TARDIS: Time And Relative Delays In Simulation

Carsten Baum[1⋆], Bernardo David[2⋆⋆], Rafael Dowsley[3⋆⋆⋆], Jesper Buus Nielsen[1], and Sabine Oechsner[1†]

[1] Aarhus University, Denmark
[2] IT University of Copenhagen, Denmark
[3] Bar-Ilan University, Israel

Abstract. This work introduces an extension of the UC framework with an abstract notion of time that allows for modeling relative delays in communication and sequential computation without requiring parties to keep track of a clock. The potential uses of this extension are demonstrated by: (1) formalizing a functionality for (semi-)synchronous secure message transmission; (2) formalizing the notion of time-lock puzzles in the UC setting and showing how to realize it in the restricted programmable and observable global random oracle model; (3) showing that UC time-lock puzzles yield UC-secure fair coin flips; (4) showing that UC-secure two-party computation realizing a new notion of output-independent abort can be obtained leveraging composable time-lock puzzles. Finally, we show that a programmable random oracle is necessary to obtain UC-secure fair coin flip, secure two-party computation with output-independent abort or time-lock puzzles, which yields a new separation between programmable and non-programmable random oracles.

1 Introduction

The Universal Composability (UC) framework [15] is the gold standard for formally analyzing cryptographic protocols as it provides strong security guarantees that allow UC-secure protocols to be arbitrarily composed. This is a very useful property and enables the modular design of cryptographic protocols. However, the original UC framework is inherently asynchronous and does not support the notion of time. Katz et al. [30] introduced a global clock functionality in order to

⋆ This work was funded by the European Research Council (ERC) under the European Unions' Horizon 2020 research and innovation programme under grant agreement No 669255 (MPCPRO).

⋆⋆ This work was supported by a grant from Concordium Foundation and by DFF grant number 9040-00399B (TrA^{2}C).

⋆⋆⋆ Supported by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office.

† Supported by the Danish Independent Research Council under Grant-ID DFF-8021-00366B (BETHE).

define universally composable synchronous computation. Their clock functionality captures the essence of loosely synchronized wall clocks that are available to all parties. This notion is particularly useful in reasoning about synchronous protocols in the UC framework, since the honest parties can use the global clock to achieve synchronization.

However, many cryptographic protocols do not depend on concrete time (or delays) provided by a wall clock, but just on the relative timing that is observable through events, such as the arrival of messages or the completion of some computation. In particular, protocols in a semi-synchronous communication model (*e.g.* [22,5]) rely on the fact that there exists a finite (but unknown) upper bound for the delay in communication channels, not necessarily requiring that events (*e.g.* the arrival of a message) occur at a specific wall clock time (or even within a concrete delay) as long as they occur in a certain order. In this case, using a global clock can make the design and security analysis of such protocols unnecessarily complicated.

Another important challenge lies in modeling sequential computation and computational delays in the UC framework. Since the environment may operate in many parallel sessions and activate parties arbitrarily, it obtains an unfair computation advantage in relation to the parties. For example, even if its computational power is constrained within a session, the environment can use multiple sessions to solve a computational problem assumed to require at least a certain amount of computational steps (and thus time) faster than a regular party. This precludes the UC modeling and construction of primitives based on sequential computation and computational delays, such as time-lock puzzles [38].

1.1 Our Contributions

In this work, we introduce a new abstract notion of time in the UC framework that allows us to reason about communication channels with delays as well as delays induced by sequential computation. We demonstrate the power of our approach by introducing the first definition and construction of composable time-lock puzzles (TLPs) without resorting to clocks, which we use to obtain the first two-party computation protocol with output-independent abort. Finally, we establish that a programmable random oracle is necessary for obtaining UC-secure TLPs. Our contributions are summarized below:

- **Abstract Time in UC:** we put forth a novel abstract notion of time for the UC framework capturing relative event ordering without a clock.
- **First Composable Treatment of Time-Lock Puzzles (TLPs):** we introduce the first composable definition and construction of time-lock puzzles.
- **First Two-Party Computation Protocol with Output Independent Abort:** we use TLPs to construct a UC-secure two-party computation protocol where the adversary cannot see the output before deciding to abort.
- **Impossibility of UC-Secure TLPs without Programmable Random Oracles:** we prove that the use of programmable random oracles are nec-

essary for constructing UC-secure TLPs, yielding a new separation between programmable and non-programmable random oracles.

The advantage of our new abstract notion of time for the UC framework is twofold: 1. it captures delays without explicitly referring to wall clock time and 2. it allows for modeling delays induced by sequential computation. This notion it makes it possible to state protocols and security proofs in terms of the relative delays between events (*e.g.* the arrival of a message or completion of a computation) and the existence of large enough delays that ensure that these events occur in a certain order.

Building on this model, we introduce the first definition and construction of UC-secure time-lock puzzles. Previous works on time-lock puzzles have not considered composability guarantees, meaning that current protocols that use time-lock puzzles as black-box building blocks cannot be formally proven secure. Our construction is based on the classical time-lock assumption of Rivest *et al.* [38], which we capture as a UC setup assumption using our model of abstract time. In order to construct a simulator for our security proofs, we further require a restricted programmable and observable global random oracle, which we prove to be *necessary*, providing a new separation between non-programmable and programmable random oracles.

As an application of our abstract time model and composable TLPs, we introduce the notion of two-party computation (2PC) with output independent abort (OIA) along with the first OIA-2PC protocol. This new security notion for secure computation guarantees that an adversary who aborts the execution cannot learn any information about the output *before* deciding to abort, only obtaining the output after this decision is made. Our new definition improves on the standard security notion with abort (realized by all known 2PC protocols), which allows for the adversary to decide whether to force the honest parties to abort without obtaining the output *after* learning the output itself. We argue that this new security notion is optimal, since fairness (*i.e.* ensuring all parties obtain the output if the adversary does so) for 2PC protocols is impossible [20].

Our results are further explored in Section 1.3.

1.2 Related Work.

Composition frameworks with time and fairness. Composition frameworks for cryptographic protocols (e.g. UC [15], constructive cryptography [34], the reactive simulatability (RSIM) framework [36]) provide strong security guarantees for protocols under concurrent composition. In all mentioned frameworks, communication is though inherently *asynchronous* channels. Several works have therefore studied general composition guarantees with *synchronous* communication by introducing a shared source of time or restricting adversarial scheduling. Modeling network timing assumptions such as bounded message delay and clock drift and the resulting concurrent composition guarantees for specific tasks was studied for zero-knowledge [23], [25] and MPC [29]. In the context of composition frameworks, Backes et al. [4] model traffic-related timing attacks in GNUC [27] by

allowing the adversary to measure the local time at which a message arrives. In this setting, each party has a local execution time, and the EXEC function of GNUC maps the local times into a global time. Backes et al. [3] studied fairness in the RSIM framework and achieve composable notion of fairness by restricting the adversary model to fair schedulers who deliver any message after at most a polynomial number of steps.

The work that is most closely related to ours is the model by Katz *et al.* [30] that extends the Global UC (GUC) framework [16] to provide all parties with access to a global clock functionality for the purpose of synchronization. This model requires all parties executing a (semi-)synchronous protocol to keep track of current global clock time and to actively query the global clock functionality in order to advance of time. In particular, even if the model of Katz *et al.* is used to define semi-synchronous communication, it implies that all parties are kept synchronized and may learn how much time has elapsed since it their last activation (*i.e.* by obtaining the current time from the global clock), which is a rather strong synchrony assumption. However, many protocols cast in this model do not crucially rely on obtaining concrete time stamps or determining concrete delays between party activations, as long as messages are guaranteed to be delivered within certain delays and in a certain order (*e.g.* as in [5]). This is exactly the kind of guarantees that our model captures without explicitly exposing time keeping to parties or requiring them to keep track of concrete time sources. By doing that, our model allows us to analyse many protocols cast in the model of Katz *et al.* while significantly relaxing synchrony assumptions. Moreover, our model can be used to capture delays induced by sequential computation, which is not captured by the global clock model of Katz *et al.*.

Time-Lock Puzzles and Computational Delay The original construction of time-lock puzzles was proposed by Rivest, Shamir and Wagner [38]. Boneh and Naor [12] introduced the notion of timed commitments. An alternative construction of time-lock puzzles was presented by Bitansky et al. [10]. Recently, the related notion of verifiable delay functions has been investigated [11,37,42]. These constructions are closely related in that they rely on sequential computational tasks that force parties to spend a certain amount of time before they are able to obtain an output. However, none of these works have considered composability issues for such time-based primitives. In particular, the issues of malleability for these time-based primitives and the relationship between computational and communication delay are notably ignored in previous works. The lack of composabillity guarantees for time-lock puzzles is a significant shortcoming, since these primitives are mostly used as building blocks for more complex protocols and current constructions do not ensure that their security guarantees are retained when composed with other primitives to obtain such protocols. Our composable treatment of time-lock puzzles addresses theses issues by introducing constructions that can be arbitrarily composed along with a framework for analysing complex protocols whose security relies on the relative delays in computation and communication.

Aborts and Fairness in Secure Computation An MPC protocol is said to be fair if a party can obtain the output if and only if all other parties also obtain the output. It is a well-known fact that fair MPC in the standard communication model is impossible with a dishonest majority [20]. Given the impossibility to achieve fairness, techniques for identifying misbehaving parties responsible for causing an abort have been investigated [28,7]. In the last few years a line of work developed which imposes financial penalties on parties who are identified as misbehaving by using cryptocurrencies and smart contracts, thus giving financial incentives for rational parties to behave in a fair way. Protocols have been designed to punish misbehavior at any point of the protocol execution (Fair Computation with Penalties) [2,33,31] or to only punish participants that learn the output but prevent others from doing the same (Fair Output Delivery with Penalties) [1,9,32,6]. However, these protocols allow the adversary to make a decision on whether to abort or not *after* seeing the output that will be obtained by the honest parties in case the execution proceeds.

The recent work of Couteau et al. [21] studies the problem of obtaining partially-fair exchange from time-lock puzzles, but in much weaker security and adversarial models. In particular, their work does not consider composability issues and is limited to the specific problem of fair exchange rather than the general problem of secure computation considered in our results.

Random oracle separation results. Our impossibility result provides yet another separation between the programmable and non-programmable random oracle models, complementing the few previously known separations [35,41,24,8].

1.3 Our Techniques

In the remainder of this section, we briefly outline the new techniques behind our results and their implications.

Abstract Time: Our goal is to express different timing assumptions (possibly related) within the GUC framework in such a way that protocols are oblivious to them. We do so by providing the adversary with a way of advancing time in the form of *ticks*. A tick represents a discrete unit of time. Time can only be advanced, and moreover only one unit at a time. In contrast to Katz et al. [30], however, these ticks and thus the passing of time are not supposed to be directly visible to the protocol. Thus instead of a global clock that parties can ask for the current time, we add a ticking interface to ideal functionalities. This way, timing-related observable behavior becomes an assumption of the underlying functionalities, e.g. of a computational problem or a channel. Parties may now observe events triggered by elapsed time, but not the time itself. Ticked functionality are free to interpret ticks in any way they like; this way we can synchronize and relate ticks representing elapsed time in different "units" like passed time or computation steps. The technical challenge is to ensure in a composable way that all honest parties have a chance at observing all relevant timing-related events. Katz et al. solved this issue inside the global clock by keeping track of

which parties have been activated in the current time period (and thus asked for the time) and refusing to advance time if necessary. Since our modeling does not have a single entity that is ticked, we take a different approach: We restrict the class of environments to those that activate all honest parties (in arbitrary order) between two ticks. To further control the observable side effects of ticks, we restrict protocols and ideal functionalities to interact in the "pull model" known from Constructive Cryptography, precluding functionalities from implicitly providing communication channels between parties and instead requiring parties to actively query functionalities in order to obtain new messages. Apart from presenting a clear abstraction of time, this notion explicitly exposes issues that must be taken in consideration when implementing protocols that realize our functionalities, *i.e.* the concrete delays in real world communication channels and computation. In particular, while the theoretical protocol description and security analysis can be carried in terms of such abstract delays, our techniques clarify the relationship between concrete time-based parameters (*e.g.* timeouts vs. network delays) that must be respected in protocol implementations. We will go into this in more detail in Section 2.

Composable Treatment of Time-Lock Puzzles: To illustrate the potential uses of our framework, we present the first definition and construction of UC-secure Time-Lock Puzzles (TLP). We depart from the classical construction by Rivest et al. [38] and provide the first UC abstraction behind the Time-Lock Assumption, which is modeled in a "generic group model" style, hiding the group description from the environment and limiting its access to group operations. A party acting as the "owner" of an instance of the TLP functionality can generate a puzzle containing a certain message that should be revealed after a certain number of computational steps. The functionality allows the parties to make progress on the solution of the puzzle every time that it is ticked. Once a party solves a puzzle, it can check that a certain message was contained in that puzzle. The ticks given to this functionality come externally from the adversary and we require in the framework that the parties get activated often enough. We show that our UC abstraction of the Time-Lock Assumption allows us to implement UC-secure TLPs in the restricted programmable and observable global random oracle model of Camenisch et al. [13] (which turns out to be necessary for UC-realizing TLPs).

Two-Party Computation with Output Independent Abort: To further showcase our framework we construct the first protocol for secure two-party computation (2PC) with output-independent abort, i.e., the adversary must decide whether to abort or not before seeing the output. In order to do so, we build on techniques from [6]: there, the authors combine an MPC protocol with linearly secret-shared outputs and an additively homomorphic commitment by having each party commit to its share of the output and then reconstruct the output inside the commitments. In [6], the output of the secure computation is obtained by opening the final commitments resulting from the reconstruction procedure, which allows the adversary to learn the output before the honest parties do and refuse to open

its commitment, causing the protocol to abort. Similarly to [6], we combine a 2PC protocol with secret-shared outputs and an additively homomorphic commitment but we define and construct commitments with a new delayed opening interface. When a delayed opening happens, the receiver is notified after a communication delay but only receives the revealed message after an *opening delay*. Hence, we can obtain output independent abort by delayed opening the final commitments obtained after reconstructing the output and considering that a party aborts if it does not execute a delayed opening of their commitments before the other parties delayed openings reveal their messages. Finally, we show how to obtain UC-secure additively homomorphic commitments with delayed opening by modifying the scheme of Cascudo et al. [18] with the help of the delayed secure message transmission and TLP functionalities.

Impossibility Result. Finally, we prove that a non-programmable random oracle is not sufficient for obtaining UC-secure fair-coin flip, secure 2PC with output-independent abort or TLP. Therefore a programmable random oracle is necessary to implement these primitives, yielding a separation between the programmable and non-programmable random oracle models. This also shows that our TLP construction which requires this strong assumption is in that sense "optimal".

1.4 Paper Outline

Section 2 introduces our model of abstract time. In Sections 3 and 4, we present two possible interpretations of abstract time for communication delay and for computational delay, respectively. In Section 5, we show how to construct UC-secure time-lock puzzles. In Section 6, we show how composable time-lock puzles can be used obtain 2PC with output-independent abort. Finally, in Section 7, we prove it is impossible to obtain composable time-lock puzzles without programmabe random oracles.

2 UC with Relative Time

This section introduces our UC framework with relative time (RUC) and discusses its interpretation using the examples of communication delays and non-parallelizable computation.

2.1 Modeling Elapsing Time

Based on the observation that many cryptographic protocols do not rely on exact timing information through clocks etc., but only on relative timing, we propose to model elapsing time by introducing adversarially controlled time "ticks" as basic discrete units of passing time. At the core, ticks are a tool for modeling physical timing assumptions like a channel introducing a certain delay to messages or a problem taking a certain time to solve. The ticks are importantly only visible to ideal functionalities, while protocols can only depend on the relative

timing that is observable through events, e.g., messages arriving or computations completing. This allows the notion of abstract time to be instantiated later as protocol implementers wish, as long as the relative timing between events is preserved.

Our model is a modification of the GUC framework [16] to incorporate relative timings. Our changes affect the activation patterns of entities as well as their interface. Assume that $\mathcal{T}$ is the set of names of all possible timing assumptions in the protocol to be executed. This set remains the same for a particular protocol, no matter in which hybrid model it is studied. We demand that:

Ticked functionalities can be activated with (tick, t) messages by the adversary for $t \in T \subseteq \mathcal{T}$. Upon this activation, the functionality can possibly perform local computations and must then return to the adversary. Upon any other activation by a party $\mathcal{P}$, the functionality must either return to the caller or activate the adversary. Moreover, we specify some default behavior that is omitted in descriptions: The functionality can be activated by party $\mathcal{P}$ with message $(\mathsf{NoQuery}, \mathcal{P})$ upon which it performs no actions and returns to the caller. When a (tick, t) message does not trigger any action within the functionality, it can be omitted to simplify the description of functionalities that do not rely on timing assumptions. If t is clear from the context, we will omit it.

Tick-forwarding adversaries must react to (tick, t) messages from the environment by forwarding them to the corresponding ideal functionality.

Ticking environments are parameterized by a set $\mathcal{T}$ of all timing assumptions in the protocol they are currently executing. An environment can instruct an adversary to trigger a tick for assumption $t \in \mathcal{T}$ by sending (tick, t). Whenever the environment sends one such message, it must send (tick, t') for all other $t' \in \mathcal{T}$ to the adversary before performing any other activations. Between two consecutive ticks to all assumptions in this way, the environment must activate all parties at least once in arbitrary order.

