $\mathcal{S}$ so that for all environments $\mathcal{Z}$ it holds:*

$$\text{EXEC}^{\bar{\mathcal{G}}}_{\sigma^{\pi},\mathcal{A},\mathcal{Z}} \approx \text{EXEC}^{\bar{\mathcal{G}},\mathcal{W}_{\mathsf{Q}',\bar{\mathcal{G}}}(\mathcal{H})}_{\mathcal{S},\mathcal{Z}}$$

Is the ledger functionality sufficient for Q *fairness?* We will construct secure computation protocols based on the ledger functionality $\bar{\mathcal{G}}_{\text{LEDGER}}$ together with other trusted setups. We may wonder if we can construct secure computation protocol from $\bar{\mathcal{G}}_{\text{LEDGER}}$ only. The answer if negative. Indeed, we prove the following statement

Theorem 2. *Let $\mathsf{Q}_{\bar{\mathcal{G}}}$ be a predicate with respect to global functionality $\bar{\mathcal{G}} = \bar{\mathcal{G}}_{\text{LEDGER}}$. There exists no protocol in the $\bar{\mathcal{G}}_{\text{LEDGER}}$ hybrid world which realizes the commitment functionality $\mathcal{F}_{\text{COM}}$ with $\mathsf{Q}_{\bar{\mathcal{G}}}$ fairness.*

The proof idea is very similar to the well-known Canetti-Fischlin [12] impossibility proof and can be found in [25].

4.2 $\mathsf{Q}_{\bar{\mathcal{G}}}$-Robustness

The above wrapper $\mathcal{W}$ allows the simulator to delay delivery of messages arbitrarily. Thus, although the predicates do guarantee the promised notion of fairness, the resulting functionality lacks the other relevant property that we discussed in the introduction, namely robustness. In the following we define Q-robustness which will ensure that if any party starts executing the protocol on its input (i.e., the protocol does not abort due to lack of resources for some party), then every honest party is guaranteed to either receive its output without loosing revenue, or receive bottom and a compensation. This property can be obtained by modifying the wrapper $\mathcal{W}$ using an idea from [24] so that in addition to the global-setup-related guarantees induced by predicate Q, it also preserves the guaranteed termination property of the wrapped functionality.[19]

[18] In GUC framework [11], this is also called, $\bar{\mathcal{G}}$-subroutine respecting.

[19] That is, we want to ensure that if the functionality $\mathcal{F}$ has guaranteed termination then the wrapped functionality will also have guaranteed termination.

More concretely, in [24], a functionality was augmented to have guaranteed termination, by ensuring that given appropriately many activations (i.e., dummy inputs), from its honest interface, it computes its output.[20] In the same spirit, a wrapper which ensures Q-robustness is derived from $\mathcal{W}$ via the following modification: As soon as a fair-output is registered (i.e., upon the wrapper receiving $(\text{FAIR-DELIVER}, \mathsf{sid}, \mathsf{mid}, (m_1, P_{i_1}), \ldots, (m_k, P_{i_k}), (m_{\mathcal{S}}, \mathcal{S}))$ from its inner functionality) it initiates a counter $\lambda = 0$ and an indicator variable $\lambda_{i_j} := 0$ for each $P_{i_j} \in \{P_{i_1}, \ldots, P_{i_k}\}$; whenever a message is received from some $P_{i_j} \in \{P_{i_1}, \ldots, P_{i_k}\}$, the wrapper sets $\lambda_{i_j} := 1$ and does the following check: if $\lambda_{i_j} = 1$ for all $P_{i_j} \in \{P_{i_1}, \ldots, P_{i_k}\}$ then increase $\lambda := \lambda + 1$ and reset $\lambda_{i_j} = 0$ for all $P_{i_j} \in \{P_{i_1}, \ldots, P_{i_k}\}$. As soon as λ reaches a set threshold T, the wrapper simultaneously delivers each $((m_1, P_{i_k}), \ldots, (m_k, P_{i_k})$ (i.e., prepares them to be fetched) without waiting for the simulator and does not accept any inputs other than $(\text{FETCH-OUTPUT}, \cdot)$ from that point on. When this happens, we will say that the wrapper *reached its termination limit.* We denote by $\hat{\mathcal{W}}^T$ the wrapper from Figure 3 modified as described above. Note that the wrapper is parameterized by the termination threshold T.

The intuition why this modification ensures guaranteed termination is the same as in [24]: if the environment wishes the experiment to terminate, the it can make it terminate irrespective of the simulator's strategy. Thus a protocol which realizes such a wrapper should also have such a guaranteed termination (the adversary cannot stall the computation indefinitely.)

Definition 2. *We say protocol π realizes functionality $\mathcal{F}$ with $\mathsf{Q}_{\bar{\mathcal{G}}}$-robustness with respect to global functionality $\bar{\mathcal{G}}$, provided the following statement is true. There exists a threshold T such that for all adversaries $\mathcal{A}$, there is a simulator $\mathcal{S}$ so that for all environments $\mathcal{Z}$ it holds:*

$$\textsc{Exec}^{\bar{\mathcal{G}}}_{\pi,\mathcal{A},\mathcal{Z}} \approx \textsc{Exec}^{\bar{\mathcal{G}},\hat{\mathcal{W}}^T_{\mathsf{Q},\bar{\mathcal{G}}}(\mathcal{F})}_{\mathcal{S},\mathcal{Z}}.$$

Moreover, whenever the wrapper reaches its termination limit, then for the state trans *of the global setup $\bar{\mathcal{G}}$ upon termination it holds that* $\mathsf{Q}^{Dlv}_{\bar{\mathcal{G}}}(\mathsf{sid}, P, RS^{\mathsf{pub}}_{P,\mathsf{sid}}, \mathsf{trans})$ *for every party $P \in \mathcal{P}$.*

The composition theorems for Q-fairness from Section 4.1 can be adapted in a straight-forward manner to Q-robustness. The statements and proofs are as in the previous section and are omitted. We note in passing that since the wrapper $\hat{\mathcal{W}}$ is in fact a wrapper which restricts the behavior of $\mathcal{S}$ on top of the restrictions which are applied by the Q-fairness wrapper $\mathcal{W}$, a protocol which is Q-robustness is also Q-fair with respect to the same predicate Q.

4.3 Computation with Fair/Robust Compensation

We are now ready to instantiate the notion of Q-fairness with a compensation mechanism. For the case when $\bar{\mathcal{G}}$ corresponds to a Bitcoin-like ledger, e.g., $\bar{\mathcal{G}} =$

[20] Of course, the simulator needs to be given sufficiently many activation so that he can provide its own inputs and perform the simulation (please see [24] for details).