Activation-forwarding parties and protocols As is standard in any protocol that models communication through physical assumptions like a channel, parties can only activate functionalities or return to the caller when activated. Before returning to the caller, a party needs to activate each functionality at least once, possibly with the default message $(\mathsf{NoQuery})$.

The execution model remains otherwise the same. These restrictions ensure in a composable way that (1) time is advanced for all functionalities in synchrony and (2) between any two ticks, each party has a chance to observe the passing of time through changes in behavior of functionalities. The ticked functionality itself is free to interpret the ticks in any way it likes. Typically, the tick will trigger a state change, e.g. new information becomes available to parties.

Definition 1 (RUC-emulation). *Let π and ϕ be PPT protocols where π is $\bar{\mathcal{G}}$-subroutine respecting. We say that π RUC-emulates ϕ with respect to shared functionality $\bar{\mathcal{G}}$ if for any PPT adversary $\mathcal{A}$ there exists a PPT adversary $\mathcal{S}$ such*

that for any $\bar{\mathcal{G}}$-externally constrained environment $\mathcal{Z}$, we have that

$$EXEC^{\bar{\mathcal{G}}}_{\phi,\mathcal{S},\mathcal{Z}} \approx EXEC^{\bar{\mathcal{G}}}_{\pi,\mathcal{A},\mathcal{Z}}.$$

Denote by $\rho^{\phi\to\pi}$ the protocol obtained by replacing π in ρ by ϕ.

Theorem 1 (Universal composition for RUC). *Let ρ, π, ϕ be PPT multiparty protocols where both ϕ and π are $\bar{\mathcal{G}}$-subroutine respecting, and where π RUC-emulates ϕ. Then $\rho^{\phi\to\pi}$ RUC-emulates ρ.*

Proof. Since the execution model of RUC and hence also this proof is very similar to the universal composition theorem in GUC, we focus here on the differences, namely the restrictions in activation patterns. The proof is a reduction of $\rho^{\phi\to\pi}$ RUC-emulating ρ to π RUC-emulating ϕ by showing that any attack on $\rho^{\phi\to\pi}$ can be translated into an attack on ϕ. Assume for the sake of contradiction that $\rho^{\phi\to\pi}$ does not RUC-emulate ρ. In this case, there exists a tick-forwarding adversary $\mathcal{D}_\rho$ such that the ticking environment $\mathcal{Z}$ can distinguish the two executions. This tick-forwarding adversary $\mathcal{D}_\rho$ can be split into a tick-forwarding adversary $\mathcal{D}_\pi$ interacting with π and the remaining tick-forwarding adversary $\mathcal{D}$. We will now argue that $\mathcal{Z}$ executing $\mathcal{D}$ and ρ is again a ticking environment in our model, which concludes the proof. Since $\mathcal{D}$ is tick-forwarding, $\mathcal{Z}$ executing $\mathcal{D}$ is still ticking all ticked functionalities as specified. To reason about $\mathcal{Z}$ executing ρ, observe that any possible message that a ticked functionality can receive from an honest party can be attributed to a specific party $\mathcal{P}$. For any party in protocol ρ, we moreover know that it is activation-forwarding, meaning that any activation of a party $\mathcal{P}$ in $\rho^{\phi\to\pi}$ will result in an activation of the corresponding dummy party in ρ. Therefore, if $\mathcal{Z}$ actives all parties of $\rho^{\phi\to\pi}$ between ticks, then $\mathcal{Z}$ executing ρ does so too. □

2.2 Interpreting ticks

For now, the ticks do not have any meaning attached to them. The interpretation, e.g. how many ticks it takes to complete which task, will depend on the concrete functionality. Note that a system will contain different ticked functionalities, and that the system description will typically omit concrete instantiations of the ticks. Instead, only relations between the ticks of different functionalities are prescribed. Examples could be that a message is transported "fast enough" (message delay is less than the time it takes to perform some computation) or that a computation is performed "slow enough" (computation time is greater than message delay). Concrete delays etc. will then only be fixed once the system is instantiated concretely.

This work focuses on two notions, bounded message delays and computational hardness, that we will describe in the following sections.

3 Communication Delay

In the context of communication, we interpret abstract time ticks in order to model message transmission delays. That is, we model the fact that message

transmission is never instantaneous and thus takes time. Moreover, we model the different synchrony assumptions for communication channels in current literature. As a concrete example, we will study the secure message transmission functionality $\mathcal{F}_{\mathsf{smt}}^{\ell}$. Any implementation of an interactive functionality must strictly speaking be in a $\mathcal{F}_{\mathsf{smt}}^{\ell}$(or similar) hybrid model and hence our modeling can be adapted to any interactive functionality. Notice that by interactive functionalities we mean any functionality that transmits information between parties, a task that is often done implicitly by UC ideal functionalities such as those for secure computation.

Functionality $\mathcal{F}_{\mathsf{smt,delay}}^{\Delta}$

$\mathcal{F}_{\mathsf{smt,delay}}^{\Delta}$ proceeds as follows, when parameterized by maximal delay $\Delta > 0$, sender $\mathcal{P}_S$, receiver $\mathcal{P}_R$ and adversary $\mathcal{S}$. Internal variable t is initally set to 0, and flags msg, $\mathsf{released}$, done to $\bot$.

Send: Upon receiving an input $(\mathsf{Send}, \mathsf{sid}, \mathcal{P}_R, m)$ from party $\mathcal{P}_S$, do:
- If $\mathsf{msg} = \bot$, record m, set $\mathsf{msg} = \top$, and send $(\mathsf{Sent}, \mathsf{sid}, \mathcal{P}_R, \ell(m))$ to $\mathcal{S}$.
- If $\mathsf{msg} = \top$, send $(\mathsf{None}, \mathsf{sid})$ to $\mathcal{P}_S$.

Receive: Upon receiving an input $(\mathsf{Rec}, \mathsf{sid}, R)$ from $\mathcal{P}_R$, do:
- If $\mathsf{released} = \bot$ and $\mathsf{done} = \bot$, then send $(\mathsf{None}, \mathsf{sid})$ to $\mathcal{P}_R$.
- If $\mathsf{released} = \top$ and $\mathsf{done} = \bot$, then $\mathsf{msg} = \top$ as well and there exists a recorded message m. Set $\mathsf{done} = \top$ and send $(\mathsf{Sent}, \mathsf{sid}, \mathcal{P}_S, \mathcal{P}_R, m)$ to $\mathcal{P}_R$.
- If $\mathsf{done} = \top$, then send $(\mathsf{done}, \mathsf{sid})$ to $\mathcal{P}_R$.

Release message: Upon receiving an input $(\mathsf{ok}, \mathsf{sid}, \mathcal{P}_S, \mathcal{P}_R)$ from $\mathcal{S}$, do:
- If $\mathsf{msg} = \bot$, then send $(\mathsf{None}, \mathsf{sid}, \mathcal{P}_S, \mathcal{P}_R)$ to $\mathcal{S}$.
- If $\mathsf{msg} = \top$ and $\mathsf{released} = \bot$, then set $\mathsf{released} = \top$.
- If $\mathsf{released} = \top$, then send $(\mathsf{None}, \mathsf{sid}, \mathcal{P}_S, \mathcal{P}_R)$ to $\mathcal{S}$.

Tick: Upon receiving an input $(\mathsf{Tick}, \mathsf{sid})$ from $\mathcal{S}$, do:
- If $\mathsf{msg} = \bot$, then send $(\mathsf{None}, \mathsf{sid}, \mathcal{P}_S, \mathcal{P}_R)$ to $\mathcal{S}$.
- If $\mathsf{msg} = \top$ and $\mathsf{released} = \bot$, then set $t = t + 1$. If now $t = \Delta$, set $\mathsf{released} = \top$. Then send $(\mathsf{Ticked}, \mathsf{sid})$ to $\mathcal{S}$.
- If $\mathsf{released} = \top$, then send $(\mathsf{None}, \mathsf{sid}, \mathcal{P}_S, \mathcal{P}_R)$ to $\mathcal{S}$.

Corrupt: Upon receiving an input $(\mathsf{Corrupt}, \mathsf{sid}, \mathcal{P})$ from $\mathcal{S}$ where $\mathcal{P} \in \{\mathcal{P}_S, \mathcal{P}_R\}$, do:
- If $\mathcal{P} = \mathcal{P}_S$ and $\mathsf{msg} = \bot$, send $(\mathsf{None}, \mathsf{sid}, \mathcal{P}_S, \mathcal{P}_R)$ to $\mathcal{S}$.
- If $\mathcal{P} = \mathcal{P}_S$ and $\mathsf{msg} = \top$, then there exists a recorded message m. Send $(\mathsf{Sent}, \mathsf{sid}, m, \mathcal{P}_S, \mathcal{P}_R)$ to $\mathcal{S}$.
- If $\mathcal{P} = \mathcal{P}_R$ and $\mathsf{done} = \bot$, send $(\mathsf{None}, \mathsf{sid}, \mathcal{P}_S, \mathcal{P}_R)$ to $\mathcal{S}$.
- If $\mathcal{P} = \mathcal{P}_R$ and $\mathsf{done} = \top$, then there exists a recorded message m. Send $(\mathsf{Sent}, \mathsf{sid}, m, \mathcal{P}_S, \mathcal{P}_R)$ to $\mathcal{S}$.

Fig. 1. Ticked ideal functionality $\mathcal{F}_{\mathsf{smt,delay}}^{\Delta}$ for secure message transmission with maximal message delay Δ.

3.1 Secure Message Transmission with Delays

Secure message transmission (SMT) is the problem of securely sending a single message m from a sender $\mathcal{P}_S$ to a receiver $\mathcal{P}_R$. Secure means that the power of an eavesdropper intercepting the channel is restricted to learning some leakage $\ell(m)$ on the message and delaying the message delivery. The standard formulation of $\mathcal{F}_{\mathsf{smt}}^{\ell}$ [14, 2019 version] assumes that message delivery can be delayed infinitely by an adversary. Here, we want to add an upper bound on the message delay. The exact constraints on this upper bound will determine whether a protocol operates over synchronous, semi-synchronous or asynchronous channels, as discussed further in Section 3.2

In order to capture elapsed time according to our model, we add a ticks interface to obtain a ticked ideal functionality. The functionality is parameterized by a maximal delay $\Delta > 0$. Requiring $\Delta > 0$ models the fact that communication always takes time. After a message is input to the functionality by the sender, each tick will increase a counter. The message is released to the receiver no later after at most Δ ticks are counted or whenever the ideal adversary instructs the functionality to release it[4]. However, a tick cannot directly trigger the activation of parties other than the adversary. Otherwise, we would be exposing the elapsed time towards the partiesand implicitly synchronizing them. As a consequence, the functionality cannot send the message to the receiver as in [14]. We solve this issue by requiring the receiver to actively query the functionality for newly released messages. Finally, the adversary can adaptively request to corrupt a party $\mathcal{P} \in \{\mathcal{P}_S, \mathcal{P}_R\}$, in which case they will learn the message if the corresponding party knows it already. Note that this corruption behavior differs crucially from Canetti's formulation: Since message transmission is explicitly taking time, adaptive corruptions at runtime are actually meaningful now. In particular, it is no longer possible to first observe leakage on a sent message to then corrupt the sender and change the message that was sent. The resulting ideal functionality $\mathcal{F}_{\mathsf{smt,delay}}^{\Delta}$ is shown in Fig. 1.

In principle, one can transform a UC-functionality also by adding a wrapper that buffers messages and handles ticks. Due to the differences in handling adaptive corruption, we chose a standalone solution for this concrete example.

3.2 Modeling (Semi)-Synchronous Channels

Besides establishing that all messages must be delivered with a maximal delay Δ, our formulation of $\mathcal{F}_{\mathsf{smt,delay}}^{\Delta}$ does not specify if it operates as a synchronous, semi-synchronous or asynchronous channel. This modeling choice is made so that this single formulation can capture all of these assumptions on communication synchrony by imposing constraints of the maximal delay Δ. We obtain a channel satisfying each communication synchrony assumption by constraining Δ as follows:

[4] The delay model could generalized even further by introducing two delay parameters Δ_{min} and Δ_{max} to model that communication *must* take time. In that case, messages are only forwarded after Δ_{min} ticks were received.

– **Synchronous Channel, finite and publicly known Δ:** a synchronous channel is modeled by setting a finite $\Delta > 0$ and allowing all parties to learn Δ, which makes it possible for parties to determine whether a given message was sent or not (since a message must be delivered within the known delay Δ).
– **Semi-Synchronous Channel, finite but unknown Δ:** a semi-synchronous channel is modeled by setting a finite $\Delta > 0$ that is only known to the adversary, which ensures parties that all messages will be eventually delivered but does not allow them to explicitly distinguish a delayed message from a dropped message (since they do not know the maximal delay Δ after which messages are guaranteed to be delivered).
– **Asynchronous Channel, infinite Δ:** an asynchronous channel is modeled by setting $\Delta = \infty$, which allows the adversary to never release messages sent through $\mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$ (*i.e.* essentially dropping these messages).

In the synchronous and asynchronous cases, the constraints on Δ simply model the usual notions of synchronous and asynchronous channels. In the semi-synchronous case, the constraints limit the way a protocol can use Δ, since no information about it is given to honest parties, precluding them from setting other parameters of the protocol relatively to a previously known Δ. We remark that Δ can potentially be chosen by the adversary itself or preset before execution starts, as long as the right constraints for the communication synchrony assumption considered in the proof are obeyed (*i.e.* in the synchronous case the adversarially chosen Δ must be made public to the honest parties and in the semi-synchronous case Δ is not revealed to the honest parties). Notice that the exact value of Δ does not affect the behavior of honest parties in our model because the honest parties cannot perceive the advance of abstract time (*i.e.* the honest parties cannot tell when a tick happened).

4 Computational Delay

We will now introduce a concept for modeling sequential computation inside the UC framework that does not suffer from degradation through composition or adversarially chosen activation of parties. As an example, we will realize the notion of a "time-lock puzzle" [38] in a composable fashion.

4.1 Modeling Time-Lock Puzzles

In a time-lock puzzle (TLP), the owner generates a computational puzzle that outputs a message to the receiver when solved. The main property of the construction is that none of the solvers can obtain the message from the puzzle substantially faster than any other solvers, thus introducing problems that cannot be parallelized.

To the best of our knowledge, this has not been formalized in the UC framework before and there are multiple pitfalls that one has to avoid when formalizing

TLPs. First, UC allows the environment to activate parties at its will throughout the session and it might be that an honest party does not even get activated before the puzzle was solved by the adversary. Even worse, such a modeling might permit that the environment can solve the puzzle in another session, so even by enforcing regular activation inside a session (as in the previous section) or equal computational powers between the iTM modeling the parties as well as the adversary one cannot achieve the aforementioned notion.

Ticked ideal functionalities help us to overcome both issues, and the resulting ticked time lock puzzle ideal functionality $\mathcal{F}_{\mathsf{tlp}}$ is shown in Fig. 2. It can easily be seen that the functionality fulfills our definitions as outlined before. First, any new instance of a puzzle can be tied to a specific party, namely the owner $\mathcal{P}_o$, who can initialize the puzzle by providing a number of computation steps Γ and a message m. The functionality outputs a puzzle $\mathtt{puz} = (\mathtt{st}_0, \Gamma, \mathtt{tag})$ consisting of an initial state $\mathtt{st}_0$, the number of steps Γ needed for reaching a final state and tag $\mathtt{tag}$ used to encode the message. After every tick, each party can use a puzzle state $\mathtt{st}_i$ to call the **Solve** interface, which will append the next state $\mathtt{st}_{i+1}$ to a list of messages delivered to the party after the next tick. By buffering messages containing the next states, we essentially limit all parties' (and the environment's and adversary's) ability to attempt performing more than one solving step per puzzle. Notice that any party who tries to call solve more than once per tick for a puzzle would have to guess the next state $\mathtt{st}_{i+1}$ in order to perform the second call, which can only be done with negligible probability. Once the final state $\mathtt{st}_\Gamma$ is reached, parties can call the **Get Message** interface in order to retrieve the message associated with the puzzle by presenting the puzzle $\mathtt{puz}$ and the final state $\mathtt{st}_\Gamma$ obtained through successive calls to **Solve**. Finally, the environment can at any point send a comp-tick command which will allow each party to "tick" again and obtain a new value, which may get it closer to the solution of the puzzle.

Observe that this model does neither restrict the actual computational power of the environment nor any other iTM. The environment can activate any party arbitrarily often, as long as the honest parties also occasionally can have the ability to access the restricted resource. Care must also taken to allow limited ideal adversarial control over the functionality's answers to queries to **Solve** containing undefined states and queries to **Get Message** containing undefined $(\mathtt{puz}, \mathtt{st})$ tuples. While the adversary is allowed to provide an arbitrary sequence of states $\mathtt{st}_0, \ldots, \mathtt{st}_\Gamma$ and an arbitrary tag $\mathtt{tag}$, the functionality enforces the fact that, once defined, the same sequence of steps will be deterministically obtained by all honest parties invoking **Solve**. However, queries to $\mathcal{F}_{\mathsf{tlp}}$ involving undefined states and puzzles are answered with messages provided by the ideal adversary. This is necessary for capturing adversaries that construct different versions of a puzzle departing from different initial states of the original sequence $\mathtt{st}_0, \ldots, \mathtt{st}_\Gamma$ or from an arbitrary state that eventually leads to this sequence.

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

$\mathcal{F}_{\mathsf{tlp}}$ is parameterized by a set of parties $\mathcal{P}$, an owner $\mathcal{P}_o \in \mathcal{P}$, a computational security parameter τ, a state space $\mathcal{ST}$ and a tag space $\mathcal{TAG}$. In addition to $\mathcal{P}$ the functionality interacts with an adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{tlp}}$ contains initially empty lists steps (honest puzzle states), omsg (output messages), in (inbox) and out (outbox).

Create puzzle: Upon receiving the first message $(\mathsf{CreatePuzzle}, \mathsf{sid}, \Gamma, m)$ from $\mathcal{P}_o$ where $\Gamma \in \mathbb{N}^+$ and $m \in \{0,1\}^\tau$, proceed as follows:

1. If $\mathcal{P}_o$ is honest sample $\mathtt{tag} \xleftarrow{\$} \mathcal{TAG}$ and $\Gamma + 1$ random distinct states $\mathtt{st}_j \xleftarrow{\$} \{0,1\}^\tau$ for $j \in \{0, \ldots, \Gamma\}$. If $\mathcal{P}_o$ is corrupted, let $\mathcal{S}$ provide values $\mathtt{tag} \in \mathcal{TAG}$ and $\Gamma + 1$ distinct values $\mathtt{st}_j \in \mathcal{ST}$.
2. Append $(\mathtt{st}_0, \mathtt{tag}, \mathtt{st}_\Gamma, m)$ to omsg, append $(\mathtt{st}_j, \mathtt{st}_{j+1})$ to steps for $j \in \{0, \ldots, \Gamma - 1\}$, output $(\mathsf{CreatedPuzzle}, \mathsf{sid}, \mathsf{puz} = (\mathtt{st}_0, \Gamma, \mathtt{tag}))$ to $\mathcal{P}_o$ and $\mathcal{S}$. $\mathcal{F}_{\mathsf{tlp}}$ stops accepting messages of this form.

Solve: Upon receiving $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st})$ from party $\mathcal{P}_i \in \mathcal{P}$ with $\mathtt{st} \in \mathcal{ST}$, if there exists $(\mathtt{st}, \mathtt{st}') \in \mathsf{steps}$, append $(P_i, \mathtt{st}, \mathtt{st}')$ to in and ignore the next steps. If there is no $(\mathtt{st}, \mathtt{st}') \in \mathsf{steps}$, proceed as follows:

- If $\mathcal{P}_o$ is honest, sample $\mathtt{st}' \xleftarrow{\$} \mathcal{ST}$.
- If $\mathcal{P}_o$ is corrupted, send $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st})$ to $\mathcal{S}$ and wait for answer $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st}, \mathtt{st}')$.

Append $(\mathtt{st}, \mathtt{st}')$ to steps and append $(P_i, \mathtt{st}, \mathtt{st}')$ to in.

Get Message: Upon receiving $(\mathsf{GetMsg}, \mathsf{sid}, \mathsf{puz}, \mathtt{st})$ from party $\mathcal{P}_i \in \mathcal{P}$ with $\mathtt{st} \in \mathcal{ST}$, parse $\mathsf{puz} = (\mathtt{st}_0, \Gamma, \mathtt{tag})$ and proceed as follows:

- If $\mathcal{P}_o$ is honest and there is no $(\mathtt{st}_0, \mathtt{tag}, \mathtt{st}, m) \in \mathsf{omsg}$, append $(\mathtt{st}_0, \mathtt{tag}, \mathtt{st}, \bot)$ to omsg.
- If $\mathcal{P}_o$ is corrupted and there exists no $(\mathtt{st}_0, \mathtt{tag}, \mathtt{st}, m) \in \mathsf{omsg}$, send $(\mathsf{GetMsg}, \mathsf{sid}, \mathsf{puz}, \mathtt{st})$ to $\mathcal{S}$, wait for $\mathcal{S}$ to answer with $(\mathsf{GetMsg}, \mathsf{sid}, \mathsf{puz}, \mathtt{st}, m)$ and append $(\mathtt{st}_0, \mathtt{tag}, \mathtt{st}, m)$ to omsg.

Get $(\mathtt{st}_0, \mathtt{tag}, \mathtt{st}, m)$ from omsg and output $(\mathsf{GetMsg}, \mathsf{sid}, \mathtt{st}_0, \mathtt{tag}, \mathtt{st}, m)$ to $\mathcal{P}_i$.

Output: Upon receiving $(\mathsf{Output}, \mathsf{sid})$ by $\mathcal{P}_i \in \mathcal{P}$, retrieve the set L_i of all entries $(\mathcal{P}_i, \cdot, \cdot)$ in out, remove L_i from out and output $(\mathsf{Complete}, \mathsf{sid}, L_i)$ to $\mathcal{P}_i$.

Tick: Upon receiving $(\mathsf{comp\text{-}tick}, \mathsf{sid})$ from $\mathcal{S}$, set $\mathsf{out} \leftarrow \mathsf{in}$, set $\mathsf{in} = \emptyset$ and send $(\mathsf{comp\text{-}ticked}, \mathsf{sid})$ to $\mathcal{S}$.

Fig. 2. Functionality $\mathcal{F}_{\mathsf{tlp}}$ for time-lock puzzles.

5 Constructing Time-Lock Puzzles in UC

The functionality given in Fig. 2 from Section 4 describes how we ideally model a TLP in our framework. We will now instantiate $\mathcal{F}_{\mathsf{tlp}}$ departing from the well-known construction by Rivest et al. [38]. In order to obtain a UC-secure protocol, we will first model the assumption that underpins Rivest *et al.*'s construction under our notion of sequential computation with ticks. Moreover, we will resort to a global random oracle, which turns out to be *necessary* for UC-realizing $\mathcal{F}_{\mathsf{tlp}}$ as discussed later in this section.

The TLP construction of Rivest *et al.* [38] is based on the assumption that it is hard to compute successive squarings of an element of $(\mathbb{Z}/N\mathbb{Z})^\times$ (*i.e.* the group of primitive residues modulo N) with a large N in less time than it takes to compute each of the squarings sequentially, unless the factorization of N is known. In other words, for a random element $g \xleftarrow{\$} (\mathbb{Z}/N\mathbb{Z})^\times$ and a large N whose factorization is unknown, this assumptions says that it is hard to compute g^{2^Γ} in less time than it takes to compute Γ sequential squarings $g^2, g^{2^2}, g^{2^3}, \ldots, g^{2^\Gamma}$. On the other hand, if $N = pq$ is generated following the key generation algorithm of the RSA cryptosystem, one obtains a trapdoor (*i.e.* the order of $(\mathbb{Z}/N\mathbb{Z})^\times$) $\phi(N) = (p-1)(q-1)$ that allows for fast computation of g^{2^Γ} requiring two exponentiations: first compute $t = 2^\Gamma \mod \phi(N)$ and then g^t. Hence, a TLP encoding a message $m \in (\mathbb{Z}/N\mathbb{Z})^\times$ with a number of steps Γ can be generated by a party who knows $N = pq, p, q$ by sampling a random $g \xleftarrow{\$} (\mathbb{Z}/N\mathbb{Z})^\times$, computing $t = 2^\Gamma \mod \phi(N)$, $g^{2^\Gamma} = g^t$ and mg^{2^Γ}, arriving at a puzzle $\mathtt{puz} = (g, \Gamma, mg^{2^\Gamma})$. From the assumption of Rivest *et al.*, it follows that any party must compute Γ sequential squarings departing from g in order to obtain g^{2^Γ} and compute $m = mg^{2^\Gamma} g^{-2^\Gamma}$.

In employing Rivest *et al.*'s time-lock assumption to UC-realize $\mathcal{F}_{\mathsf{tlp}}$ we face an important challenge: even if the environment is computationally constrained in a session, it can use the representation of $(\mathbb{Z}/N\mathbb{Z})^\times$ (*i.e.* N) to compute all Γ squarings needed to obtain g^{2^Γ} from g across multiple sessions. Hence, it would be impossible to construct a simulator for a protocol realizing $\mathcal{F}_{\mathsf{tlp}}$, since the environment would be able to immediately extract the message encoded in the puzzle. Notice that an environment that can immediately solve a TLP makes it impossible for the simulator to provide a TLP containing a random message and later equivocate the opening of this TLP so that it yields an arbitrary message obtained from $\mathcal{F}_{\mathsf{tlp}}$. In order to address this issue, we need to model this time-lock assumption using our notion of sequential computation with ticks, which will limit the environment's power for computing squarings of elements of $(\mathbb{Z}/N\mathbb{Z})^\times$.

5.1 Modeling Rivest *et al.*'s Time-Lock Assumption [38]

We describe in Fig. 3 an ideal functionality $\mathcal{F}_{\mathsf{rsw}}$ that captures the hardness assumption used by Rivest et al. [38] to build a time-lock puzzle protocol. This functionality essentially treats group $(\mathbb{Z}/N\mathbb{Z})^\times$ as in the generic group model [40] and only gives handles to the group elements to all parties. In order to perform operations, the parties then need to interact with the functionality. They can ask for any number of operations to be performed between two computational ticks. However, the outcome of the operation (*i.e.* the handle of the resulting group element) will only be released after the next computational tick occurs. However, a special owner party $\mathcal{P}_o$ who initializes $\mathcal{F}_{\mathsf{rsw}}$ receives a trapdoor $\mathtt{td}$ that allows it to perform arbitrary operations on group elements. Upon learning $\mathtt{td}$ any party gains the power to perform arbitrary operations in $\mathcal{F}_{\mathsf{rsw}}$ but parties who do not know $\mathtt{td}$ are restricted to sequential operations and have no information

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

$\mathcal{F}_{\mathsf{rsw}}$ is parameterized by a set of parties $\mathcal{P}$, an owner $\mathcal{P}_o \in \mathcal{P}$, an adversary $\mathcal{S}$ and a computational security parameter τ and a parameter $N \in \mathbb{N}^+$. $\mathcal{F}_{\mathsf{rsw}}$ contains a map group which maps strings $\mathtt{el} \in \{0,1\}^\tau$ to $\mathbb{N}$ as well as maps in and out associating parties in $\mathcal{P}$ to a list of entries from $(\{0,1\}^\tau)^2$ or $(\{0,1\}^\tau)^3$. The functionality maintains the group of primitive residues modulo N with order $\phi(N)$ denoted as $(\mathbb{Z}/N\mathbb{Z})^\times$.

Create Group: Upon receiving the first message $(\mathsf{Create}, \mathsf{sid})$ from $\mathcal{P}_o$:

1. If $\mathcal{P}_i$ is corrupted then wait for message $(\mathsf{Group}, \mathsf{sid}, N, \phi(N))$ from $\mathcal{S}$ with $N \in \mathbb{N}^+, N < 2^\tau$ and store $N, \phi(N)$.
2. If $\mathcal{P}_o$ is honest then sample two random distinct prime numbers p, q of length approximately $\tau/2$ bits according to the RSA key generation procedure. Set $N = pq$ and $\phi(N) = (p-1)(q-1)$.
3. Set $\mathtt{td} = \phi(N)$ and output $(\mathsf{Created}, \mathsf{sid}, \mathtt{td})$ to $\mathcal{P}_o$.

Random: Upon receiving $(\mathsf{Rand}, \mathsf{sid}, \mathtt{td}')$ from $\mathcal{P}_i \in \mathcal{P}$, if $\mathtt{td}' \neq \mathtt{td}$, send $(\mathsf{Rand}, \mathsf{sid}, \mathsf{Invalid})$ to $\mathcal{P}_i$. Otherwise, sample $\mathtt{el} \xleftarrow{\$} \{0,1\}^\tau$ and $g \xleftarrow{\$} (\mathbb{Z}/N\mathbb{Z})^\times$, add $(\mathtt{el}, g)$ to group and output $(\mathsf{Rand}, \mathsf{sid}, \mathtt{el})$ to $\mathcal{P}_i$.

GetElement: Upon receiving $(\mathsf{GetElement}, \mathsf{sid}, \mathtt{td}', g)$ from $\mathcal{P}_i \in \mathcal{P}$, if $g \notin (\mathbb{Z}/N\mathbb{Z})^\times$ or $\mathtt{td}' \neq \mathtt{td}$, send $(\mathsf{GetElement}, \mathsf{sid}, \mathtt{td}', q, \mathsf{Invalid})$ to $\mathcal{P}_i$. Otherwise, if there exists an entry $(\mathtt{el}, g)$ in group then retrieve $\mathtt{el}$, else sample a random string $\mathtt{el}$ and add $(\mathtt{el}, g)$ to group. Output $(\mathsf{GetElement}, \mathsf{sid}, \mathtt{td}', g, \mathtt{el})$ to $\mathcal{P}_i$.

Power: Upon receiving $(\mathsf{Pow}, \mathsf{sid}, \mathtt{td}', \mathtt{el}, x)$ from $\mathcal{P}_i \in \mathcal{P}$ with $x \in \mathbb{Z}$, if $\mathtt{td}' \neq \mathtt{td}$ or $(\mathtt{el}, a) \notin \mathsf{group}$, output $(\mathsf{Pow}, \mathsf{sid}, \mathtt{td}', \mathtt{el}', x, \mathsf{Invalid})$ to $\mathcal{P}_i$. Otherwise, proceed:

1. Convert $x \in \mathbb{Q}$ into a representation $\overline{x} \in \mathbb{Z}_{\varphi(N)}$. If no such $\overline{x}$ exists in $\mathbb{Z}_{\varphi(N)}$ then output $(\mathsf{Pow}, \mathsf{sid}, \mathtt{td}', \mathtt{el}', x, \mathsf{Invalid})$ to $\mathcal{P}_i$.
2. Compute $y \leftarrow a^{\overline{x}} \bmod N$. If $(\mathtt{el}', y) \notin \mathsf{group}$ then sample $\mathtt{el}' \xleftarrow{\$} \{0,1\}^\tau$ randomly but different from all group entries and add $(\mathtt{el}', y)$ to group.
3. Output $(\mathsf{Pow}, \mathsf{sid}, \mathtt{td}, \mathtt{el}, x, \mathtt{el}')$ to $\mathcal{P}_i$.

Multiply: Upon receiving $(\mathsf{Mult}, \mathsf{sid}, \mathtt{el}_1, \mathtt{el}_2)$ from $\mathcal{P}_i \in \mathcal{P}$:

1. If $(\mathtt{el}_1, a) \notin \mathsf{group}$ or $(\mathtt{el}_2, b) \notin \mathsf{group}$, then output $(\mathsf{Invalid}, \mathsf{sid})$ to $\mathcal{P}_i$.
2. Compute $c \leftarrow ab \bmod N$. If $(\mathtt{el}_3, c) \notin \mathsf{group}$ then sample $\mathtt{el}_3 \xleftarrow{\$} \{0,1\}^\tau$ randomly but different from all group entries and add $(\mathtt{el}_3, c)$ to group.
3. Add $(\mathcal{P}_i, (\mathtt{el}_1, \mathtt{el}_2, \mathtt{el}_3))$ to in and return $(\mathsf{Mult}, \mathsf{sid}, \mathtt{el}_1, \mathtt{el}_2)$ to $\mathcal{P}_i$.

Invert: Upon receiving $(\mathsf{Inv}, \mathsf{sid}, \mathtt{el})$ from some party $\mathcal{P}_i \in \mathcal{P}$:

1. If $(\mathtt{el}, a) \notin \mathsf{group}$ then output $(\mathsf{Invalid}, \mathsf{sid})$ to $\mathcal{P}_i$.
2. Compute $y \leftarrow a^{-1} \bmod N$. If $(\mathtt{el}', y) \notin \mathsf{group}$ then sample $\mathtt{el}' \xleftarrow{\$} \{0,1\}^\tau$ randomly but different from all group entries and add $(\mathtt{el}', y)$ to group.
3. Add $(\mathcal{P}_i, (\mathtt{el}, \mathtt{el}'))$ to in and return $(\mathsf{Inv}, \mathsf{sid}, \mathtt{el})$ to $\mathcal{P}_i$.

Output: Upon receiving $(\mathsf{Output}, \mathsf{sid})$ by $\mathcal{P}_i \in \mathcal{P}$, retrieve the set L_i of all entries $(\mathcal{P}_i, \cdot)$ in out, remove L_i from out and output $(\mathsf{Complete}, \mathsf{sid}, L_i)$ to $\mathcal{P}_i$.

Tick: Upon receiving $(\mathsf{comp\text{-}tick}, \mathsf{sid})$ from $\mathcal{S}$, set $\mathsf{out} \leftarrow \mathsf{in}$, set $\mathsf{in} = \emptyset$ and send $(\mathsf{comp\text{-}ticked}, \mathsf{sid})$ to $\mathcal{S}$.

Fig. 3. Functionality $\mathcal{F}_{\mathsf{rsw}}$ capturing the time lock assumption of [38].

about the group representation. In particular, in case of an honestly generated group the order will remain completely hidden from the adversary. Finally, this functionality is treated as a global functionality in order to make sure that a simulator does not obtain an unreal advantage in computing the solution of a TLP without waiting for enough ticks.

We remark that our modeling of this time-lock assumption is corroborated by a recent result [39] showing that delay functions (such as TLP) based on cyclic groups that do not exploit any particular property of the underlying group cannot be constructed if the order is known. It is clear that we cannot reveal any information about the group structure to the environment, since it could use this information across multiple sessions to solve TLPs quicker than the parties. Hence, in order to make it possible to UC-realize $\mathcal{F}_{\mathsf{tlp}}$ based on cyclic groups (and in particular the time-lock assumption of Rivest et al. [38]), we must model the underlying group in such a way that both its structure and its order are hidden from the environment and the parties.