$\bar{\mathcal{G}}_{\text{LEDGER}}$, and $\mathsf{Q}_{\bar{\mathcal{G}}}$ provides compensation of c coins, where $c > 0$, in the case of an abort, the resource-setup generation algorithm Gen a pair of $(\mathtt{address}, \mathsf{sk})$ where $\mathtt{address}$ is a bitcoin address and sk is the corresponding secret-key and the predicate $\mathsf{Q}_{\bar{\mathcal{G}}}^{\mathsf{coin}} = (\mathsf{Q}_{\bar{\mathcal{G}}}^{\text{C-Init}}, \mathsf{Q}_{\bar{\mathcal{G}}}^{\text{C-Dlv}}, \mathsf{Q}_{\bar{\mathcal{G}}}^{\text{C-Abt}})$ operates as follows. On input a session ID sid, a party id P, a wallet address $RS_{P,\mathsf{sid}}^{\mathsf{pub}}$, and a string trans which is parsed as a bitcoin ledger that contains transactions:[21]

- $\mathsf{Q}_{\bar{\mathcal{G}}}^{\text{C-Init}}$ outputs true if and only if the balance of all transactions (both incoming and outgoing) that concern $RS_{P,\mathsf{sid}}^{\mathsf{pub}}$ in trans and carry the meta-data sid is higher than a fixed pre-agreed initialization amount.[22]
- $\mathsf{Q}_{\bar{\mathcal{G}}}^{\text{C-Dlv}}$ outputs true if and only if the balance of all transactions (both incoming and outgoing) that concern $RS_{P,\mathsf{sid}}^{\mathsf{pub}}$ in trans and carry the meta-data sid is greater or equal to 0.
- $\mathsf{Q}_{\bar{\mathcal{G}}}^{\text{C-Abt}}$ outputs true if and only if the balance of all transactions (both incoming and outgoing) that concern $RS_{P,\mathsf{sid}}^{\mathsf{pub}}$ in trans and carry the meta-data sid is greater or equal to a fixed pre-agreed compensation amount.

If a protocol π realizes a functionality $\mathcal{F}$ with $\mathsf{Q}_{\bar{\mathcal{G}}}^{\mathsf{coin}}$-fairness (resp. $\mathsf{Q}_{\bar{\mathcal{G}}}^{\mathsf{coin}}$-robustness), i.e., with respect to the global functionality $\bar{\mathcal{G}}_{\text{LEDGER}}$, we say that π realizes $\mathcal{F}$ with fair compensation (resp. with robust compensation). Because our results are proved for $\mathsf{Q}_{\bar{\mathcal{G}}}^{\mathsf{coin}}$, to keep the notation simple in the remainder of the paper we might drop the superscript from $\mathsf{Q}_{\bar{\mathcal{G}}}^{\mathsf{coin}}$, i.e., we write Q or $\mathsf{Q}_{\bar{\mathcal{G}}}$ instead of $\mathsf{Q}_{\bar{\mathcal{G}}}^{\mathsf{coin}}$.

5 Our $\mathsf{Q}_{\bar{\mathcal{G}}}^{\mathsf{coin}}$-Robust Protocol Compiler

In this section we present our fair and robust protocol compiler. Our compiler compiles a synchronous protocol π_{SH} which is secure (i.e., private) against a corrupted majority in the semi-honest correlated randomness model (e.g, an OT-hybrid protocol where the OT's have been pre-computed) into a protocol π which is secure with fair-compensation in the malicious correlated randomness model. The high-level idea is the following: We first compile π_{SH} into a protocol in the malicious correlated randomness model, which is executed over a broadcast channel and is secure with publicly identifiable abort. (Roughly, this means that someone observing the protocol execution can decide, upon abort, which party is not executing its code.) This protocol is then transformed into a protocol with fair compensation as follows: Every party (after receiving his correlated randomness setup) posts to the ledger transactions that the other parties can claim only if they, later, post transactions that prove that they follow their protocol. Transactions that are not claimed this way are returned to the source

[21] Transactions in trans can also be marked with metadata.

[22] In our construction $\mathsf{Q}_{\bar{\mathcal{G}}}^{\text{C-Init}}$ will check additional properties for the initial set of transactions that concern $RS_{P,\mathsf{sid}}^{\mathsf{pub}}$; specifically, not only that a fixed amount μ is present but also that it is distributed in a special way.

address; thus, if some party does not post such a proof it will not be able to claim the corresponding transaction, and will therefore leave the honest parties with a positive balance as their transactions will be refunded. Observe that these are not standard Bitcoin transactions, but they have a special format which is described in the following.

Importantly, the protocol we describe is guaranteed to either produce output in as many (Bitcoin) rounds as the rounds of the original malicious protocol, or to compensate all honest parties. This *robustness* property is achieved by a novel technique which ensures that once the honest parties make their initial transaction, the adversary has no way of preventing them from either computing their output or being compensated. Informally, our technique consists of splitting the parties into "islands" depending on the transactions they post (so that all honest parties are on the same island) and then allowing them to either compute the function within their island, or if they abort to get compensated. (The adversary has the option of being included or not in the honest parties' island.)

5.1 MPC with Publicly Identifiable Abort

As a first step in our compiler we invoke the semi-honest to malicious with identifiable abort compiler of Ishai, Ostrovsky, and Zikas [23] (hereafter referred to as the *IOZ compiler*). This compiler takes a semi-honest protocol $\pi_{\mathtt{SH}}$ in the correlated randomness model and transforms it to a protocol in the malicious correlated randomness model (for an appropriate setup) which is secure with identifiable abort, i.e., when it aborts, every party learns the identity of a corrupted party. The compiler in [23] follows the so called GMW paradigm [21], which in a nutshell has every party commit to its input and randomness for executing the semi-honest protocol $\pi_{\mathtt{SH}}$ and then has every party run $\pi_{\mathtt{SH}}$ over a broadcast channel, where in each round ρ every party broadcasts his round ρ messages and proves in zero-knowledge that the broadcasted message is correct, i.e., that he knows the input and randomness that are consistent with the initial commitments and the (public) view of the protocol so far. The main difference of the IOZ compiler and the GMW compiler is that the parties are not only committed to their randomness, but they are also committed to their entire setup string, i.e., their private component of the correlated randomness. In the following, for the sake of completeness, we enumerate some key properties of the resulting maliciously secure protocol $\pi_{\mathtt{Mal}}$ (which is based on the compiler in [23]) that will be important for our construction:

- Every party is committed to his setup, i.e., the part of the correlated randomness it holds. That is, every party P_i receives from the setup his randomness (which we refer to as P_i's *private component* of the setup) along with one-to-many commitments[23] on the private components of all parties. Without loss of generality, we also assume that a common-reference string (CRS) and a

[23] These are commitments that can be opened so that every party agrees on whether or not the opening succeeded.

public-key infrastructure (PKI) are included in every party's setup. We refer to the distribution of this correlated randomness as $\mathcal{D}_{\texttt{Mal}}$.

- The protocol $\pi_{\texttt{Mal}}$ uses *only* the broadcast channel for communication.
- Given the correlated randomness setup, the protocol $\pi_{\texttt{Mal}}$ is completely deterministic. This is achieved in [23] by ensuring that all the randomness used in the protocol, even the one needed for the zero-knowledge proofs, is part of the private components that are distributed by the sampling functionality.[24]
- $\pi_{\texttt{Mal}}$ starts off by having every party broadcast a one-time pad encryption of its input with its (committed) randomness and a NIZK that it knows the input and randomness corresponding to the broadcasted message.
- By convention, the next-message function of $\pi_{\texttt{Mal}}$ is such that if in any round the transcript seen by a party is an aborting transcript, i.e., is not consistent with an accepting run of the semi-honest protocol, then the party outputs $\perp$. Recall that the identifiable abort property ensures that in this case every party will also output the identity of a malicious party (the same for all parties).
- There is a (known) upper bound on the number ρ_{c} of rounds of $\pi_{\texttt{Mal}}$.