5.2 Realizing $\mathcal{F}_{\mathsf{tlp}}$ in the $\mathcal{F}_{\mathsf{rsw}}, \mathcal{G}_{\mathsf{rpoRO}}$-hybrid model

Using Rivest *et al.*'s time-lock assumptions modeled in $\mathcal{F}_{\mathsf{rsw}}$ following our sequential computation with ticks framework, we can instantiate Rivest *et al.*'s original time-lock puzzle without running into the issues described before. However, we now face a different issues: 1. because all parties are forced by $\mathcal{F}_{\mathsf{rsw}}$ to do sequential computation, a simulator for Rivest *et al.*'s construction would not be able to extract m from mg^{2^Γ}; 2. because $\mathcal{F}_{\mathsf{rsw}}$ deterministically assigns handles to each group element, a simulator would not be able to equivocate mg^{2^Γ} in such a way that it yields an arbitrary message m'. In order to address these issues, we must resort to a random oracle. More specifically, we work in the restricted programmable and observable global random oracle model $\mathcal{G}_{\mathsf{rpoRO}}$ of [13] (see Fig. 12 in Appendix A for the description). It turns out that a programmable random oracle is indeed necessary for UC-realizing $\mathcal{F}_{\mathsf{tlp}}$, since this functionality implies coin flipping with output independent abort as shown in Section 6, which is impossible without a programmable random oracle as we discuss in Section 7.

We present Protocol π_{tlp} in Figure 4. The main idea behind this protocol is to follow Rivest *et al.*'s construction to compute $\mathtt{puz} = (\mathtt{el}_0, \Gamma, \mathtt{tag})$ while encoding the initial random group element $\mathtt{el}_0$, the message m, the final group element $\mathtt{el}_\Gamma$ and the trapdoor $\mathtt{td}$ for $\mathcal{F}_{\mathsf{rsw}}$ in a $\mathtt{tag}$ generated with the help of the random oracle. This tag is generated in such a way that a party who solves the puzzle can retrieve $\mathtt{td}, m$ and test whether the tag is consistent with these values and with initial and final group elements $\mathtt{el}_0, \mathtt{el}_\Gamma$. More specifically, the tag $\mathtt{tag} = (\mathtt{tag}_2, \mathtt{tag}_2)$ is generated by computing $h_1 = H_1(\mathtt{el}_0|\mathtt{el}_\Gamma)$, $\mathtt{tag}_1 = h_1 \oplus (m|\mathtt{td})$ and $\mathtt{tag}_2 = H_2(h_1|m|\mathtt{td})$, where $H_1(\cdot), H_2(\cdot)$ are random oracles. A party who solves this puzzle obtaining $\mathtt{el}_\Gamma$ by performing Γ sequential squarings of $\mathtt{el}_0$ can retrieve h_1, obtain $(m|\mathtt{td})$ and check that these values are consistent with h_2. Notice that this also allows a simulator who observes queries to random oracles $H_1(\cdot), H_2(\cdot)$ to extract all parameters of a puzzle (including the message) and

Protocol π_{tlp}

Protocol π_{tlp} is parameterized by a security parameter τ, a state space $\mathcal{ST} = \{0,1\}^\tau$ and a tag space $\mathcal{TAG} = \{0,1\}^\tau \times \{0,1\}^\tau$. π_{tlp} is executed by an owner $\mathcal{P}_o$ and a set of parties $\mathcal{P}$ interacting among themselves and with functionalities $\mathcal{F}_{\mathsf{rsw}}$, $\mathcal{G}_{\mathsf{rpoRO1}}$ (an instance of $\mathcal{G}_{\mathsf{rpoRO}}$ with domain $\{0,1\}^{2*\tau}$ and output size $\{0,1\}^{2*\tau}$) and $\mathcal{G}_{\mathsf{rpoRO2}}$ (an instance of $\mathcal{G}_{\mathsf{rpoRO}}$ with domain $\{0,1\}^{3*\tau}$ and output size $\{0,1\}^{\tau}$).

Create Puzzle: Upon receiving input $(\mathsf{CreatePuzzle}, \mathsf{sid}, \Gamma, m)$ for $m \in \{0,1\}^\tau$, $\mathcal{P}_o$ proceeds as follows:

1. Send $(\mathsf{Create}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rsw}}$ obtaining $(\mathsf{Created}, \mathsf{sid}, \mathtt{td})$.
2. Send $(\mathsf{Rand}, \mathsf{sid}, \mathtt{td})$ to $\mathcal{F}_{\mathsf{rsw}}$, obtaining $(\mathsf{Rand}, \mathsf{sid}, \mathtt{el}_0)$.
3. Send $(\mathsf{Pow}, \mathsf{sid}, \mathtt{td}, \mathtt{el}_0, 2^\Gamma)$ to $\mathcal{F}_{\mathsf{rsw}}$, obtaining $(\mathsf{Pow}, \mathsf{sid}, \mathtt{td}, \mathtt{el}_0, 2^\Gamma, \mathtt{el}_\Gamma)$.
4. Send $(\textsc{Hash-Query}, (\mathtt{el}_0 | \mathtt{el}_\Gamma))$ to $\mathcal{G}_{\mathsf{rpoRO1}}$, obtaining $(\textsc{Hash-Confirm}, h_1)$.
5. Send $(\textsc{Hash-Query}, (h_1 | m | \mathtt{td}))$ to $\mathcal{G}_{\mathsf{rpoRO2}}$, obtaining $(\textsc{Hash-Confirm}, h_2)$.
6. Compute $\mathtt{tag}_1 = h_1 \oplus (m|\mathtt{td})$ and $\mathtt{tag}_2 = h_2$, set $\mathtt{tag} = (\mathtt{tag}_1, \mathtt{tag}_2)$ and output $(\mathsf{CreatedPuzzle}, \mathsf{sid}, \mathtt{puz} = (\mathtt{el}_0, \Gamma, \mathtt{tag}))$ to $\mathcal{P}_o$.

Solve: Upon receiving input $(\mathsf{Solve}, \mathsf{sid}, \mathtt{el})$, a party $\mathcal{P}_i \in \mathcal{P}$, send $(\mathsf{Mult}, \mathsf{sid}, \mathtt{el}, \mathtt{el})$ to $\mathcal{F}_{\mathsf{rsw}}$. If $\mathcal{P}_i$ obtains $(\mathsf{Invalid}, \mathsf{sid})$, it aborts.

Get Message: Upon receiving $(\mathsf{GetMsg}, \mathtt{puz}, \mathtt{el})$ as input, a party $\mathcal{P}_i \in \mathcal{P}$ parses $\mathtt{puz} = (\mathtt{el}_0, \Gamma, \mathtt{tag})$, parses $\mathtt{tag} = (\mathtt{tag}_1, \mathtt{tag}_2)$ and proceeds as follows:

1. Send $(\textsc{Hash-Query}, (\mathtt{el}_0 | \mathtt{el}))$ to $\mathcal{G}_{\mathsf{rpoRO1}}$, obtaining $(\textsc{Hash-Confirm}, h_1)$.
2. Compute $(m|\mathtt{td}) = \mathtt{tag}_1 \oplus h_1$ and send $(\textsc{Hash-Query}, (h_1|m|\mathtt{td}))$ to $\mathcal{G}_{\mathsf{rpoRO2}}$, obtaining $(\textsc{Hash-Confirm}, h_2)$.
3. Send $(\mathsf{Pow}, \mathsf{sid}, \mathtt{td}, \mathtt{el}_0, 2^\Gamma)$ to $\mathcal{F}_{\mathsf{rsw}}$, obtaining $(\mathsf{Pow}, \mathsf{sid}, \mathtt{td}, \mathtt{el}_0, 2^\Gamma, \mathtt{el}_\Gamma)$.
4. Send $(\textsc{IsProgrammed}, (\mathtt{el}_0|\mathtt{el}))$ and $(\textsc{IsProgrammed}, (h_1|m|\mathtt{td}))$ to $\mathcal{G}_{\mathsf{rpoRO1}}$ and $\mathcal{G}_{\mathsf{rpoRO2}}$, obtaining $(\textsc{IsProgrammed}, b_1)$ and $(\textsc{IsProgrammed}, b_2)$, respectively. Abort if $b_1 = 0$ or $b_2 = 0$,.
5. If $\mathtt{tag}_2 = h_2$ and $\mathtt{el} = \mathtt{el}_\Gamma$, output $(\mathsf{GetMsg}, \mathsf{sid}, \mathtt{el}_0, \mathtt{tag}, \mathtt{el}, m)$. Otherwise, output $(\mathsf{GetMsg}, \mathsf{sid}, \mathtt{el}_0, \mathtt{tag}, \mathtt{el}, \perp)$.

Output: Upon receiving $(\mathsf{Output}, \mathsf{sid})$ as input, a party $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{Output}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rsw}}$, receives $(\mathsf{Complete}, \mathsf{sid}, L_i)$ in response and outputs it.

Fig. 4. Protocol π_{tlp} realizing time-lock puzzle functionality $\mathcal{F}_{\mathsf{tlp}}$ in the $\mathcal{F}_{\mathsf{rsw}}, \mathcal{G}_{\mathsf{rpoRO}}$-hybrid model.

check whether it is a valid puzzle. A simulator who also has the additional (and provably necessary) power of programming the output of these random oracles can deliver an arbitrary message m' to a party who solves the puzzle.

We formally state the security of π_{tlp} in Theorem 2.

Theorem 2. *Protocol π_{tlp} UC-realizes $\mathcal{F}_{\mathsf{tlp}}$ in the $\mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{rsw}}$-hybrid model with computational security against a static adversary. Formally, there exists a simulator $\mathcal{S}$ such that for every static adversary $\mathcal{A}$, and any environment $\mathcal{Z}$, the environment cannot distinguish π_{tlp} composed with $\mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{rsw}}$ and $\mathcal{A}$ from $\mathcal{S}$ composed with $\mathcal{F}_{\mathsf{tlp}}$. That is:*

$$\mathsf{IDEAL}_{\mathcal{F}_{\mathsf{tlp}}, \mathcal{S}, \mathcal{Z}} \approx_{\mathsf{c}} \mathsf{HYBRID}^{\mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{rsw}}}_{\pi_{\mathsf{tlp}}, \mathcal{A}, \mathcal{Z}} .$$

Proof. In order to prove this theorem we construct a simulator $\mathcal{S}$ that interacts with the functionality $\mathcal{F}_{\mathsf{tlp}}$, the environment $\mathcal{Z}$ and an internal copy of the adversary $\mathcal{A}$, towards which it executes π_{tlp} and simulates $\mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{rsw}}$. We describe $\mathcal{S}$ separately for two important cases: Corrupted $\mathcal{P}_o$ in Figure 5 and Honest $\mathcal{P}_o$ in Figure 6. In the case of a corrupted $\mathcal{P}_o$, we focus on the steps necessary for dealing with an adversary $\mathcal{A}$ who corrupts $\mathcal{P}_o$ and leave the steps necessary for dealing with an $\mathcal{A}$ who also corrupts parties in $\mathcal{P}$ to the case of an honest $\mathcal{P}_o$.

In the case of a corrupted $\mathcal{P}_o$, the simulator $\mathcal{S}$ must extract the message m from a puzzle $\mathtt{puz} = (\mathtt{el}_0, \Gamma, \mathtt{tag} = (\mathtt{tag}_1, \mathtt{tag}_2))$ and deliver it to $\mathcal{F}_{\mathsf{tlp}}$ if $\mathtt{puz}$ is valid. In order to do this, $\mathcal{S}$ observes the queries to $\mathcal{G}_{\mathsf{rpoRO2}}$ and finds a query $(\textsc{Hash-Query}, (h_1|m|\mathtt{td}))$ from $\mathcal{A}$ to $\mathcal{G}_{\mathsf{rpoRO2}}$ for which there was a response $(\textsc{Hash-Confirm}, \mathtt{tag}_2)$, *i.e.* a query matching $\mathtt{tag}_2$. Using $\mathtt{td}$ and $\mathtt{el}_0$, $\mathcal{S}$ can compute $\mathtt{el}_\Gamma$ with the help of $\mathcal{F}_{\mathsf{rsw}}$. Finally, the simulator checks $\mathcal{G}_{\mathsf{rpoRO1}}$ returns $(\textsc{Hash-Confirm}, h_1)$ when queried with $(\textsc{Hash-Query}, (\mathtt{el}_0|\mathtt{el}_\Gamma))$. If this check succeeds $\mathcal{S}$ sends $(\mathsf{CreatePuzzle}, \mathsf{sid}, \Gamma, m)$ to $\mathcal{F}_{\mathsf{tlp}}$. This procedure works because $\mathcal{S}$ performs exactly the same computation that an honest party would in order to obtain and verify the solution to this puzzle, with the difference that it can perform all steps without waiting for the next tick, since it extracts $\mathtt{td}$ from $\mathtt{tag}_2$.

In the case of an honest $\mathcal{P}_o$, there is no need to extract the message from a puzzle. Instead, in this case we focus on dealing with an adversary $\mathcal{A}$ that corrupts parties in $\mathcal{P}$. Now the simulator $\mathcal{S}$ must provide a puzzle to $\mathcal{A}$ containing a random message in such a way that later on it can simulate a solution of this puzzle yielding an arbitrary message upon learning the ideal puzzle solution from $\mathcal{F}_{\mathsf{tlp}}$. The main point to be observed in this case, is that $\mathcal{S}$ provides a puzzle $\mathtt{puz} = (\mathtt{el}_0, \Gamma, \mathtt{tag})$ with a random $\mathtt{tag}$ that remains indistinguishable from an honestly generated puzzle until the moment $\mathcal{A}$ obtains $\mathtt{el}_\Gamma$ from $\mathcal{F}_{\mathsf{rsw}}$ after performing Γ sequential squarings. At this point, $\mathcal{A}$ would be able to query $\mathcal{G}_{\mathsf{rpoRO1}}$ and $\mathcal{G}_{\mathsf{rpoRO2}}$ and find out that this puzzle is invalid. However, since $\mathcal{S}$ sends the ticks to $\mathcal{F}_{\mathsf{tlp}}$, it can learn m immediately after sending the last of the Γ ticks needed for solving the puzzle and before any other party learns the outputs of operations performed with $\mathcal{F}_{\mathsf{rsw}}$ (*i.e.* learning $\mathtt{el}_\Gamma$). This allows $\mathcal{S}$ to program $\mathcal{G}_{\mathsf{rpoRO1}}$ and $\mathcal{G}_{\mathsf{rpoRO2}}$ with negligible probability of failure, since the probability $\mathcal{G}_{\mathsf{rpoRO1}}$ has been queried on $(\mathtt{el}_0|\mathtt{el}_\Gamma)$ is negligible before the $\mathtt{el}_\Gamma$ randomly sampled by $\mathcal{F}_{\mathsf{rsw}}$ is known by other parties. $\mathcal{S}$ can also program $\mathcal{G}_{\mathsf{rpoRO2}}$ with negligible probability of failure, since the probability of $\mathcal{G}_{\mathsf{rpoRO2}}$ having been queried on $(h_1|m|\mathtt{td})$ before these values become known is also negligible. □

Simulator $\mathcal{S}$ for the case of a corrupted $\mathcal{P}_o$ in π_{tlp}

Simulator $\mathcal{S}$ interacts with environment $\mathcal{Z}$, functionalities $\mathcal{F}_{\mathsf{tlp}}, \mathcal{G}_{\mathsf{rpoRO1}}, \mathcal{G}_{\mathsf{rpoRO2}}, \mathcal{F}_{\mathsf{rsw}}$ and an internal copy of an $\mathcal{A}$ corrupting $\mathcal{P}_o$. $\mathcal{S}$ forwards all messages between $\mathcal{A}$ and $\mathcal{Z}$. Moreover, $\mathcal{S}$ forwards all queries to $\mathcal{G}_{\mathsf{rpoRO1}}$, $\mathcal{G}_{\mathsf{rpoRO2}}$ and $\mathcal{F}_{\mathsf{rsw}}$ unless explicitly stated, keeping lists of all such requests, which are updated every time $\mathcal{S}$ checks these lists by appending the $\mathcal{Q}_s$ set of request obtained by sending (OBSERVE, sid) to $\mathcal{G}_{\mathsf{rpoRO1}}$ and $\mathcal{G}_{\mathsf{rpoRO2}}$. All queries to $\mathcal{G}_{\mathsf{rpoRO1}}$ or $\mathcal{G}_{\mathsf{rpoRO2}}$ made by $\mathcal{S}$ go through dummy honest parties so that the queries are not marked as illegitimate. $\mathcal{S}$ keeps a initially empty lists tag-$\overline{\mathsf{tag}}$, el-st, omsg.

Create Puzzle: Upon receiving a puzzle $\mathtt{puz}$ from $\mathcal{A}$, $\mathcal{S}$ proceeds as follows to check if the tag is valid with respect to the puzzle and extract the message m:

1. Parse $\mathtt{puz} = (\mathtt{el}_0, \Gamma, \mathtt{tag})$, parse $\mathtt{tag} = (\mathtt{tag}_1, \mathtt{tag}_2)$ and check that there exists a request (HASH-QUERY, $(h_1|m|\mathtt{td})$) from $\mathcal{A}$ to $\mathcal{G}_{\mathsf{rpoRO2}}$ for which there was a response (HASH-CONFIRM, $\mathtt{tag}_2$).
2. Send $(\mathsf{Pow}, \mathsf{sid}, \mathtt{td}, \mathtt{el}_0, 2^\Gamma)$ to $\mathcal{F}_{\mathsf{rsw}}$, obtaining $(\mathsf{Pow}, \mathsf{sid}, \mathtt{td}, \mathtt{el}, 2^\Gamma, \mathtt{el}_\Gamma)$. Check that there exists a request (HASH-QUERY, $(\mathtt{el}_\Gamma|\mathtt{el}_\Gamma)$) from $\mathcal{A}$ to $\mathcal{G}_{\mathsf{rpoRO1}}$ for which there was a response (HASH-CONFIRM, h_1).
3. Check that $(m|\mathtt{td}) = \mathtt{tag}_1 \oplus h_1$.

If any of the checks above fail, it means that verifying the opening of this puzzle will always fail, so $\mathcal{S}$ sets $m = \perp$. $\mathcal{S}$ proceeds as follows to simulate the creation of a puzzle with message m:

1. For $j \in \{0, \ldots, \Gamma\}$, sample $\mathtt{st}_j \xleftarrow{\$} \{0,1\}^\tau$, add $(\mathtt{el}_j, \mathtt{st}_j)$ to el-st and send $(\mathsf{Pow}, \mathsf{sid}, \mathtt{td}, \mathtt{el}_j, 2)$ to $\mathcal{F}_{\mathsf{rsw}}$, obtaining $(\mathsf{Pow}, \mathsf{sid}, \mathtt{td}, \mathtt{st}_i, 2, \mathtt{el}_{j+1})$.
2. Sample $\overline{\mathtt{tag}} \xleftarrow{\$} \mathcal{TAG}$, append $(\mathtt{tag}, \overline{\mathtt{tag}})$ to tag-$\overline{\mathsf{tag}}$ and append $(\mathtt{st}_0, \mathtt{tag}, \mathtt{st}_\Gamma, m)$ to omsg.
3. Send $(\mathsf{CreatePuzzle}, \mathsf{sid}, \Gamma, m)$ to $\mathcal{F}_{\mathsf{tlp}}$ and provide $\mathtt{st}_0, \ldots, \mathtt{st}_\Gamma, \mathtt{tag}$.

Solve: Upon receiving $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st})$ from $\mathcal{F}_{\mathsf{tlp}}$, $\mathcal{S}$ proceeds as follows:

- If there is $(\mathtt{el}, \mathtt{st}) \in \mathsf{el}$-$\mathsf{st}$, send $(\mathsf{Pow}, \mathsf{sid}, \mathtt{td}, \mathtt{el}, 2)$ to $\mathcal{F}_{\mathsf{rsw}}$, obtaining $(\mathsf{Pow}, \mathsf{sid}, \mathtt{td}, \mathtt{st}, 2, \mathtt{el}')$.
- If there is no $(\mathtt{el}, \mathtt{st}) \in \mathsf{el}$-$\mathsf{st}$, send $(\mathsf{Rand}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rsw}}$, obtaining $(\mathsf{Rand}, \mathsf{sid}, \mathtt{el}')$.

Sample $\mathtt{st}' \xleftarrow{\$} \{0,1\}^\tau$ and add $(\mathtt{el}', \mathtt{st}')$ to el-st. Finally, send $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st}, \mathtt{st}')$ to $\mathcal{F}_{\mathsf{tlp}}$.

Get Message: Upon receiving $(\mathsf{GetMsg}, \mathsf{sid}, \mathtt{puz}, \mathtt{st})$ from $\mathcal{F}_{\mathsf{tlp}}$, $\mathcal{S}$ parses $\mathtt{puz} = (\mathtt{st}_0, \Gamma, \mathtt{tag})$ and proceeds as follows:

1. Check that there exist entries $(\mathtt{el}_0, \mathtt{st}_0)$ and $(\mathtt{el}, \mathtt{st})$ in el-st and $(\mathtt{tag}, \overline{\mathtt{tag}})$ in tag-$\overline{\mathsf{tag}}$, using $\mathtt{el}_0, \mathtt{el}, \mathtt{tag}$ for the remaining checks.
2. Check that the tag $\mathtt{tag} = (\mathtt{tag}_1, \mathtt{tag}_2)$ is valid with respect to the puzzle $\mathtt{puz}$ and the solution $\mathtt{el}$ by proceeding as in the protocol: Send (HASH-QUERY, $(\mathtt{el}_0|\mathtt{el})$) to $\mathcal{G}_{\mathsf{rpoRO1}}$, obtain(HASH-CONFIRM, h_1), compute $(m|\mathtt{td}) = \mathtt{tag}_1 \oplus h_1$, send (HASH-QUERY, $(h_1|m|\mathtt{td})$) to $\mathcal{G}_{\mathsf{rpoRO2}}$, obtain (HASH-CONFIRM, h_2), send $(\mathsf{Pow}, \mathsf{sid}, \mathtt{td}, \mathtt{el}_0, 2^\Gamma)$ to $\mathcal{F}_{\mathsf{rsw}}$, obtaining $(\mathsf{Pow}, \mathsf{sid}, \mathtt{td}, \mathtt{st}_0, 2^\Gamma, \mathtt{el}_\Gamma)$. Check that $\mathtt{tag}_2 = h_2$ and $\mathtt{el} = \mathtt{el}_\Gamma$.

If the above checks are successful, $\mathcal{S}$ sends $(\mathsf{GetMsg}, \mathsf{sid}, \mathtt{st}_0, \mathtt{tag}, \mathtt{st}, m)$ to $\mathcal{F}_{\mathsf{tlp}}$. Otherwise, $\mathcal{S}$ sends $(\mathsf{GetMsg}, \mathsf{sid}, \mathtt{st}_0, \mathtt{tag}, \mathtt{st}, \perp)$ to $\mathcal{F}_{\mathsf{tlp}}$.

Ticks: $\mathcal{S}$ sends tick messages to all functionalities when activated if all honest parties have been activated after the last tick.

Fig. 5. Simulator $\mathcal{S}$ for the case of a corrupted $\mathcal{P}_o$ in π_{tlp} .

Simulator $\mathcal{S}$ for the case of an honest $\mathcal{P}_o$ in π_{tlp}

Simulator $\mathcal{S}$ interacts with environment $\mathcal{Z}$, functionalities $\mathcal{F}_{\mathsf{tlp}}, \mathcal{G}_{\mathsf{rpoRO1}}, \mathcal{G}_{\mathsf{rpoRO2}}, \mathcal{F}_{\mathsf{rsw}}$ and an internal copy of an $\mathcal{A}$ corrupting one or more parties $\mathcal{P}_i \in \mathcal{P} \setminus \mathcal{P}_o$. $\mathcal{S}$ forwards all messages between $\mathcal{A}$ and $\mathcal{Z}$. Moreover, $\mathcal{S}$ forwards all queries to $\mathcal{G}_{\mathsf{rpoRO1}}$, $\mathcal{G}_{\mathsf{rpoRO2}}$ and $\mathcal{F}_{\mathsf{rsw}}$ unless explicitly stated, keeping lists of all such requests. However, for every query $(\text{IsProgrammed}, m)$ to $\mathcal{G}_{\mathsf{rpoRO1}}$ or $\mathcal{G}_{\mathsf{rpoRO2}}$, $\mathcal{S}$ always answers with $(\text{IsProgrammed}, 0)$. $\mathcal{S}$ keeps an initially empty lists $\mathsf{el\text{-}st}, \mathsf{omsg}, \mathsf{next}$.

Create Puzzle: Upon receiving $(\mathsf{CreatedPuzzle}, \mathsf{sid}, \mathsf{puz} = (\mathtt{st}_0, \Gamma, \overline{\mathtt{tag}}))$ from $\mathcal{F}_{\mathsf{tlp}}$, $\mathcal{S}$ proceeds as follows to create a puzzle $(\mathtt{el}_0, \Gamma, \mathtt{tag})$ that can be later programmed to yield an arbitrary message obtained from $\mathcal{F}_{\mathsf{tlp}}$:

1. Sample a random $m \xleftarrow{\$} \{0,1\}^\tau$ and $\mathtt{tag}_1 \xleftarrow{\$} \{0,1\}^{2\tau}$ and $\mathtt{tag}_2 \xleftarrow{\$} \{0,1\}^\tau$.
2. Send $(\mathsf{Create}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rsw}}$ obtaining $(\mathsf{Created}, \mathsf{sid}, \mathtt{td})$. Send $(\mathsf{Rand}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rsw}}$, obtaining $(\mathsf{Rand}, \mathsf{sid}, \mathtt{el}_0)$. Send $(\mathsf{Pow}, \mathsf{sid}, \mathtt{td}, \mathtt{el}, 2^\Gamma)$ to $\mathcal{F}_{\mathsf{rsw}}$, obtaining $(\mathsf{Pow}, \mathsf{sid}, \mathtt{td}, \mathtt{el}, 2^\Gamma, \mathtt{el}_\Gamma)$.
3. Append $(\mathtt{el}_0, \mathtt{st}_0)$ to $\mathsf{el\text{-}st}$, set $\mathtt{tag} = (\mathtt{tag}_1, \mathtt{tag}_2)$, append $(\mathtt{tag}, \overline{\mathtt{tag}})$ to $\mathsf{tag}\text{-}\overline{\mathsf{tag}}$ and output $(\mathsf{CreatedPuzzle}, \mathsf{sid}, \mathsf{puz} = (\mathtt{el}_0, \Gamma, \mathtt{tag}))$.

Solve: If $\mathcal{A}$ makes a query $(\mathsf{Mult}, \mathsf{sid}, \mathtt{el}, \mathtt{el})$ to $\mathcal{F}_{\mathsf{rsw}}$ on behalf of $\mathcal{P}_i \in \mathcal{P} \setminus \mathcal{P}_o$ such there exists an entry $(\mathtt{el}, \mathtt{st})$ in $\mathsf{el\text{-}st}$, $\mathcal{S}$ proceeds as follows:

1. Send $(\mathsf{Pow}, \mathsf{sid}, \mathtt{td}, \mathtt{el}, 2)$ to $\mathcal{F}_{\mathsf{rsw}}$, obtaining $(\mathsf{Pow}, \mathsf{sid}, \mathtt{td}, \mathtt{el}, 2, \mathtt{el}')$.
2. If there is no entry $(\mathtt{el}', \mathtt{st}')$ in $\mathsf{el\text{-}st}$, append $(\mathtt{el}', \mathtt{st})$ to next and send $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st})$ to $\mathcal{F}_{\mathsf{tlp}}$ on behalf of $\mathcal{P}_i$.

Get Message: Forward queries to $\mathcal{G}_{\mathsf{rpoRO1}}$, $\mathcal{G}_{\mathsf{rpoRO2}}$ and $\mathcal{F}_{\mathsf{rsw}}$ from $\mathcal{A}$ on behalf of corrupted parties $\mathcal{P}_i \in \mathcal{P} \setminus \mathcal{P}_o$, allowing $\mathcal{A}$ to perform the necessary steps for GetMessage. However, for every query $(\text{IsProgrammed}, m)$ to $\mathcal{G}_{\mathsf{rpoRO1}}$ or $\mathcal{G}_{\mathsf{rpoRO2}}$, $\mathcal{S}$ always answers with $(\text{IsProgrammed}, 0)$.

Ticks: $\mathcal{S}$ sends tick messages to all functionalities when activated if all honest parties have been activated after the last tick. Immediately after each tick, if $\mathcal{S}$ sent a query $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st})$ to $\mathcal{F}_{\mathsf{tlp}}$ before this tick, it sends $(\mathsf{Output}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{tlp}}$ on behalf of each corrupted $\mathcal{P}_i \mathcal{P} \setminus \mathcal{P}_o$, obtaining $(\mathsf{Output}, \mathsf{sid}, L_i)$. For each L_i and each entry $(\mathcal{P}_i, \mathtt{st}, \mathtt{st}') \in L_i$, $\mathcal{S}$ proceeds as follows:

1. If there exists an entry $(\mathtt{el}', \mathtt{st})$ in next, remove $(\mathtt{el}', \mathtt{st})$ from next and append $(\mathtt{el}', \mathtt{st}')$ to $\mathsf{el\text{-}st}$.
2. If there is an entry $(\mathtt{el}_\Gamma, \mathtt{st}')$ in $\mathsf{el\text{-}st}$, it means $\mathcal{A}$ should be able to execute GetMessage and obtain message m in puzzle puz when activated after this tick. $\mathcal{S}$ proceeds as follows to program the global random oracles so that executing GetMessage with $(\mathtt{el}_0, \Gamma, \mathtt{tag}), \mathtt{el}_\Gamma$ will return m:
   (a) Send $(\mathsf{GetMsg}, \mathsf{sid}, \mathsf{puz}, \mathtt{st}')$ to $\mathcal{F}_{\mathsf{tlp}}$, obtaining $(\mathsf{GetMsg}, \mathsf{sid}, \mathsf{puz}, \mathtt{st}', m)$.
   (b) Compute $h_1 = \mathtt{tag}_1 \oplus (m|\mathtt{td})$ and send $(\text{Program-RO}, (\mathtt{el}_0|\mathtt{el}_\Gamma), h_1)$ to $\mathcal{G}_{\mathsf{rpoRO1}}$. Since $\mathtt{el}_\Gamma$ is randomly chosen by $\mathcal{F}_{\mathsf{rsw}}$ and still unknown to $\mathcal{A}$, $\mathcal{Z}$ or any other party at this point, the probability that this programming fails in negligible.
   (c) Send $(\text{Program-RO}, (h_1|m|\mathtt{td})), h_2)$ to $\mathcal{G}_{\mathsf{rpoRO2}}$. Since h_1 is randomly chosen by $\mathcal{S}$ and still unknown to $\mathcal{A}$, $\mathcal{Z}$ or any other party at this point, the probability that this programming fails in negligible.

Fig. 6. Simulator $\mathcal{S}$ for the case of an honest $\mathcal{P}_o$ in π_{tlp}.

6 Secure Two-Party Computation with Output-Independent Abort

We now discuss how to compile any 2PC with secret-shared outputs into a 2PC with output-independent abort using homomorphic commitments with delayed opening. We focus on the two-party setting, although our construction could be generalized to the multi-party setting as we will describe.

6.1 Functionalities

We begin by describing the functionalities that are used in this section and which have not appeared before when modeled with respect to time:

- The functionality $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ (Fig. 7) for Output-Independent 2PC.
- The functionality $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ (Fig. 8 and Fig. 9) for secure 2PC with secret-shared output which naturally arises from existing protocols.
- The functionality $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ (Fig. 10) for homomorphic commitments with delayed non-interactive openings that naturally arises from homomorphic commitments that are combined with $\mathcal{F}_{\mathsf{tlp}}$.

An additional functionality $\mathcal{F}_{\mathsf{ct}}$ for coin-flipping with abort in the timed message model appears in Appendix A.

All of the functionalities assume that one of the parties is honest while the other is corrupted, but this is only for simplicity of exposition of the functionalities. We choose to write functionalities where the parties have to send messages to trigger "regular behavior" instead of giving full one-sided control to $\mathcal{S}$ as this appears more natural. Messages to dishonest parties, on the other hand, go directly to $\mathcal{S}$ that can act upon them.

2PC with Output-Independent Abort. The functionality $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ as outlined in Fig. 7 shows how Output-Independent Abort for 2PC can be modeled. Similar to other 2PC functionalities, it allows parties to fix the circuit C to be computed, provide inputs, compute with these inputs and then output the result of the computation. In comparison to regular UC functionalities, there are two distinct differences how this is handled:

- Parties using $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ do not receive messages from $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ in a push-model where they get activated upon each new message, but instead they have to pull messages from the functionality (which was also already the case for $\mathcal{F}_{\mathsf{smt,delay}}^{\Delta}$). The reasoning behind this is that the functionality is ticked and it might happen that multiple messages arrive to multiple receivers in the same "tick" round. But upon receiving a message from $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$, a party may not return activation to it. This means that another "tick" may happen before another message gets delivered, which would break the guaranteed delivery requirement. A pull-model is a solution as each party is guaranteed to get activated between any two "ticks" in our model, allowing it to receive messages if it wants to. We will also use this modeling for the other functionalities in this section.

Functionality $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$

The functionality runs with parties $\mathcal{P}_1, \mathcal{P}_2$ and an adversary $\mathcal{S}$ who may corrupt either of the parties. It is parameterized by parameters $\Delta, \delta \in \mathbb{N}^+$. The computed circuit is defined over $\mathbb{F}_2$. The functionality internally has two lists $\mathcal{M}, \mathcal{Q}$ and flags $\mathtt{output}, \mathtt{noabort} \leftarrow \bot$.

Init: On input $(\mathsf{Init}, \mathsf{sid}, C)$ by $\mathcal{P}_i \in \{\mathcal{P}_1, \mathcal{P}_2\}$:
1. Add $(\Delta, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_{3-i}, (\mathsf{Init}, C))$ to $\mathcal{Q}$ for an unused mid.
2. If both parties sent $(\mathsf{Init}, \mathsf{sid}, C)$ then store C locally.
3. Send $(\mathsf{Init}, \mathsf{sid}, \mathcal{P}_i, C, \mathsf{mid})$ to $\mathcal{S}$.

Input: On first input $(\mathsf{Input}, \mathsf{sid}, i, x_i)$ by $\mathcal{P}_i$ for $i \in \{1, 2\}$:
1. Add $(\Delta, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_{3-i}, (\mathsf{Input}, \mathcal{P}_i))$ to $\mathcal{Q}$ for an unused mid.
2. Accept x_i as input for $\mathcal{P}_i$.
3. Send $(\mathsf{Input}, \mathsf{sid}, \mathcal{P}_i, x_i, \mathsf{mid})$ to $\mathcal{S}$ if $\mathcal{P}_i$ is corrupted and $(\mathsf{Input}, \mathsf{sid}, \mathcal{P}_i, \mathsf{mid})$ otherwise.