We remark that, given appropriate setup, the IOZ-compiler achieves information-theoretic security, and needs therefore to build information-theoretic commitments and zero-knowledge proofs. As in this work we are only after computational security, we modify the IOZ compiler so that we use (computationally) UC secure one-to-many commitments [14] and computationally UC secure non-interactive zero-knowledge proofs (NIZKs) instead if their information-theoretic instantiation suggested in [23]. Both the UC commitment and the NIZKs can be built in the CRS model. Moreover, the use of UC secure instantiations of zero-knowledge and commitments ensures that the resulting protocol will be (computationally) secure.

Using the setup within a subset of parties. A standard property of many protocols in the correlated-randomness model is that once the parties in $\mathcal{P}$ have received the setup, any subset $\mathcal{P}' \subset \mathcal{P}$ is able to use it to perform a computation of a $|\mathcal{P}'|$-party function amongst them while ignoring parties in $\mathcal{P} \setminus \mathcal{P}'$. More concretely, assume the parties in $\mathcal{P}$ have been handed a setup allowing them to execute some protocol π for computing any $|\mathcal{P}|$-party function f; then for any $\mathcal{P}' \subseteq \mathcal{P}$, the parties in $\mathcal{P}'$ can use their setup within a protocol $\pi|_{\mathcal{P}'}$ to compute any $|\mathcal{P}'|$-party function $f|_{|\mathcal{P}'|}$. This property which will prove very useful for obtaining computation with robustness or compensation, is also satisfied by the IOZ protocol, as the parties in $\mathcal{P}'$ can simply ignore the commitments (public setup component) corresponding to parties in $\mathcal{P} \setminus \mathcal{P}'$. It should be noted that this is not an inherent property of the correlated randomness model: e.g., protocols based on threshold encryption do not immediately satisfy this property (as players would have to readjust the threshold).

[24] As an example, the challenge for the zero-knowledge proofs is generated by the parties opening appropriate parts of their committed random strings.

Making Identifiability Public. The general idea of our protocol is to have every party issue transactions by which he commits to transferring a certain amount of coins per party for each protocol round. All these transactions are issued at the beginning of the protocol execution. Every party can claim the "committed" coins transferred to him associated to some protocol round ρ only under the following conditions: (1) the claim is posted in the time-interval corresponding to round ρ; (2) the party has claimed all his transferred coins associated to the previous rounds; and (3) the party has posted a transaction which includes his valid protocol message for round ρ.

In order to ensure that a party cannot claim his coins unless he follows the protocol, the ledger (more concretely the validation predicate) should be able to check that the party is indeed posting its valid next message. In other words, in each round ρ, P_i's round-ρ message acts as a witness for P_i claiming all the coins committed to him associated with this round ρ. To this direction we make the following modification to the protocol: Let $f(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$ denote the n-party function we wish to compute, and let f^{+1} be the $(n+1)$-party function which takes input x_i from each P_i, $i \in [n]$, and no input from P_{n+1} and outputs y_i to each P_i and a special symbol (e.g., 0) to P_{n+1}. Clearly, if $\pi_{\texttt{SH}}$ is a semi-honest n-party protocol for computing f over broadcast, then the $n+1$ protocol $\pi_{\texttt{SH}}^{+1}$ (in which every P_i with $i \in [n]$ executes $\pi_{\texttt{SH}}$ and P_{n+1} simply listens to the broadcast channel and outputs 0) is a semi-honest secure protocol for f^{+1}.

Now if $\pi_{\texttt{Mal}}^{+1}$ denotes the $(n+1)$-party malicious protocol which results by applying the above modified IOZ compiler on the $(n+1)$-party semi-honest protocol $\pi_{\texttt{SH}}^{+1}$ for computing the function f^{+1}, then, by construction this protocol computes function f^{+1} with identifiable abort and has the following additional properties:

- Party P_{n+1} does not make any use of his private randomness whatsoever; this is true because he broadcasts no messages and simply verifies the broadcasted NIZKs.
- If some party P_i, $i \in [n]$ deviates from running $\pi_{\texttt{SH}}$ with the correlated (committed) randomness as distributed from the sampling functionality, then this is detected by all parties, including P_{n+1} (and protocol $\pi_{\texttt{Mal}}^{+1}$ aborts identifying P_i as the offender). This follows by the soundness of the NIZK which P_i needs to provide proving that he is executing $\pi_{\texttt{SH}}$ in every round.

Due to P_{n+1}'s role as an observer who gets to decide if the protocol is successful (P_{n+1} outputs 0) or some party deviated (P_{n+1} observes that the corresponding NIZK verification failed) in the following we will refer to P_{n+1} in the above protocol as the *judge*. The code of the judge can be used by anyone who has the public setup and wants to follow the protocol execution and decide whether it should abort or not given the parties' messages. Looking ahead, the judge's code in the protocol will be used by the ledger to decide wether or not a transaction that claims some committed coins is valid.

5.2 Special Transactions supported by our Ledger

In this section we specify the Validate and the Blockify predicates that are used for achieving our protocol's properties. More specifically, our protocol uses the following type of transactions which transfer v coins from wallet $\mathtt{address}_i$ to wallet $\mathtt{address}_j$ conditioned on a statement Σ:

$$\mathbb{B}_{v,\mathtt{address}_i,\mathtt{address}_j,\Sigma,\mathtt{aux},\sigma_i,\tau} \tag{1}$$

where σ_i is a signature of the transaction, which can be verified under wallet $\mathtt{address}_i$; τ is the time-stamp, i.e., the current value of the clock when this transaction is added to the state by the ledger—note that this timestamp is added by the ledger and not by the users,—$\mathtt{aux} \in \{0,1\}^*$ is an arbitrary string[25]; and the statement Σ consists of three arguments, i.e., $\Sigma = (\mathtt{arg1}, \mathtt{arg2}, \mathtt{arg3})$, which are processed by the Validate predicate in order to decide if the transaction is valid (i.e., if it will be included in the ledger's next block).

The Validate *predicate.* The validation happens by processing the arguments of Σ in a sequential order, where if while processing of some argument the validation rejects, algorithm Validate stops processing at that point and this transaction is dropped. The arguments are defined/processed as follows:

Time-Restrictions: The first argument is a pair $\mathtt{arg1} = (\tau_-, \tau_+) \in \mathbb{Z} \times (\mathbb{Z}^+ \cup \{\infty\})$ of points in time. If $\tau_- > \tau_+$ then the transaction is invalid (i.e., it will be dropped by the ledger). Otherwise, before time τ_- the coins in the transaction "remain" blocked, i.e., no party can spend them; from time τ_- until time τ_+, the money can be spent by the owner of wallet $\mathtt{address}_j$ provided that the spending statement satisfies also the rest of the requirements/arguments in statement Σ (listed below). After time τ_+ the money can be spent by the owner of wallet $\mathtt{address}_i$ without any additional restrictions (i.e., the rest of the arguments in Σ are not parsed). As a special case, if $\tau_+ = \infty$ then the transferred coins can be spent from $\mathtt{address}_j$ at any point (provided the spending statement is satisfied); we say then that the transaction is *time-unrestricted*,[26] otherwise we say that the transaction is *time restricted.*

Spending Link: Provided that the processing of the first argument, as above, was not rejecting, the Validate predicate proceeds to the second argument, which is a unique "anchor", $\mathtt{arg2} = \alpha \in \{0,1\}^*$. Informally, this serves as a unique identifier for linked transactions[27]; that is, when $\alpha \neq \perp$, then the Validate algorithm of the ledger looks in the ledger's state and buffer to confirm that the balance of transactions to/from the wallet address $\mathtt{address}_i$

[25] This string will be included to the Ledger's state as soon as the transaction is posted and can be, therefore, referred to by other spending statements.