Computation: On first input $(\mathsf{Compute}, \mathsf{sid})$ by $\mathcal{P}_i \in \{\mathcal{P}_1, \mathcal{P}_2\}$ and if both x_1, x_2 were accepted:
1. Add $(\Delta, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_{3-i}, (\mathsf{Compute}))$ to $\mathcal{Q}$ for an unused mid.
2. If both parties sent $(\mathsf{Compute}, \mathsf{sid})$ compute $y = C(x_1, x_2)$ and store y.
3. Send $(\mathsf{Compute}, \mathsf{sid}, \mathcal{P}_i, \mathsf{mid})$ to $\mathcal{S}$.

Output: On first input $(\mathsf{Output}, \mathsf{sid})$ by both parties and if y has been stored:
1. Add $(\Delta + \delta, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_j, (\mathsf{Output}, y))$ to $\mathcal{Q}$ for the honest party $\mathcal{P}_j$ and for some unused mid.
2. Set $\mathtt{output} \leftarrow \mathsf{mid}$ and send $(\mathsf{OutputOrAbort}, \mathsf{sid}, \mathsf{mid})$ to $\mathcal{S}$.

Fetch Message: Upon receiving $(\mathsf{FetchMsg}, \mathsf{sid})$ by $\mathcal{P} \in \{\mathcal{P}_1, \mathcal{P}_2\}$ retrieve the set L of all entries $(\mathcal{P}, \mathsf{sid}, \cdot)$ in $\mathcal{M}$, remove L from $\mathcal{M}$ and return $(\mathsf{FetchMsg}, \mathsf{sid}, L)$ to $\mathcal{P}$.

Scheduling: On input of $\mathcal{S}$:
- If $\mathcal{S}$ sent $(\mathsf{Deliver}, \mathsf{sid}, \mathsf{mid})$ and $\mathsf{mid} \neq \mathtt{output}$ then remove each $(\mathtt{cnt}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}, m)$ from $\mathcal{Q}$ and add $(\mathcal{P}, \mathsf{sid}, m)$ to $\mathcal{M}$.
- If $\mathcal{S}$ sent $(\mathsf{DeliverOrAbort}, \mathsf{sid}, \mathsf{mid}, f)$ and $\mathsf{mid} = \mathtt{output} \neq \bot$:
  if $f = \bot$ then replace $(\mathtt{cnt}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, (\mathsf{Output}, y))$ in $\mathcal{Q}$ with $(\mathtt{cnt}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, (\mathsf{NoOutput}))$.
  if $f = \top$ then leave the entry in $\mathcal{Q}$ unchanged.
  Then set $\mathtt{noabort} \leftarrow \top$, $\mathtt{output} \leftarrow \bot$ and send $(\mathsf{Output}, \mathsf{sid}, y)$ to $\mathcal{S}$.
- If $\mathcal{S}$ sent $(\mathsf{Abort}, \mathsf{sid})$ and $\mathtt{noabort} = \bot$ then add $(\mathcal{P}_1, \mathsf{sid}, \mathsf{Abort})$, $(\mathcal{P}_2, \mathsf{sid}, \mathsf{Abort})$ to $\mathcal{M}$ and ignore all further calls to the functionality except to **Fetch Message**.

Tick: Upon receiving $(\mathsf{Tick}, \mathsf{sid})$ from $\mathcal{S}$:
1. For each query $(0, \mathsf{mid}, \mathsf{sid}, \mathcal{P}, m) \in \mathcal{Q}$:
   (a) Remove $(0, \mathsf{mid}, \mathsf{sid}, \mathcal{P}, m)$ from $\mathcal{Q}$.
   (b) Add $(\mathcal{P}, \mathsf{sid}, m)$ to $\mathcal{M}$.
2. Replace each $(\mathtt{cnt}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}, m)$ in $\mathcal{Q}$ with $(\mathtt{cnt} - 1, \mathsf{mid}, \mathsf{sid}, \mathcal{P}, m)$.

Fig. 7. The $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ Functionality for 2PC with Output-Independent Abort.

– The functionality does not directly deliver messages to receivers, but instead internally queries them first. This is because it is necessary to use communication using $\mathcal{F}_{\mathsf{smt,delay}}^{\Delta}$, which means that the adversary may arbitrarily control how messages get delivered, and he may reorder delivery at his will within the maximal delay that $\mathcal{F}_{\mathsf{smt,delay}}^{\Delta}$ permits. We also allow the adversary to influence delivery "adaptively", meaning depending on other events outside of $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$'s scope. We could theoretically model this freedom of $\mathcal{S}$ as macros in the functionality's definition, but believe that this explicit modeling is better suited to our framework.

Towards achieving this pull-model and adversarial reordering of messages, $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ has two internal lists $\mathcal{Q}$ and $\mathcal{M}$. $\mathcal{Q}$ contains all the buffered messages which can be delivered in the future, while messages in $\mathcal{M}$ can be retrieved right now by the respective receivers. When **Tick** is called $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ will then move all messages from $\mathcal{Q}$ to $\mathcal{M}$ which get available in the next round, and which can be retrieved via the interface **Fetch Message**.

$\mathcal{S}$ may use **Scheduling** to prematurely move messages to $\mathcal{M}$ by sending a special message that contains the message id mid — that means that $\mathcal{S}$ gets notified about every new mid whenever a message is added to $\mathcal{Q}$ which $\mathcal{S}$ can influence. $\mathcal{S}$ may also cancel the delivery of messages, though this will lead to a break-down of the functionality as $\mathcal{F}_{\mathsf{smt,delay}}^{\Delta}$ does not allow to drop messages altogether.

Additionally, we let **Scheduling** be responsible to realize the output-independent abort property of $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$. To see why this is the case, observe that once both parties activate the output phase the functionality stores the message id `output` that represents the output in $\mathcal{Q}$. The adversary can then send a message DeliverOrAbort and will thereby learn the output y only after deciding if the honest party obtains it as well: once it allows delivery or denies it via DeliverOrAbort then the Abort command cannot be used anymore. Furthermore, DeliverOrAbort can only be called once and thereby fixes the message that will eventually be delivered to the honest party in $\mathcal{Q}$. Once this decision was made, $\mathcal{S}$ can then adjust the delivery time via Deliver as before. If $\mathcal{S}$ never calls DeliverOrAbort then both $\mathcal{S}$ and the honest party learn the output at the same time. In that case, even after calling Abort the honest party can still read the output from $\mathcal{M}$.

Two-Party Computation with Secret-Shared Output. In Fig. 8 and Fig. 9 we describe a 2PC functionality $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ which will be the foundation for our compiler that will realise $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$. $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ has the same initialization, input and computation interfaces as other 2PC functionalities. The two main differences between a standard 2PC functionality and $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$: first, $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ is again a "ticked" functionality, meaning that it similarly to $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ considers a 2PC protocol that implements communication via $\mathcal{F}_{\mathsf{smt,delay}}^{\Delta}$. Second, $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ does not directly output the outcome of the computation. Instead, it reveals a secret-sharing of it to both parties. The parties can then manipulate shares using the functionality, generate additional random shares or reconstruct them.

Functionality $\mathcal{F}^{\Delta}_{\mathsf{2pcsso}}$ (Computation, Message Handling)

The functionality interacts with two parties $\mathcal{P}_1, \mathcal{P}_2$ and an adversary $\mathcal{S}$ which may corrupt either of the parties. The functionality will internally have two lists $\mathcal{M}, \mathcal{Q}$.

Init: On input $(\mathsf{Init}, \mathsf{sid}, C)$ by $\mathcal{P}_i \in \{\mathcal{P}_1, \mathcal{P}_2\}$:
1. Add $(\Delta, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_{3-i}, (\mathsf{Init}, C))$ to $\mathcal{Q}$ for an unused mid.
2. If both parties sent $(\mathsf{Init}, \mathsf{sid}, C)$ then store C locally and let m be the length of the output of C.
3. Send $(\mathsf{Init}, \mathsf{sid}, \mathcal{P}_i, C, \mathsf{mid})$ to $\mathcal{S}$.

Input: On first input $(\mathsf{Input}, \mathsf{sid}, i, x_i)$ by $\mathcal{P}_i$ for $i \in \{1, 2\}$:
1. Add $(\Delta, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_{3-i}, (\mathsf{Input}, \mathcal{P}_i))$ to $\mathcal{Q}$ for an unused mid.
2. Accept x_i as input for $\mathcal{P}_i$.
3. Send $(\mathsf{Input}, \mathsf{sid}, \mathcal{P}_i, x_i, \mathsf{mid})$ to $\mathcal{S}$ if $\mathcal{P}_i$ is corrupted and $(\mathsf{Input}, \mathsf{sid}, \mathcal{P}_i, \mathsf{mid})$ otherwise.

Computation: On first input $(\mathsf{Compute}, \mathsf{sid})$ by $\mathcal{P}_i \in \{\mathcal{P}_1, \mathcal{P}_2\}$ and if both x_1, x_2 were accepted:
1. Add $(\Delta, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_{3-i}, (\mathsf{Compute}))$ to $\mathcal{Q}$ for an unused mid.
2. If both parties sent $(\mathsf{Compute}, \mathsf{sid})$ compute $\boldsymbol{y} = (y_1, \ldots, y_m) \leftarrow C(x_1, x_2)$ and store $\boldsymbol{y}$.
3. Send $(\mathsf{Compute}, \mathsf{sid}, \mathcal{P}_i, \mathsf{mid})$ to $\mathcal{S}$.

Fetch Message: Upon receiving $(\mathsf{FetchMsg}, \mathsf{sid})$ by $\mathcal{P} \in \{\mathcal{P}_1, \mathcal{P}_2\}$ retrieve the set L of all entries $(\mathcal{P}, \mathsf{sid}, \cdot)$ in $\mathcal{M}$, remove L from $\mathcal{M}$ and return $(\mathsf{Output}, \mathsf{sid}, L)$ to $\mathcal{P}$.

Scheduling: On input of $\mathcal{S}$:
- If $\mathcal{S}$ sent $(\mathsf{Deliver}, \mathsf{sid}, \mathsf{mid})$ then remove each $(\mathtt{cnt}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}, m)$ from $\mathcal{Q}$ and add $(\mathcal{P}, \mathsf{sid}, m)$ to $\mathcal{M}$.
- If $\mathcal{S}$ sent (Abort) add $(\mathcal{P}_S, \mathsf{sid}, \mathsf{Abort})$, $(\mathcal{P}_R, \mathsf{sid}, \mathsf{Abort})$ to $\mathcal{M}$ and ignore all further calls to the functionality except to **Fetch Message**.

Tick: Upon receiving $(\mathsf{Tick}, \mathsf{sid})$ from $\mathcal{S}$:
1. For each query $(0, \mathsf{mid}, \mathsf{sid}, \mathcal{P}, m) \in \mathcal{Q}$:
   (a) Remove $(0, \mathsf{mid}, \mathsf{sid}, \mathcal{P}, m)$ from $\mathcal{Q}$.
   (b) Add $(\mathcal{P}, \mathsf{sid}, m)$ to $\mathcal{M}$.
2. Replace each $(\mathtt{cnt}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}, m)$ in $\mathcal{Q}$ with $(\mathtt{cnt} - 1, \mathsf{mid}, \mathsf{sid}, \mathcal{P}, m)$.

Fig. 8. Functionality $\mathcal{F}^{\Delta}_{\mathsf{2pcsso}}$ for 2PC with Secret-Shared Output and Linear Secret Share Operations.

We will not show in this work how to realize $\mathcal{F}^{\Delta}_{\mathsf{2pcsso}}$. This is because it's output-sharing property is rather standard (albeit not always modeled as explicitly as here) and it follows directly from any 2PC protocol that is entirely based on secret-sharing or on BMR protocols that secret-share the output. For example, [6] show how to implement a multi-party version of $\mathcal{F}^{\Delta}_{\mathsf{2pcsso}}$ (albeit without abstract time) using the BMR protocol of [26].

Additively Homomorphic Commitments with Delayed Openings. In order to implement $\mathcal{F}^{\Delta,\delta}_{\mathsf{2pcoia}}$ we also need a special commitment scheme that al-

Functionality $\mathcal{F}^{\Delta}_{\mathsf{2pcsso}}$ (Computation on Outputs)

Share Output: Upon input $(\mathsf{ShareOutput}, \mathsf{sid}, \mathcal{I})$ by $\mathcal{P}_i \in \{\mathcal{P}_1, \mathcal{P}_2\}$ for fresh identifiers $\mathcal{I} = \{\mathsf{cid}_1, \ldots, \mathsf{cid}_m\}$ and if **Computation** was finished:

1. Add $(\Delta, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_{3-i}, (\mathsf{ShareOutput}))$ to $\mathcal{Q}$ for an unused mid. Then send $(\mathsf{ShareOutput}, \mathsf{sid}, \mathcal{P}_i, \mathsf{mid})$ to $\mathcal{S}$.
2. If both parties sent **ShareOutput** (and letting $\mathcal{P}_j$ be the corrupted party):
   (a) Send $(\mathsf{RequestShares}, \mathsf{sid}, \mathcal{I})$ to $\mathcal{S}$, which replies with $(\mathsf{OutputShares}, \mathsf{sid}, \{(\mathsf{cid}, s_{j,\mathsf{cid}})\}_{\mathsf{cid}\in\mathcal{I}})$ for the corrupted party $\mathcal{P}_j$. Then set $s_{3-j,\mathsf{cid}_h} = y_h \oplus s_{j,\mathsf{cid}_h}$.
   (b) For $\mathsf{cid} \in \mathcal{I}$ store $(\mathsf{cid}, s_{1,\mathsf{cid}}, s_{2,\mathsf{cid}})$. Then add $(\Delta, \mathsf{mid}_1, \mathsf{sid}, \mathcal{P}_{3-j}, (\mathsf{OutputShares}, \{(\mathsf{cid}, s_{3-j,\mathsf{cid}})\}_{\mathsf{cid}\in\mathcal{I}}))$ for a fresh mid_1 to $\mathcal{Q}$ and send $(\mathsf{OutputShares}, \mathsf{sid}, \mathcal{P}_{3-j}, \mathsf{mid}_1)$ to $\mathcal{S}$.

Share Random Value: Upon input $(\mathsf{ShareRandom}, \mathsf{sid}, \mathcal{I})$ by both parties, for fresh identifiers $\mathcal{I}$ and letting $\mathcal{P}_j$ be the corrupted party:

1. Send $(\mathsf{RequestShares}, \mathsf{sid}, \mathcal{I})$ to $\mathcal{S}$, which replies with $(\mathsf{RandomShares}, \mathsf{sid}, \{(\mathsf{cid}, s_{j,\mathsf{cid}})\}_{\mathsf{cid}\in\mathcal{I}})$ for the corrupted party $\mathcal{P}_j$. Then sample $s_{3-j,\mathsf{cid}} \xleftarrow{\$} \mathbb{F}$ for each $\mathsf{cid} \in \mathcal{I}$.
2. For each $\mathsf{cid} \in \mathcal{I}$ store $(\mathsf{cid}, s_{1,\mathsf{cid}}, s_{2,\mathsf{cid}})$. Then add $(\Delta, \mathsf{mid}_1, \mathsf{sid}, \mathcal{P}_{3-j}, (\mathsf{RandomShares}, \{(\mathsf{cid}, s_{3-j,\mathsf{cid}})\}_{\mathsf{cid}\in\mathcal{I}}))$ for a fresh mid_1 to $\mathcal{Q}$ and send $(\mathsf{RandomShares}, \mathsf{sid}, \mathcal{P}_{3-j}, \mathsf{mid}_1)$ to $\mathcal{S}$.

Linear Combination: Upon input $(\mathsf{Linear}, \mathsf{sid}, \{(\mathsf{cid}, \alpha_{\mathsf{cid}})\}_{\mathsf{cid}\in\mathcal{I}}, \mathsf{cid}')$ from both parties: If all $\alpha_{\mathsf{cid}} \in \mathbb{F}$, all $\mathsf{cid} \in \mathcal{I}$ have stored values and cid' is unused, set $s_{i,\mathsf{cid}'} \leftarrow \sum_{\mathsf{cid}\in\mathcal{I}} \alpha_{\mathsf{cid}} \cdot s_{i,\mathsf{cid}}$ for $i \in \{1, 2\}$ and record $(\mathsf{cid}', s_{1,\mathsf{cid}'}, s_{2,\mathsf{cid}'})$.

Reveal: Upon input $(\mathsf{Reveal}, \mathsf{sid}, \mathsf{cid})$ by $\mathcal{P}_i \in \{\mathcal{P}_1, \mathcal{P}_2\}$, if (cid, s_1, s_2) is stored and $\mathcal{P}_j$ is corrupted:

1. Add $(\Delta, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, (\mathsf{Reveal}))$ to $\mathcal{Q}$ for an unused mid. Then send $(\mathsf{Reveal}, \mathsf{sid}, \mathcal{P}_i, \mathsf{mid})$ to $\mathcal{S}$.
2. If both parties sent $(\mathsf{Reveal}, \mathsf{sid}, \mathsf{cid})$ then send $(\mathsf{Reveal}, \mathsf{sid}, \mathsf{cid}, s_1 \oplus s_2)$ to $\mathcal{S}$.
3. If $\mathcal{S}$ sends $(\mathsf{DeliverReveal}, \mathsf{sid}, \mathsf{cid})$ then add $(\Delta, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_{3-j}, (\mathsf{Reveal}, \mathsf{cid}, s_1 \oplus s_2))$ for a fresh mid to $\mathcal{Q}$.
4. Send $(\mathsf{DeliverReveal}, \mathsf{sid}, \mathsf{cid}, \mathcal{P}_{3-j}, \mathsf{mid})$ to $\mathcal{S}$.