[26] This is the case with standard Bitcoin transactions.

[27] Looking ahead arg2 will be used to point to specific transactions of a protocol instance. The mechanism may be simulated by generating multiple addresses however it is more convenient for the protocol description and for this reason we adopt it.

with this anchor `arg2` is at least $v' \geq v$ coins. That is, the sum of coins in the state or in the buffer with receiver address $\texttt{address}_i$ and anchor `arg2` minus the sum of coins in the state or in the buffer with sender address $\texttt{address}_i$ and anchor `arg2` is greater equal to v. If this is not the case then the transaction is rendered invalid; otherwise the validation of this argument succeeds and the algorithm proceeds to the next argument.

State-Dependent Condition: The last argument to be validated is `arg3`, which is a relation $\mathcal{R} : \mathcal{S} \times \mathcal{B} \times \mathcal{T} \to \{0,1\}$, where $\mathcal{S}$, $\mathcal{B}$, and $\mathcal{T}$ are the domains of possible ledger-states, ledger-buffers, and transactions, respectively (in a given encoding). This argument defines which type of transactions can spend the coins transferred in the current transaction. That is, in order to spend the coins, the receiver needs to submit a transaction $\texttt{tx} \in \mathcal{T}$ such that $\mathcal{R}(\mathsf{state}, \mathsf{buffer}, \texttt{tx}) = 1$ at the moment when `tx` is to be validated and inserted in the buffer. In our construction this is the part of the transaction that we will take advantage to detect cheating (and thus $\mathcal{R}$ will encode a NIZK verifier etc.).

We point out that as with standard Bitcoin transactions, the validation predicate will always also check validity of the signature σ_i with respect to the wallet $\texttt{address}_i$. Moreover, the standard Bitcoin-like transactions can be trivially casted as transactions of the above type by setting $\alpha = \perp$ and $\Sigma = ((0,\infty), \perp, \mathcal{R}_\emptyset)$, where $\mathcal{R}_\emptyset$ denotes the relation which is always true.

To simplify the structure of our special transactions and ease their implementation, we impose the following additional constraints: whenever a time-restriction is given, i.e., $\text{arg}_1 = (\tau_-, \tau_+)$ then it must be that $\alpha \neq \perp$. Furthermore, if a time-restricted transaction is present with anchor α from $\texttt{address}_1$ to $\texttt{address}_2$, the only transactions that are permitted with anchor α in the ledger would be time-unrestricted transactions originating from either $\texttt{address}_2$ within the specified time-window, or $\texttt{address}_1$ after the specified time window.

The $\mathsf{Blockify}$ *algorithm.* This algorithm simply groups transactions in the current buffer and adds a timestamp from the current round. We choose to ignore any additional functionality (e.g., such as a reward mechanism for mining that is present in typical cryptocurrencies — however such mechanism can be easily added independently of our results).

5.3 The Protocol

Let $\pi_{\texttt{Mal}}^{+1}$ denote the protocol described in section 5.1. Let $\texttt{Round2Time}(1)$ denote the time in which the parties have agreed to start the protocol execution. Without loss of generality we assume that $\texttt{Round2Time}(1) > \texttt{T} + 1$ where `T` is the number clock ticks for each block generation cf. Figure 2.[28] Furthermore, for simplicity, we assume that each party P_i receives its input x_i with its first activation from the environment at time $\texttt{Round2Time}(1)$ (if some honest party

[28] That is we assume that at least one ledger rounds plus one extra clock-ticks have passed from the beginning of the time.

does not have an input by that time it will execute the protocol with a default input, e.g., 0).

Informally, the protocol proceeds as follows: In a pre-processing step, before the parties receive input, the parties invoke the sampling functionality for $\pi^{+1}_{\texttt{Mal}}$ to receive their correlated randomness.[29] The public component of this randomness includes their protocol-associated wallet $\texttt{address}_i$ which they output (to the environment). This corresponds to the resources allocation step in the Q-robustness wrapper $\hat{\mathcal{W}}$. The environment is then expected to submit ρ_{c} special (as above) transactions for each pair of parties $P_i \in \mathcal{P}$ and $P_j \in \mathcal{P}$; the source wallet-address for each such transaction is P_i's, i.e., $\texttt{address}_i$ and the target wallet-address for is P_j's, i.e., $\texttt{address}_j$, and the corresponding anchors are as follows: $\alpha_{i,j,\rho} = (\mathsf{pid}, i, j, \rho)$, for $(i, j, \rho) \in [n]^2 \times [\rho_{\mathsf{c}}]$, where[30] pid is the (G)UC protocol ID for $\pi^{+1}_{\texttt{Mal}}$. Since by assumption, $\texttt{Round2Time}(1) > \texttt{T} + 1$, the environment has sufficient time to submit these transaction so that by the time the protocol starts they have been posted on the ledger.

At time $\texttt{Round2Time}(1)$ the parties receive their inputs and initiate the protocol execution by first checking that sufficient funds are allocated to their wallets linked to the protocol executions by appropriate anchors, as above. If some party does not have sufficient funds then it broadcasts an aborting message and all parties abort.[31] This aborting in case of insufficient funds is consistent with the behavior of the wrapper $\hat{\mathcal{W}}$ when $\mathsf{Q}^{\text{C-Init}}_{\bar{\mathcal{G}}}$ is false. Otherwise, parties make the special transactions that commit them (see below) into executing the protocol, and then proceed into claiming them one-by-one by executing their protocol in a round-by-round fashion.

Note that each protocol round lasts one ledger round so that the parties have enough time to claim their transactions. This means that $\texttt{Round2Time}(i + 1) - \texttt{Round2Time}(i) \geq \texttt{T}$, which guarantees that any transaction submitted for round ρ, $\rho = 1, \ldots, \rho_{\mathsf{c}} - 1$, of the protocol, has been posted on the ledger by the beginning of round $\rho + 1$. Observe that by using a constant round protocol $\pi^{+1}_{\texttt{Mal}}$ (e.g., the modified compiled protocol from [23] instantiated with a constant round semi-honest protocol) we can ensure that our protocol will terminate in a constant number of ledger rounds and every honest party will either receive its input, or will have a positive balance in its wallet.

Remark 1 (On availability of funds). Unlike existing works, we choose to explicitly treat the issue of how funds become available to the protocol by making the off-line transfers external to the protocol itself (i.e., the environment takes care of them). However, the fact that the environment is in charge of "pouring" money into the wallets that are used for the protocol does not exclude that the parties might be actually the ones having done so. Indeed, the environment's

[29] In an actual application, the parties will use an unfair protocol for computing the correlated randomness. As this protocol has no inputs, an abort will not be unfair (i.e., the simulator can always simulate the view of the adversary in an aborting execution.)

[30] Recall that we assume $|\mathcal{P}| = n$.

[31] Note that this is a fair abort, i.e., no party has spent time into making transactions.

goal is to capture everything that is done on the side of, before, or after the protocol, including other protocols that the parties might have participated in. By giving the environment enough time to ensure these transactions are posted we ensure that some honest party not having enough funds corresponds to an environment that makes the computation abort (in a fair way and only in the pre-processing phase, before the parties have invested time into posting protocol transactions).

Here is how we exploit the power of our special transactions in order to arrange that the balance of honest parties is positive in case of an abort. We require that the auxiliary string of a transaction of a party P_j which claims a committed transaction for some round ρ includes his ρ-round protocol message. We then have the relation of this transaction be such that it evaluates to 1 if only if this is indeed P_j's next message. Thus, effectively the validate predicate implements the judge in $\pi_{\mathtt{Mal}}^{+1}$ and can, therefore, decide if some party aborted: if some party broadcasts a message that would make the judge abort, then the validate predicate drops the corresponding transaction and all claims for committed transactions corresponding to future rounds, thus, all other parties are allowed to reclaim their committed coins starting from the next round.

Before we give the protocol description there is a last question: how is the ledger able to know which parties should participate in the protocol? Here is the problem: The adversary might post in the first round (as part of the committing transaction for the first round) a fake, maliciously generated setup. Since the ledger is not part of the correlated randomness sampling, it would be impossible to decide which is the good setup. We solve this issue by the following technique that is inspired by [6]: The ledger[32] groups together parties that post the same setup; these parties form "islands", i.e, subsets of $\mathcal{P}$. For each such subset $\mathcal{P}' \subseteq \mathcal{P} \cup \{P_{n+1}\}$ which includes the judge P_{n+1}, the ledger acts as if the parties in $\mathcal{P}'$ are executing the protocol $\pi_{\mathtt{Mal}}^{+1}|_{\mathcal{P}'}$ (which, recall, is the restriction of $\pi_{\mathtt{Mal}}^{+1}$ to the parties in $\mathcal{P}'$) for computing the $|\mathcal{P}'|$-party function $f^{+1}|_{\mathcal{P}'}(\boldsymbol{x})$ defined as follows: let the function to be computed be $f(\boldsymbol{x})$, where $\boldsymbol{x} = (x_1, \ldots, x_n)$, and f^{+1} be as above, then $f^{+1}|_{\mathcal{P}'}(\boldsymbol{x}) = f^{+1}(\boldsymbol{x}_{\mathcal{P}'})$ where $\boldsymbol{x}_{\mathcal{P}'} = (x'_1, \ldots, x'_n)$ with $x'_i = x_i$ for $P_i \in \mathcal{P}'$ and x'_i being a default value for every $P_i \notin \mathcal{P}'$. This solves the problem as all honest parties will be in the same island $\mathcal{P}' \subset \mathcal{P}$ (as they will all post the same value for public randomness); thus if the adversary chooses not to post this value on behalf of some corrupted party, he is effectively setting this party's input to a default value, a strategy which is easily simulatable. (Of course, the above solution will allow the adversary to also have "islands" of only corrupted parties that might execute the protocol, but this is also a fully simulatable strategy and has no effect on fair-compensation whatsoever—corrupted parties are not required to have a positive balance upon abort).

The final protocol $\pi_{\mathtt{Mal}}^{\mathbb{B}}$ is detailed in the following. The protocol ID is sid. The function to be computed is $f(x_1, \ldots, x_n)$. The protocol parties are $\mathcal{P} =$

[32] Throughout the following description, we say that the ledger does some check to refer to the process of checking a corresponding relation, as part of validating a special transaction.

$\{P_1, \ldots, P_n\}$. We assume all parties have registered with the clock functionality in advance and are therefore synchronized once the following steps start.

Phase 1: Setup Generation

Time $\tau_{-2} = \texttt{Round2Time}(1) - \texttt{T} - 2$:

The parties invoke the sampling functionality for $\mathcal{D}_{\texttt{Mal}}$, i.e., every party $P_i \in \mathcal{P}$ starts off by sending the sampling functionality a message $(\textsc{request}, \mathsf{sid})$; the sampling functionality returns $(R_i^{\texttt{priv}}, R^{\mathsf{pub}})$ to P_i where $R_i^{\texttt{priv}}$ is P_i's private component (including all random coins he needs to run the protocol, along with his signing key sk_i) of the setup and R^{pub} is the public component (the same for every party P_j) which includes the vector of UC commitments $(\mathsf{Com}_1, \ldots, \mathsf{Com}_n)$, where for $j \in [n]$, Com_j is a commitment to $R_j^{\texttt{priv}}$, along with a vector of public (verification) keys $(\mathsf{vk}_1, \ldots, \mathsf{vk}_n)$ corresponding to the signing keys $(\mathsf{sk}_1, \ldots, \mathsf{sk}_n)$ and a common reference string CRS. Every party outputs its own public key, as its wallet address for the protocol, i.e., $\texttt{address}_i = \mathsf{vk}_i$.

Phase 2: Inputs and Protocol Execution

Time $\tau_{-1} = \texttt{Round2Time}(1) - 1$:

Every party $P_i \in \mathcal{P}$ receives its input x_i ($x_i = 0$ if no input is received in the first activation of P_i for time $\texttt{Round2Time}(1)$) and does the following to check that it has sufficient fund available: P_i reads the current state from the ledger. If the state does not include for each $(i, j, \rho) \in [n]^2 \times [\rho_c]$ a transaction $\mathbb{B}_{c,\texttt{address},\texttt{address}_i,\Sigma^0_{i,j,\rho},\texttt{aux}^0_{i,j,\rho},\sigma,\tau}$, for some arbitrary $\texttt{address}$ and where $\Sigma^0_{i,j,\rho} = ((0, \infty), (\mathsf{sid}, i, j, \rho), \mathcal{R}_\emptyset)$ then P_i broadcasts $\perp$ and every party aborts the protocol execution with output $\perp$ (i.e., no party does anything from that point on.Recall that ρ_c is the upper bound on the number of rounds of $\pi^{+1}_{\texttt{Mal}}$, cf. Section 5.1.