Fig. 9. Functionality $\mathcal{F}^{\Delta}_{\mathsf{2pcsso}}$ for 2PC with Secret-Shared Output and Linear Secret Share Operations, Part 2.

lows for delayed openings. The functionality is naturally ticked, as its implementation will use both $\mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$ and $\mathcal{F}_{\mathsf{tlp}}$ (see Appendix B). In addition to regular commit and opening procedures, the functionality has a special **Delayed Open** command which releases the message in a commitment after a delay δ. The adversary $\mathcal{A}$ is notified that an honest party has requested a delayed opening and may introduce a (communication) delay of maximum Δ ticks before the honest party receives the delayed opening (or it may decide to abort the opening process). However, $\mathcal{A}$ cannot choose to abort the delayed opening anymore once the process started. $\mathcal{A}$ will learn the opening after δ ticks while an honest receiver $\mathcal{P}_R$ might have to wait $\delta + \Delta$ ticks in total as the ticking for the delayed open-

Functionality $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$

The functionality is parameterized by $h, \Delta \in \mathbb{N}$ and interacts with two parties $\mathcal{P}_S, \mathcal{P}_R$ and an adversary $\mathcal{S}$ who may corrupt either of $\mathcal{P}_S, \mathcal{P}_R$. The functionality will internally have initially empty lists $\mathcal{M}, \mathcal{Q}$ and $\mathcal{O}$, a map $\mathsf{commits}$, and a flag $\mathtt{open} = 0$.

Commit: Upon receiving $(\mathsf{Commit}, \mathsf{sid}, \mathsf{cid}, \boldsymbol{x})$ from $\mathcal{P}_S$ where cid is an unused identifier and $\boldsymbol{x} \in \mathbb{F}^h$, ignore if $\mathtt{open} = 1$. Otherwise, proceed as follows:
1. Set $\mathsf{commits}[\mathsf{cid}] = \boldsymbol{x}$.
2. Add $(\Delta, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_R, (\mathsf{Commit}, \mathsf{cid}))$ to $\mathcal{Q}$ for an unused mid.
3. Send $(\mathsf{Commit}, \mathsf{sid}, \mathsf{mid}, \mathsf{cid})$ to $\mathcal{S}$. If $\mathcal{S}$ answers with $(\mathsf{sid}, \mathsf{abort})$, the functionality halts.

Open: Upon receiving $(\mathsf{Open}, \mathsf{sid}, \mathsf{cid}_1, \ldots, \mathsf{cid}_o)$ from $\mathcal{P}_S$, ignore if $\mathtt{open} = 1$. Otherwise, if $\mathsf{commits}[\mathsf{cid}_i] = \boldsymbol{x}_i \neq \perp$ for $i \in [o]$, proceed as follows:
1. Add $(\Delta, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_R, (\mathsf{Open}, (\mathsf{cid}_i, \boldsymbol{x}_i)_{i\in[o]})$ to $\mathcal{Q}$ for a fresh mid.
2. Send $(\mathsf{Open}, \mathsf{sid}, \mathsf{mid}, (\mathsf{Open}, (\mathsf{cid}_i, \boldsymbol{x}_i)_{i\in[o]})$ to $\mathcal{S}$ and set $\mathtt{open} = 1$.

Delayed Open: Upon receiving $(\mathsf{DOpen}, \mathsf{sid}, \mathsf{cid}_1, \ldots, \mathsf{cid}_o, \delta)$ from $\mathcal{P}_S$, ignore if $\mathtt{open} = 1$. Otherwise, if $\mathsf{commits}[\mathsf{cid}_i] = \boldsymbol{x}_i \neq \perp$ for $i \in [o]$, proceed as follows:
1. Add $(\Delta, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_R, (\mathsf{DOpen}, \mathsf{cid}_1, \ldots, \mathsf{cid}_o))$ to $\mathcal{Q}$ for a fresh mid.
2. Add $(\delta, \mathcal{P}_R, (\mathsf{cid}_i, \boldsymbol{x}_i)_{i\in[o]})$ and $(\delta, \mathcal{S}, (\mathsf{cid}_i, \boldsymbol{x}_i)_{i\in[o]})$ to $\mathcal{O}$.
3. Send $(\mathsf{DOpen}, \mathsf{sid}, \mathsf{cid}, \ldots, \mathsf{cid}_o, \delta)$ to $\mathcal{S}$ and set $\mathtt{open} = 1$.

Linear Combination: Upon receiving $(\mathsf{Linear}, \mathsf{sid}, \{(\mathsf{cid}, \alpha_{\mathsf{cid}})\}_{\mathsf{cid}\in\mathcal{I}}, \boldsymbol{\beta}, \mathsf{cid}')$ where all $\alpha_{\mathsf{cid}} \in \mathbb{F}$ and $\boldsymbol{\beta} \in \mathbb{F}^h$ from $\mathcal{P}_S$, ignore if $\mathtt{open} = 1$. Otherwise, if $\mathsf{commits}[\mathsf{cid}] = \boldsymbol{x}_{\mathsf{cid}} \neq \perp$ for all $\mathsf{cid} \in \mathcal{I}$ and cid' is unused, set $\mathsf{commits}[\mathsf{cid}'] = \boldsymbol{\beta} + \sum_{\mathsf{cid}\in\mathcal{I}} \alpha_{\mathsf{cid}} \cdot \boldsymbol{x}_{\mathsf{cid}}$.

Fetch Message: Upon receiving $(\mathsf{FetchMsg}, \mathsf{sid})$ by $\mathcal{P} \in \{\mathcal{P}_S, \mathcal{P}_R\}$ retrieve the set L of all entries $(\mathcal{P}, \mathsf{sid}, \cdot)$ in $\mathcal{M}$, remove L from $\mathcal{M}$ and return $(\mathsf{Output}, \mathsf{sid}, L)$ to $\mathcal{P}$.

Scheduling: On input of $\mathcal{S}$:
If $\mathcal{S}$ sent $(\mathsf{Deliver}, \mathsf{sid}, \mathsf{mid})$ then remove each $(\mathtt{cnt}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}, m)$ from $\mathcal{Q}$ and add $(\mathcal{P}, \mathsf{sid}, m)$ to $\mathcal{M}$.

Tick: Upon receiving $(\mathsf{Tick}, \mathsf{sid})$ from $\mathcal{S}$:
1. For each query $(0, \mathsf{mid}, \mathsf{sid}, \mathcal{P}, m) \in \mathcal{Q}$:
   (a) Remove $(0, \mathsf{mid}, \mathsf{sid}, \mathcal{P}, m)$ from $\mathcal{Q}$.
   (b) Add $(\mathcal{P}, \mathsf{sid}, m)$ to $\mathcal{M}$.
2. Replace each $(\mathtt{cnt}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}, m)$ in $\mathcal{Q}$ with $(\mathtt{cnt} - 1, \mathsf{mid}, \mathsf{sid}, \mathcal{P}, m)$.
3. For each entry $(\delta, \mathcal{S}, \mathsf{cid}, \boldsymbol{x}) \in \mathcal{O}$, proceed as follows:
   – If $\delta = 0$, output $(\mathsf{DOpen}, \mathsf{sid}, \mathcal{S}, (\mathsf{cid}_i, \boldsymbol{x}_i)_{i\in[o]})$ to $\mathcal{S}$ and append $(\mathcal{P}_S, \mathsf{sid}, (\mathsf{DOpened}, \mathsf{cid}_1, \ldots, \mathsf{cid}_o))$ to $\mathcal{M}$.
   – If $\delta > 0$, replace $(\delta, \mathcal{S}, (\mathsf{cid}_i, \boldsymbol{x}_i)_{i\in[o]})$ with $(\delta - 1, \mathcal{S}, (\mathsf{cid}_i, \boldsymbol{x}_i)_{i\in[o]})$ in $\mathcal{O}$.
4. For each entry $(\delta, \mathcal{P}_R, (\mathsf{cid}_i, \boldsymbol{x}_i)_{i\in[o]}) \in \mathcal{O}$, if there is no entry $(\mathtt{cnt}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_R, (\mathsf{DOpen}, \mathsf{cid}_1, \ldots, \mathsf{cid}_o)) \in \mathcal{Q}$, proceed as follows:
   – If $\delta = 0$, output $(\mathsf{DOpen}, \mathsf{sid}, \mathcal{P}_R, (\mathsf{cid}_i, \boldsymbol{x}_i)_{i\in[o]})$ to $\mathcal{P}_R$.
   – If $\delta > 0$, replace $(\delta, \mathcal{P}_R, (\mathsf{cid}_i, \boldsymbol{x}_i)_{i\in[o]})$ with $(\delta - 1, \mathcal{P}_R, (\mathsf{cid}_i, \boldsymbol{x}_i)_{i\in[o]})$ in $\mathcal{O}$.

Fig. 10. Functionality $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ For Homomorphic Commitments with Delayed Opening.

ing of a commitment naturally can only happen once notification arrives on the receiver's side.

Coin Tossing. In our protocol we additionally need to use a functionality for coin tossing, as mentioned before. It could actually already be implemented, albeit inefficiently, using $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$. For completeness, we instead use the functionality $\mathcal{F}_{\mathsf{ct}}$ which can be found in Appendix A.

6.2 Achieving Output-Independent Abort for 2PC in UC

Intuitively, the protocol realizing $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ works as follows: first, both parties use $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ to perform the secure computation. They then don't directly obtain an output, but instead each get a vector of shares $\boldsymbol{s}_i$. Afterwards, the parties will commit to $\boldsymbol{s}_i$ and use the homomorphic property of $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ to show consistency between the values in both functionalities for which they use $\mathcal{F}_{\mathsf{ct}}$. At this stage the protocol might still fail and an adversary might still abort, but no information will leak. Finally, both parties use the **Delayed Open** to reveal their share $\boldsymbol{s}_i$ which allows each party to reconstruct the output. At this stage, $\mathcal{A}$ might decide not to activate **Delayed Open**, but we can set the parameters of $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ such that it will have to do so before the commitment of the honest party opens. If it does not do so, then the honest party will decide that an abort happened and just ignore any future messages of $\mathcal{A}$.

The full protocol π_{2pcoia} can be found in Fig. 11. We can then show the following statement:

Theorem 3. *Let $\delta > \Delta$ and $\kappa \in \mathbb{N}^+$ be a statistical security parameter. Then the protocol π_{2pcoia} implements $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ in the $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}, \mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}, \mathcal{F}_{\mathsf{ct}}$-hybrid model against any static active adversary corrupting at most one of the two parties that follows our "ticking" requirement.*

In the proof, we will make crucial use of the following standard fact:

Lemma 1. *Fix values $\boldsymbol{s}, \boldsymbol{r}_1, \ldots, \boldsymbol{r}_m, \boldsymbol{s}', \boldsymbol{r}'_1, \ldots, \boldsymbol{r}'_m \in \mathbb{F}^h$. Then pick $\alpha_k \xleftarrow{\$} \mathbb{F}$ for $k \in [\kappa]$ uniformly at random. If for all $k \in [\kappa]$*

$$\boldsymbol{r}_k \oplus \alpha_k \cdot \boldsymbol{s}_k = \boldsymbol{r}'_k \oplus \alpha_k \cdot \boldsymbol{s}'_k$$

then $\boldsymbol{s} = \boldsymbol{s}'$ and $\boldsymbol{r}_k = \boldsymbol{r}'_k$ for all $k \in [\kappa]$ except with probability $O(2^{-\kappa})$.

Proof. To simplify notation, assume that $\mathcal{A}$ will corrupt $\mathcal{P}_2$. In the proof, we will construct a simulator $\mathcal{S}$ which will run a "fake" instance of the protocol π_{2pcoia} with the adversary $\mathcal{A}$ while actually interacting with the ideal functionality $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$. It therefore simulates a party $\mathcal{P}_1$ as well as functionalities $\mathcal{F}_{\mathsf{ct}}, \mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}, \mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$. During the simulation, whenever $\mathcal{A}$ will tick the hybrid functionalities then $\mathcal{S}$ will tick $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ after $\mathcal{A}$ has finished the ticking. Whenever $\mathcal{S}$ calls **Scheduling** of $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ then it will only make the query just before executing **Tick** on $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$.

Protocol π_{2pcoia}

This protocol is for two parties $\mathcal{P}_1, \mathcal{P}_2$ and uses the functionalities $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$, $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ and $\mathcal{F}_{\mathsf{ct}}$. The parties want to compute the circuit C over $\mathbb{F}$ with output length m. We assume for simplicity that the commitment functionality $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ commits to vectors of length m.

Init: Each $\mathcal{P}_i$ sends $(\mathsf{Init}, \mathsf{sid}, C)$ to $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ and then queries $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ for an output $(\mathsf{Init}, \mathsf{sid}, C)$. If $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ instead aborts, then output $(\mathsf{Abort}, \mathsf{sid})$ and stop.

Input: Each $\mathcal{P}_i$ sends $(\mathsf{Input}, \mathsf{sid}, i, x_i)$ to $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ and then queries $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ for an output $(\mathsf{Input}, \mathsf{sid}, \mathcal{P}_{3-i})$. If $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ instead aborts, then output $(\mathsf{Abort}, \mathsf{sid})$ and stop.

Computation: Each $\mathcal{P}_i$ sends $(\mathsf{Compute}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ and then queries $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ for an output $(\mathsf{Compute}, \mathsf{sid})$. If $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ instead aborts, then output $(\mathsf{Abort}, \mathsf{sid})$ and stop.

Output:

1. Each party $\mathcal{P}_i$ sends $(\mathsf{ShareOutput}, \mathsf{sid}, \mathsf{cid}_1, \ldots, \mathsf{cid}_m)$ for fixed cid_h to $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ to receive its shares $s_{1,i}, \ldots, s_{m,i}$.
2. Each party $\mathcal{P}_i$ sends $(\mathsf{RandomOutput}, \mathsf{sid}, \widehat{\mathsf{cid}}_1, \ldots, \widehat{\mathsf{cid}}_{m\cdot\kappa})$ for fixed $\widehat{\mathsf{cid}}_t$ to $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ to receive its shares $r_{1,i}, \ldots, r_{m\cdot\kappa,i}$.
3. Each party uses $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ to commit to $\boldsymbol{s}_i = (s_{1,i}, \ldots, s_{m,i})$ as well as $\boldsymbol{r}_{k,i} = (r_{(k-1)\cdot m+1,i}, \ldots, r_{k\cdot m,i})$ for $k \in [\kappa]$ using the cid's $\mathsf{cid}_i^s, \mathsf{cid}_{1,i}^r, \ldots, \mathsf{cid}_{\kappa,i}^r$.
4. Each $\mathcal{P}_i$ sends $(\mathsf{Toss}, \mathsf{sid}, \kappa)$ to $\mathcal{F}_{\mathsf{ct}}$ and obtains $\alpha_1, \ldots, \alpha_\kappa$.
5. For $i \in [2], k \in [\kappa]$ the parties use **Linear Combination** on $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ to compute commitments for the κ values $d_{k,i} = \alpha_k \cdot \boldsymbol{s}_i \oplus \boldsymbol{r}_{k,i}$. These have cid's $\mathsf{cid}_{1,i}^d, \ldots, \mathsf{cid}_{\kappa,i}^d$.
6. For $k \in [\kappa], h \in [m]$ the parties use **Linear Combination** on $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ to compute the linear relations $d_{k,h} = \alpha_k \cdot s_h \oplus r_{(k-1)\cdot m+h}$.
7. The parties use **Reveal** to open $d_{k,h}$ for all $k \in [\kappa], h \in [m]$.
8. Each party $\mathcal{P}_i$ sends $(\mathsf{DOpen}, \mathsf{sid}, \mathsf{cid}_i^s, \mathsf{cid}_{1,i}^d, \ldots, \mathsf{cid}_{\kappa,i}^d, \delta)$ to its instance of $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$.
9. In a loop, each party $\mathcal{P}_i$:
   (a) Queries the instance of $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ where $\mathcal{P}_i$ was a receiver to see if it obtained a message $(\mathsf{DOpen}, \mathsf{cid}_{3-i}^s, \mathsf{cid}_{1,3-i}^d, \ldots, \mathsf{cid}_{\kappa,3-i}^d)$. If so, then exit the loop.
   (b) Queries the instance of $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ where $\mathcal{P}_i$ was a sender to see if it obtained a message $(\mathsf{DOpened}, \mathsf{cid}_i^s, \mathsf{cid}_{1,i}^d, \ldots, \mathsf{cid}_{\kappa,i}^d)$. If so, then exit the loop.
   (c) Returns activation.
10. After having obtained either of the above messages, $\mathcal{P}_i$ does the following:
    – If $\mathsf{DOpened}$ arrived before DOpen then output $\perp$.
    – If DOpen arrived before $\mathsf{DOpened}$ then query $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ until $\tilde{\boldsymbol{s}}_{3-1}, \tilde{\boldsymbol{d}}_{1,3-i}, \ldots, \tilde{\boldsymbol{d}}_{\kappa,3-i}$ were obtained. Then output $\boldsymbol{y} = \boldsymbol{s}_i \oplus \tilde{\boldsymbol{s}}_{3-i}$ if $\tilde{\boldsymbol{d}}_{k,3-i}[h] = d_{k,h} \oplus \boldsymbol{d}_{k,i}[h]$ for all $k \in [\kappa], h \in [m]$ and $\perp$ otherwise.