Time $\tau_0 = \texttt{Round2Time}(1)$:

Every P_i submits to the ledger the following "commitment" transactions:[33]

1. For each $P_j \in \mathcal{P} : \mathbb{B}_{c,\texttt{address}_i,\texttt{address}_j,\Sigma_{i,j,1},\texttt{aux}_{i,j,1},\sigma,\tau}$, where $\texttt{aux}_{i,j,1} = R^{\mathsf{pub}}$ and $\Sigma_{i,j,1} = (\texttt{arg1}_{i,j,1}, \texttt{arg2}_{i,j,1}, \texttt{arg3}_{i,j,1})$ with
   - $\texttt{arg1}_{i,j,1} = (\texttt{Round2Time}(1) + \texttt{T}, \texttt{Round2Time}(1) + 2\texttt{T} - 1)$
   - $\texttt{arg2}_{i,j,1} = (\mathsf{sid}, i, j, 1)$
   - $\texttt{arg3}_{i,j,1} = \mathcal{R}_{i,j,1}$ defined as follows: Let $\mathcal{P}^{+1} = \mathcal{P} \cup \{P_{n+1}\}$, where P_{n+1} denotes the judge, be the player set implicit in R^{pub}, [34] and let $\mathcal{P}_i^{+1} \subseteq \mathcal{P}^{+1}$ denote the island of party i including the judge, i.e., the set of parties (wallets), such that in the first block posted after time $\texttt{Round2Time}(1)$ all parties $P_k \in \mathcal{P}_i^{+1}$ have exactly one

[33] Recall that, by definition of the clock, every party has as much time as it needs to complete all the steps below before the clock advances time.

[34] Recall that R^{pub} includes commitments to all parties' private randomness (including the judge's P_d) used for running the protocol, which is an implicit representation of the player set.

transaction for every $P_j \in \mathcal{P}$ with $\mathtt{arg1}_{k,j,1} = (\mathtt{Round2Time}(1) + \mathtt{T}, \mathtt{Round2Time}(1) + 2\mathtt{T} - 1)$, $\mathtt{arg2}_{k,j,1} = (\mathsf{sid}, k, j, 1)$, and $\mathtt{aux}^1_{k,j,1} = R^{\mathsf{pub}}$. Furthermore, let $\pi^{+1}_{\mathtt{Mal}}|_{\mathcal{P}^{+1}_i}$ be the protocol with public identifiability for computing $f^{+1}|_{\mathcal{P}^{+1}_i}$, described above and denote by $R^{\mathsf{pub}}|_{\mathcal{P}^{+1}_i}$ the restriction of the public setup to the parties in $\mathcal{P}^{+1}_i$. Then $\mathcal{R}_{i,j,1}(\mathsf{state}, \mathsf{buffer}, \mathtt{tx}) = 1$ if and only if the protocol of the judge with public setup $R^{\mathsf{pub}}|_{\mathcal{P}^{+1}_i}$ accepts the auxiliary string $\mathtt{aux}_{\mathtt{tx}}$ in $\mathtt{tx}$ as P_i's first message in $\pi^{+1}_{\mathtt{Mal}}|_{\mathcal{P}^{+1}_i}$ (i.e., it does not abort in the first round).

2. For each protocol round $\rho = 2, \ldots, \rho_{\mathrm{c}}$ and each $P_j \in \mathcal{P}$: each party posts the transaction: $\mathbb{B}_{c,\mathtt{address}_i,\mathtt{address}_j,\Sigma_{i,j,\rho},\mathtt{aux}^1_{i,j,\rho},\sigma,\tau}$, where $\mathtt{aux}^1_{i,j,\rho} = R^{\mathsf{pub}}$ and $\Sigma_{i,j,\rho} = (\mathtt{arg1}, \mathtt{arg2}, \mathtt{arg3})$ with
   - $\mathtt{arg1} = (\mathtt{Round2Time}(\rho) + \mathtt{T}, \mathtt{Round2Time}(\rho + 1) + 2\mathtt{T} - 1)$
   - $\mathtt{arg2} = (\mathsf{sid}, i, j, \rho)$.
   - $\mathtt{arg3} = \mathcal{R}_{i,j,\rho}$ defined as follows: Let $\mathcal{P}^{+1}_i, \pi^{+1}_{\mathtt{Mal}}|_{\mathcal{P}^{+1}_i}$ be defined as above (and assume $\mathcal{P}^{+1}_i = \{P_{i_1}, \ldots, P_{i_m}\}$. Then $\mathcal{R}_{i,j,\rho}(\mathsf{state}, \mathsf{buffer}, \mathtt{tx}) = 1$ if and only if, for each $r = 1, \ldots, \rho - 1$ and each party $P_{i_k} \in \mathcal{P}^{+1}_i$, the state state includes transactions in which the auxiliary input is $\mathtt{aux}_{i_k,r}$ and the protocol of the judge with public setup $R^{\mathsf{pub}}|_{\mathcal{P}^{+1}_i}$, and transcript $(\mathtt{aux}_{i_1,1}, \ldots, \mathtt{aux}_{i_m,1}), \ldots, (\mathtt{aux}_{i_1,\rho-1}, \ldots, \mathtt{aux}_{i_m,\rho-1})$, accepts the auxiliary string $\mathtt{aux}$ in $\mathtt{tx}$ as P_i's next (ρ-round) message in $\pi^{+1}_{\mathtt{Mal}}|_{\mathcal{P}^{+1}_i}$ (i.e., it does not abort in the ρ-th round).

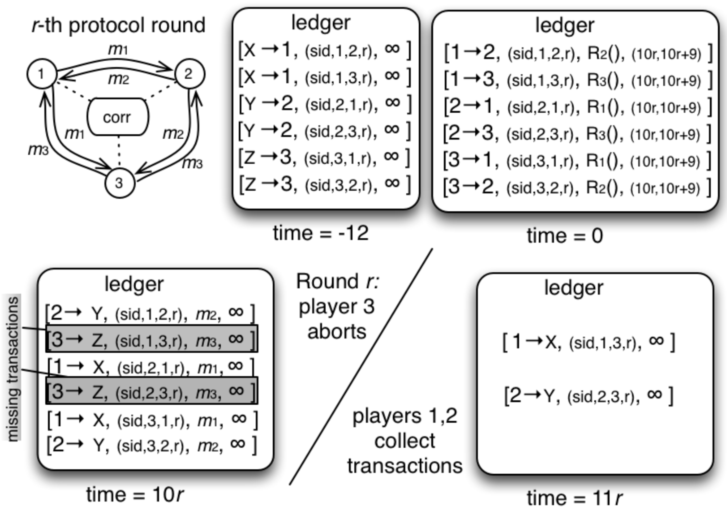
Phase 3: Claiming Committed Transactions/Executing the Protocol Time $\tau \geq \mathtt{Round2Time}(1)$:

For each $\rho = 1, \ldots, \rho_{\mathrm{c}}+1$, every P_i does the following at time $\mathtt{Round2Time}(\rho)$,:

1. If $\tau = \mathtt{Round2Time}(\rho_{\mathrm{c}} + 1)$ then go to Step 4; otherwise do the following:
2. Read the ledger's state, and compute $\mathcal{P}^{+1}_i, \pi^{+1}_{\mathtt{Mal}}|_{\mathcal{P}^{+1}_i}$ as above.
3. If the state state is not aborting for $\mathcal{P}^{+1}_i = \{P_{i_1}, \ldots, P_{i_m}\}$, i.e., it includes for each $r = 1, \ldots, \rho - 1$ and each party $P_{i_k} \in \mathcal{P}^{+1}_i$ a transaction in which the auxiliary input is $\mathtt{aux}_{i_k,r}$ such that P_i executing $\pi^{+1}_{\mathtt{Mal}}|_{\mathcal{P}^{+1}_i}$ with public setup $R^{\mathsf{pub}}|_{\mathcal{P}^{+1}_i}$, private setup $R^{\mathtt{priv}}_i$, and transcript $(\mathtt{aux}_{i_1,1}, \ldots, \mathtt{aux}_{i_m,1}), \ldots, (\mathtt{aux}_{i_1,\rho-1}, \ldots, \mathtt{aux}_{i_m,\rho-1})$ for the first $r - 1$ rounds does not abort, then compute P_i's message for round ρ, denoted as msg_ρ, and submit to the ledger for each $P_k \in \mathcal{P}^{+1}_i$ a transaction $\mathbb{B}_{c,\mathtt{address}_i,\mathtt{address},\Sigma'_{k,i,\rho},\mathtt{aux}^{\rho}_{k,i,\rho},\sigma,\tau}$, where $\mathtt{aux}^{\rho}_{k,i,\rho} = \mathrm{msg}_\rho$, $\mathtt{address}$ is the address that was the input of the first transaction with link $(\mathsf{sid}, i, k, \rho)$ and $\Sigma'_{k,i,\rho} = (\mathtt{arg1}, \mathtt{arg2}, \mathtt{arg3})$ instantiated as follows: $\mathtt{arg1} = (0, \infty)$; $\mathtt{arg2} = (\mathsf{sid}, k, i, \rho)$; $\mathtt{arg3} = \mathcal{R}_{\emptyset}$. For each such transaction posted enter $(\mathsf{sid}, k, i, \rho)$ in a set of "claimed" transactions CLAIM_i.
4. Otherwise, i.e., if the state state is aborting, then prepare for each round $r = 1, \ldots, \rho - 1$, and each $P_k \in \mathcal{P}$ a transaction by which the committed transaction towards P_k corresponding to round r is claimed

back to $\texttt{address}_i$, i.e., $\mathbb{B}_{c,\texttt{address}_k,\texttt{address}_i,\Sigma,\texttt{aux},\sigma,\tau}$, where $\texttt{aux} = \perp$ and $\Sigma = (\texttt{arg1}, \texttt{arg2}, \texttt{arg3})$ instantiated as follows: $\texttt{arg1} = (0, \infty)$; $\texttt{arg2} = (\mathsf{sid}, i, k, r)$; $\texttt{arg3} = \mathcal{R}_\emptyset$. The above transaction is posted as long as it is not claimed already, i.e., $(\mathsf{sid}, i, k, r) \in \text{CLAIM}_i$ in a previous step.

This completes the description of the protocol. The protocol terminates in $O(\rho_c)$ ledger rounds. A depiction of the transactions that are associated with a protocol round is given in Figure 4



r-th protocol round
m1
m2
corr
m1
m2
m3
m3
1
2
3
ledger
[X →1, (sid,1,2,r), ∞]
[X →1, (sid,1,3,r), ∞]
[Y →2, (sid,2,1,r), ∞]
[Y →2, (sid,2,3,r), ∞]
[Z →3, (sid,3,1,r), ∞]
[Z →3, (sid,3,2,r), ∞]
time = -12
ledger
[1→2, (sid,1,2,r), R2(), (10r,10r+9)]
[1→3, (sid,1,3,r), R3(), (10r,10r+9)]
[2→1, (sid,2,1,r), R1(), (10r,10r+9)]
[2→3, (sid,2,3,r), R3(), (10r,10r+9)]
[3→1, (sid,3,1,r), R1(), (10r,10r+9)]
[3→2, (sid,3,2,r), R2(), (10r,10r+9)]
time = 0
missing transactions
ledger
[2→ Y, (sid,1,2,r), m2, ∞]
[3→ Z, (sid,1,3,r), m3, ∞]
[1→ X, (sid,2,1,r), m1, ∞]
[3→ Z, (sid,2,3,r), m3, ∞]
[1→ X, (sid,3,1,r), m1, ∞]
[2→ Y, (sid,3,2,r), m2, ∞]
time = 10r
Round r: player 3 aborts
players 1,2 collect transactions
ledger
[1→X, (sid,1,3,r), ∞]
[2→Y, (sid,2,3,r), ∞]
time = 11r

Fig. 4. The transactions associated with the first round r of our protocol compiler. $R_i(\cdot)$ is a relation which is true given the r-th round message of P_i (for the given correlated randomness and previous messages); m_i is the message of player P_i for round r. Player 3 aborts in the r-th round of the protocol and players 1,2 collect their reward.

Observe that by using a constant-round protocol $\pi_{\texttt{Mal}}$ [23], we obtain a protocol with constantly many ledger rounds. Furthermore, as soon as an honest party posts a protocol-related transaction, he is guaranteed to either receive his output or have a positive balance (of at least c coins) after $O(\rho_c)$ ledger rounds. The following theorem states the achieved security. We assume the protocol is executed in the synchronous model of Section 3.1.

Theorem 3. *Let $\bar{\mathcal{G}} = (\bar{\mathcal{G}}_{\text{LEDGER}}, \bar{\mathcal{G}}_{\text{CLOCK}})$, The above protocol in the $(\bar{\mathcal{G}}, \mathcal{F}_{\text{CORR}}^{\mathcal{D}_{\texttt{Mal}}})$-hybrid world realizes $\widetilde{\mathcal{W}}(\mathcal{F})$ with robust compensation.*

Proof (sketch). We first prove that the above protocol is simulatable, by sketching the corresponding simulator $\mathcal{S}$. If the protocol aborts already before the parties make their transactions, then the simulator can trivially simulate such an abort, as he needs to just receive the state of the ledger and see if all wallets corresponding to honest parties have sufficient funds to play the protocol. In the following we show that the rest of the protocol (including the ledger's contents) can be simulated so that if there is an abort, honest parties' wallets have a positive balance as required by Q fairness. First we observe that the simulator $\mathcal{S}$ can easily decide the islands in which the parties are split, as he internally simulates the sampling functionality. Any island other than the one of honest parties (all honest parties will be in the same island because they will post transactions including the same public setup-component) is trivially simulatable as it only consists of adversarial parties and no guarantee is given about their wallets by Q-fairness. Therefore, it suffices to provide a simulator for the honest parties' island. To this direction, the simulator uses the simulator $\mathcal{S}_{\pi^{+1}_{\mathtt{Mal}}}$ which is guaranteed to exist from the security of $\pi^{+1}_{\mathtt{Mal}}$ to decide which messages to embed in the transactions of honest parties (the messages corresponding to corrupted parties are provided by the adversary). If $\mathcal{S}_{\pi^{+1}_{\mathtt{Mal}}}$ would abort, then $\mathcal{S}$ interacts the ideal functionality to abort and continues by claiming back all the committed transactions to the honest parties' wallets, as the protocol would. The soundness of the simulation of $\mathcal{S}_{\pi^{+1}_{\mathtt{Mal}}}$ ensure that the output of the parties and the contents of the ledger in the real and the ideal world are indistinguishable.

The fact that the protocol will eventual terminate given sufficient rounds of activating every party (i.e., in the terminology of Definition 2, given a sufficiently high threshold T) follows by inspection of the protocol: in each round every party needs at most a (fixed) polynomial number of activations to post the transactions corresponding to his current-round message-vector. (In fact, the polynomial is only needed in the initial committing-transactions round and from that point on it is linear). To complete the proof, we argue that **(1)** when the protocol does not abort, every honest party has a non-negative balance, and **(2)** when the protocol aborts, then honest parties have a positive balance of at least c coins as required by predicate Q for the simulator to be able to complete its simulation and deliver the (possibly aborting) outputs. These properties are argued as follows:

Property **(1)**: The parties that are not in the honest parties' islands cannot claim any transaction that honest parties make towards them as the ledger will see they as not in the island and reject them. Thus by the last round every honest party will have re-claimed all transactions towards parties not in his island. As far as parties in the honest island are concerned, if no abort occurs then every party will claim all the transactions from parties in his island, and therefore his balance will be 0.

Property **(2)**: Assume that the protocol aborts because some (corrupted) P_i broadcasts an inconsistent message in some round ρ. By inspection of the protocol one can verify that honest parties will be able to claim all transaction-commitments done to them up to round ρ (as they honestly execute their protocol) plus all committed transactions that they made for rounds $\rho + 1 \ldots, \rho_{\mathrm{c}}$. Additionally, because P_i broadcasts an inconsistent message in round ρ, he will be unable to claim transactions of honest parties done from round ρ and on; these bitcoins will be reclaimed by the honest parties, thus giving their wallets a positive balance of at least c coins.

Acknowledgements. The first author was supported by ERC project CODAMODA # 259152, and the third author was supported partly by the Swiss National Science Foundation (SNF) Ambizione grant PZ00P2-142549. This work was done (in part) while the authors were visiting the Simons Institute for the Theory of Computing, supported by the Simons Foundation and by the DIMACS/Simons Collaboration in Cryptography through NSF grant #CNS-1523467 and (in part) when visiting the National Kapodistrian University of Athens. The authors thank Andrew Miller for helpful discussions.

References

1. M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Fair two-party computations via the bitcoin deposits. In *1st Workshop on Bitcoin Research 2014 (in Assocation with Financial Crypto)*, 2014. http://eprint.iacr.org/2013/837.
2. M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Secure multiparty computations on bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 443–458. IEEE Computer Society Press, May 2014.
3. G. Asharov, Y. Lindell, and H. Zarosim. Fair and efficient secure multiparty computation with reputation systems. In K. Sako and P. Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 201–220. Springer, Heidelberg, Dec. 2013.
4. N. Asokan, M. Schunter, and M. Waidner. Optimistic protocols for fair exchange. In *ACM CCS 97*, pages 7–17. ACM Press, Apr. 1997.
5. N. Asokan, V. Shoup, and M. Waidner. Optimistic fair exchange of digital signatures (extended abstract). In K. Nyberg, editor, *EUROCRYPT'98*, volume 1403 of *LNCS*, pages 591–606. Springer, Heidelberg, May / June 1998.
6. B. Barak, R. Canetti, Y. Lindell, R. Pass, and T. Rabin. Secure computation without authentication. In V. Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 361–377. Springer, Heidelberg, Aug. 2005.
7. I. Bentov and R. Kumaresan. How to use bitcoin to design fair protocols. In J. A. Garay and R. Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 421–439. Springer, Heidelberg, Aug. 2014.
8. D. Boneh and M. Naor. Timed commitments. In M. Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 236–254. Springer, Heidelberg, Aug. 2000.
9. C. Cachin and J. Camenisch. Optimistic fair secure computation. In M. Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 93–111. Springer, Heidelberg, Aug. 2000.
10. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, Oct. 2001.
11. R. Canetti, Y. Dodis, R. Pass, and S. Walfish. Universally composable security with global setup. In S. P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 61–85. Springer, Heidelberg, Feb. 2007.
12. R. Canetti and M. Fischlin. Universally composable commitments. In J. Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 19–40. Springer, Heidelberg, Aug. 2001.
13. R. Canetti, A. Jain, and A. Scafuro. Practical UC security with a global random oracle. In G.-J. Ahn, M. Yung, and N. Li, editors, *ACM CCS 14*, pages 597–608. ACM Press, Nov. 2014.

14. R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. Universally composable two-party and multi-party secure computation. In *34th ACM STOC*, pages 494–503. ACM Press, May 2002.
15. R. Canetti and T. Rabin. Universal composition with joint state. In D. Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 265–281. Springer, Heidelberg, Aug. 2003.
16. R. Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In J. Hartmanis, editor, *STOC*, pages 364–369. ACM, 1986.
17. J. A. Garay, R. Gelles, D. S. Johnson, A. Kiayias, and M. Yung. A little honesty goes a long way - the two-tier model for secure multiparty computation. In Y. Dodis and J. B. Nielsen, editors, *TCC 2015, Part I*, volume 9014 of *LNCS*, pages 134–158. Springer, Heidelberg, Mar. 2015.
18. J. A. Garay, J. Katz, U. Maurer, B. Tackmann, and V. Zikas. Rational protocol design: Cryptography against incentive-driven adversaries. In *54th FOCS*, pages 648–657. IEEE Computer Society Press, Oct. 2013.
19. J. A. Garay, P. D. MacKenzie, M. Prabhakaran, and K. Yang. Resource fairness and composability of cryptographic protocols. In S. Halevi and T. Rabin, editors, *TCC 2006*, volume 3876 of *LNCS*, pages 404–428. Springer, Heidelberg, Mar. 2006.
20. O. Goldreich. *Foundations of Cryptography: Basic Tools*, volume 1. Cambridge University Press, Cambridge, UK, 2001.
21. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In A. Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.
22. S. D. Gordon and J. Katz. Complete fairness in multi-party computation without an honest majority. In O. Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 19–35. Springer, Heidelberg, Mar. 2009.
23. Y. Ishai, R. Ostrovsky, and V. Zikas. Secure multi-party computation with identifiable abort. In J. A. Garay and R. Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 369–386. Springer, Heidelberg, Aug. 2014.
24. J. Katz, U. Maurer, B. Tackmann, and V. Zikas. Universally composable synchronous computation. In A. Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 477–498. Springer, Heidelberg, Mar. 2013.
25. A. Kiayias, H.-S. Zhou, and V. Zikas. Fair and robust multi-party computation using a global transaction ledger. Cryptology ePrint Archive, Report 2015/574, 2015. `http://eprint.iacr.org/2015/574`.
26. A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. Cryptology ePrint Archive, Report 2015/675, 2015. `http://eprint.iacr.org/2015/675`.
27. R. Kumaresan and I. Bentov. How to use bitcoin to incentivize correct computations. In G.-J. Ahn, M. Yung, and N. Li, editors, *ACM CCS 14*, pages 30–41. ACM Press, Nov. 2014.
28. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf, 2008.
29. B. Pinkas. Fair secure two-party computation. In E. Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 87–105. Springer, Heidelberg, May 2003.
30. T. Ruffing, A. Kate, and D. Schröder. Liar, liar, coins on fire!: Penalizing equivocation by loss of bitcoins. In I. Ray, N. Li, and C. Kruegel:, editors, *ACM CCS 15*, pages 219–230. ACM Press, Oct. 2015.
31. A. C.-C. Yao. Protocols for secure computations (extended abstract). In *23rd FOCS*, pages 160–164. IEEE Computer Society Press, Nov. 1982.