Fig. 11. Protocol π_{2pcoia} For 2PC with Output-Independent Abort.

$\mathcal{S}$ will let $\mathcal{P}_1$ follow the protocol in steps **Init**, **Input** and **Computation**, where it will use a random input x_1 for $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$, additionally extract the input

x_2 which $\mathcal{A}$ inputs into $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ and forward it to $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$. The exact delay of the $\mathsf{Init}, \mathsf{Input}, \mathsf{Computation}$ messages can directly be forwarded by $\mathcal{S}$ to $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ - whenever $\mathcal{A}$ inputs a message of $\mathcal{P}_2$ into these interfaces or accelerates message delivery, then $\mathcal{S}$ can do the respective operation in $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ using **Scheduling**. During **Output** $\mathcal{S}$ runs the protocol until Step 8 begins. If in Step 3 $\mathcal{A}$ will input values $\boldsymbol{s}_2, \boldsymbol{r}_{k,2}$ into $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ that it did not obtain from $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ then set $\texttt{cheated} \leftarrow \top$.

In Step 8 $\mathcal{S}$ will follow the protocol as before, but will send $(\mathsf{Output}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ and for the next Δ ticks it will wait if it obtains $(\mathsf{DOpen}, \mathsf{sid}, \mathcal{P}_1, (\mathsf{cid}_i, \cdot)_{i\in[o]})$ from $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ where $\mathcal{P}_1$ is receiver. If during this period it will either not obtain DOpen or $\mathsf{cheated} = \top$ then $\mathcal{S}$ will send $(\mathsf{DeliverOrAbort}, \mathsf{sid}, \mathsf{mid}, \bot)$ to $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$, otherwise it will send $(\mathsf{DeliverOrAbort}, \mathsf{sid}, \mathsf{mid}, \top)$ for the output message-id mid of $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$. One of these two messages will be sent until Δ ticks have passed. Then, $\mathcal{S}$ obtains the output $\boldsymbol{y}$ from $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$. In the "tick-round" when $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ outputs a value to the simulated $\mathcal{P}_1$ send $(\mathsf{Deliver}, \mathsf{sid}, \mathsf{mid})$ to $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$. In the round when $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ outputs the committed messages to the adversary replace the value for cid_1^s with $\boldsymbol{y} \oplus \boldsymbol{s}_2$ where the latter was extracted from the $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ instance where $\mathcal{P}_2$ was sender.

Indistinguishability. We first observe that $\mathcal{S}$ is poly-time and also ticks $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ whenever $\mathcal{A}$ ticks the hybrid functionalities. Therefore, $\mathcal{S}$ preserves the required ticking property.

The output which $\mathcal{A}$ obtains is exactly the output of $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ which follows by linearity on how the output $\boldsymbol{s}_1$ of $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ is constructed. $\mathcal{A}$ obtains the output in the same round as in the real protocol, and $\mathcal{P}_1$ obtains the output from $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ in the same round as in a real protocol. The $\mathsf{DeliverOrAbort}$ message is sent by $\mathcal{S}$ after Δ ticks, and since $\delta > \Delta$ it will only have to reprogram the output of $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ after actually having obtained it from $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$. The other messages which $\mathcal{A}$ obtains from $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ and $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ have the same distribution as in the protocol as they are generated the same way and due to the uniform choice of the $\boldsymbol{r}_{k,1}$.

The only difference in the output distribution towards $\mathcal{A}$ is the influence of the flag $\texttt{cheated}$. In the simulation we will make $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ output Abort whenever the values that go into $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ are not the same as the ones $\mathcal{A}$ received from $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$. In the real protocol, we instead abort if the conditions in Step 10 of π_{2pcoia} do not hold. The difference is exactly captured by Lemma 1 as being negligible in the statistical security parameter κ and the claim follows. □

6.3 Extensions

It is possible to extend the protocol π_{2pcoia} to have timeouts, support more than two parties as well as achieve identifiable abort and public verifiability. We will now sketch the approaches but leave a detailed analysis for future work.

Timeouts. $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ does not make any abort guarantees before the output phase, meaning that if an adversary never sents an **Init** message to it then the honest party might wait for a response forever. This is due to the fact that the parties themselves are unaware of the actual delays involved in $\mathcal{F}_{\mathsf{smt,delay}}^{\Delta}$ and $\mathcal{F}_{\mathsf{tlp}}$. We can avoid this by introducing local clocks to the implementing protocols which tick correlated with the "ticks" issued to the functionalities by $\mathcal{A}$. Using such local clocks we can then achieve a protocol that always terminates.

Realizating Output-Independent Identifiable Abort in the Multiparty Setting. As the $\mathcal{F}_{\mathsf{tlp}}$ functionality already can handle multiple parties, this functionality can be instantiated the same way. In order to generalize $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$, $\mathcal{F}_{\mathsf{ct}}$ and $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ we would then additionally have to assume the existence of a broadcast channel and explicitly model such a channel in our framework. The modifications to a multiparty-setting of $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ and $\mathcal{F}_{\mathsf{ct}}$ are folklore, whereas the protocol that we use to instantiate $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ can be generalized to the multiparty-setting as well.

In the presence of multiple parties we can furthermore modify the protocol so all honest parties agree on a set of cheaters. Using multi-receiver commitments this property can be proven to hold for the output phase of a multiparty version of our protocol, since the output reconstruction will only consist of an opening phase for a multi-receiver commitment with delayed opening. We leave a formal analysis of this as future work.

Public Verifiability. While a protocol with identifiable abort allows the participants of it to agree on cheaters, a protocol with publicly verifiable abort additionally generates a publicly available certificate that can be used by an external party to detect the same set of cheaters. This approach has been used in previous work to incentivize fairness by financially punishing adversarial behavior. Our protocol can be made compatible with the approach of [6] in the multiparty setting by using their publicly verifiable commitments and a publicly verifiable version of $\mathcal{F}_{\mathsf{tlp}}$. Notice that π_{tlp} already supports public verifiability, since the trapdoor for $\mathcal{F}_{\mathsf{rsw}}$ is obtained by each party who solves the puzzle, allowing them to publicly prove that a certain message was obtained by simply handing the puzzle along with the trapdoor to a third party verifier. We leave a detailed construction and formal analysis as future work.

7 The Impossibility Result

We show that in the UC model one cannot implement fair coin-flip without using a random oracle, or similar programmable setup assumption. This holds even if one is allowed to use time-lock puzzles, and non-programmable random oracles and 2PC with abort. We first show the impossibility result for the *simple case* where we assume there is no setup, no random oracles and that the protocol has a fixed round complexity. This allows us to focus on the central new idea. After that we show the result for the full case.

The ideal functionality $\mathcal{F}_{\mathsf{cf}}$ for fair coin-flip (without abort) proceeds as follows. When activated by any party in round 0 it will sample a uniformly random bit c and output it to both parties in some round ρ specified by the adversary. The adversary cannot refuse the output to be given. The ideal functionality is rushing: the adversary gets c in round 0. The honest parties do not get the coin until round ρ.

Implications. Below we show that in several settings, called the excluded settings, one cannot UC securely realize $\mathcal{F}_{\mathsf{cf}}$. By the UC composition theorem this impossibility result has wide implications. In particular, it holds for all ideal functionalities $\mathcal{G}$ that if one can UC securely realize $\mathcal{F}_{\mathsf{cf}}$ in the $\mathcal{G}$-hybrid model, then one cannot UC realize $\mathcal{G}$ in the excluded settings either.

Impossibility of Two-Party Coinflip with Output-Independent Abort It follows that two-party coin-flip with output-independent abort is impossible in the excluded settings. Namely, given a protocol π_{cfoia} for two-party coin-flip with output-independent abort one can get a two-party coin-flip protocol π_{cf} without abort as follows. We describe the protocol in the $\mathcal{F}_{\mathsf{cfoia}}$-hybrid model and get the result by composition. Run $\mathcal{F}_{\mathsf{cfoia}}$. If neither of the parties aborts, take the output of $\mathcal{F}_{\mathsf{cfoia}}$ to be the output. If one of the parties aborts, let the other party sample and announce a uniformly random c and take c as the output. To simulate the protocol, get from $\mathcal{F}_{\mathsf{cf}}$ the coin c to hit in the simulation. Simulate a copy of $\mathcal{F}_{\mathsf{cfoia}}$ to the adversary. If the adversary does not abort, let the output of $\mathcal{F}_{\mathsf{cfoia}}$ be c. Otherwise, let the output of $\mathcal{F}_{\mathsf{cfoia}}$ be a uniformly random c', and then simulate that the honest party samples and announces c.

Notice that it was crucial for this simulation that we could change the output of $\mathcal{F}_{\mathsf{cfoia}}$ from c to an independent c' when there was an abort. Namely, when there is an abort we still need to hit the c output by $\mathcal{F}_{\mathsf{cf}}$ in the simulation, so we are forced to simulate that the honest party samples and announces c in the simulation. But if we were then also forced to let $\mathcal{F}_{\mathsf{cfoia}}$ output c, then in the simulation the bits output by $\mathcal{F}_{\mathsf{cfoia}}$ and the honest party when there is an abort will always be the same. In the protocol they are independent. This would make it easy to distinguish. A generalisation of this observation will later be the basis for our impossibility result.

Impossibility of UC 2PC with Output-Independent Abort It also follows that 2PC with output-independent abort is impossible in the excluded settings. Namely, given a functionality $\mathcal{F}_{\mathsf{2pcoia}}$ for 2PC with output-independent abort (as described in the previous section) one can UC securely realize $\mathcal{F}_{\mathsf{cfoia}}$. Namely, use $\mathcal{F}_{\mathsf{2pcoia}}$ to compute the function which takes one bit as input from each party and outputs the exclusive or. Let each party input a uniformly random bit. If any party aborts on $\mathcal{F}_{\mathsf{2pcoia}}$, abort in π_{cfoia}. It is straight forward to simulate π_{cfoia} given $\mathcal{F}_{\mathsf{2pcoia}}$.

Impossibility of UC Time-lock Puzzles It also follows that UC time-lock puzzles are impossible in the excluded settings. Namely, we have shown that given UC time-lock puzzles one can UC securely realize $\mathcal{F}_{\mathsf{2pcoia}}$, which was excluded above.

7.1 Technical Details of the Simple Setting

We now show that one cannot UC securely implement $\mathcal{F}_{\mathsf{cf}}$ using a synchronous protocol using only point-to-point communication even if one is allowed time-lock puzzles.

We will assume without loss of generality that π_{cf} is of a particular form described now. Consider a synchronous protocol for two parties Alice and Bob. They have no inputs. They begin the protocol in the same round, call it round 1. In round 1 Alice first sends m^1 to Bob. In round 2 Bob sends m^2 to Alice, and so on. We assume a rushing adversary, so we can without loss of generality assume the parties take turns sending messages.

To simplify our treatment of abort, we assume that the parties abort by sending a special symbol abort. We also imagine that in all rounds after the protocol ended a party sends abort, whereas in practice it would terminate a send nothing. This is just to make notation easier. We make the convention that if a party $\mathcal{P}$ sends abort in a given round i the other party $\mathcal{P}'$ proceeds as if $\mathcal{P}$ sends abort in all rounds $\geq i$ and party $\mathcal{P}'$ will itself send abort in all following rounds. Therefore, once a party sent abort the other parties will de facto ignore at its future messages. In particular, it follows that a party can eventually compute its output bit $c_{\mathcal{P}}$ the first time it receives abort from the other party without considering any future message from the other party. This is purely a notational convenience for talking about aborts. We have to some extend just made abort a legal message. This is possible as when implementing $\mathcal{F}_{\mathsf{cf}}$ the parties must always output a bit c even when the other party aborts. The only way to "abort" is hence to send the "end of protocol" signal abort prematurely. For a given execution, we use ρ to denote the first round in which a parties sent abort.

Checkpoints. We define a number of so-called checkpoints in the execution. These are just internal states of parties at particular points in the execution. We let σ^0 be the initial state of Bob. For round $0 < i \leq \rho$ we let $\mathcal{P}$ be the party computing m^i and we let σ^i be the state of $\mathcal{P}$ right after computing m^i and right before sending m^i.

Default Values. For each round $i \geq 0$ we define a bit c^i called the default value of round i. This is the value that the party $\mathcal{P}$ that sent m^i will output as result of the coin-flip if the other party replies to m^i by aborting. Due to the use of for instance time-lock puzzles $\mathcal{P}$ might not be able to compute c^i during round i, but $\mathcal{P}$ will eventually be able to compute c^i as it is its output if the other party $\mathcal{P}'$ sends abort in all following rounds. Therefore $\mathcal{P}$ can just start from its state

after computing m^i and simulate that $\mathcal{P}'$ sends abort in all subsequent rounds, and run until $\mathcal{P}$ gives an output. This will by definition be c^i.

- We let c^0 be the output c that Bob would output if Alice sends abort in the first round. In more details, let σ^0 be the initial state of Bob. Continue running from σ^0 and simulate that Alice sends abort in all subsequent rounds. When Bob gives an output c_{Bob}, let $c^0 = c_{\mathsf{Bob}}$.
- For each round $0 < i \leq \rho$ we let c^i be the output bit the sending party $\mathcal{P}$ would give if after sending message m^i to the other party that other party replies with abort. Let σ^i be the state of $\mathcal{P}$ right after computing m^i. Continue running $\mathcal{P}$ from σ^i and simulate that the other party $\mathcal{P}'$ sends abort in all subsequent rounds. When $\mathcal{P}$ gives an output $c_{\mathcal{P}}$, let $c^i = c_{\mathcal{P}}$.
- For each round $i > \rho$ we let $c^i = c^{i-2}$.

We record a property of the default value which is crucial to the proof.

Eventual Revelation: After a party $\mathcal{P}$ computed m^i it can eventually compute c^i. This holds no matter whether m^i is sent or withheld or in general how the future executing proceeds. This is because c^i can be computed solely from the checkpoint σ^i, the state of $\mathcal{P}$ after computing m^i. This property ensures that the default value c^i will eventually be revealed to the party $\mathcal{P}$ computing m^i. Note that this would not be true in the presence of an arbitrary ideal functionality used by Alice and Bob for communication, as the sending of m^i might change the behaviour of the ideal functionality, preventing $\mathcal{P}$ from later computing c^i. However, in our simple setting the property clearly holds.

Let $\mathcal{Z}$ be an environment which we will specify below. Let $\text{EXEC}_{\pi_{\mathsf{cf}},\mathcal{Z}}$ be the execution in the real world model with a dummy adversary. Let $\mathcal{S}$ be any potential simulator and let $\text{EXEC}_{\mathcal{F}_{\mathsf{cf}},\mathcal{S},\mathcal{Z}}$ be the simulation. We will assume that it holds for all $\mathcal{Z}$ that $\text{EXEC}_{\mathcal{F}_{\mathsf{cf}},\mathcal{S},\mathcal{Z}} \approx \text{EXEC}_{\pi_{\mathsf{cf}},\mathcal{Z}}$, and reach a contradiction.

For any execution of π_{cf} define the default values c^i as above. Let c be the output of an honest party in π_{cf}. We can assume without loss of generality that if there are two honest parties, then they output the same c (except with negligible probability). If not this would allow a trivial distinguishing attack. Let $e^i = \Pr[c^i = c]$ and let $d^i = \Pr[c^i = c^{i-1}]$. The value e^i is the probability that the default value that $\mathcal{P}$ was "planning" to output after sending m^i in case the other party would abort equals the value it ended up outputting in round ρ. The value d^i is the probability that the default value that $\mathcal{P}$ was planning to output after sending m^i in case the other party would abort equals the value that the other party defaulted to in the round before. The key to our impossibility result is that sometimes the values e^i and d^i are different in the real world execution, whereas in the simulation they are the same as c^{i-1} must be the output of $\mathcal{F}_{\mathsf{cf}}$.

Lemma 2. *In a random honest run in the real world it holds that* $e^0 \approx \frac{1}{2}$.

Proof. If $\Pr[c^0 = c] > \frac{1}{2}$, then $\Pr[c = 0|c^0 = 0] > \frac{1}{2}$ or $\Pr[c = 1|c^0 = 1] > \frac{1}{2}$. Assume without loss of generality that it is $\Pr[c = 0|c^0 = 0] > \frac{1}{2}$. Recall that

c^0 is the output c that Bob would output if Alice sends abort in the first round. This bits depends only on the random tape of Bob. So, if $\Pr[c = 0|c^0 = 0] > \frac{1}{2}$, then Bob can use a hard coded random tape where $c^0 = 0$ and hence ensure that $\Pr[c = 0] > \frac{1}{2}$ in an honest run with that random tape. This biases the coin, which in particular allows to distinguish the real world execution from the ideal one, where the coin is unbiased. □

Lemma 3. *In a random honest run in the real world it holds that $e^\rho = 1$.*

Proof. The party $\mathcal{P}$ sending m^ρ gives output after sending it without considering any new inputs from the other party, by definition of ρ. Hence the output of that party is independent of what the other party sends in round $\rho+1$. Therefore by definition $c^\rho = c_{\mathcal{P}}$, which implies that $e^\rho = 1$. □

The following Lemma also follows from [20].

Lemma 4. *There exists a round $i > 0$ such that it is not the case that $e^i \approx d^i$.*

Proof. It follows from $e^\rho = 1$ and $e^0 = \frac{1}{2}$ that we can find a round $i > 0$ such that $e^i \approx 1$ and $e^{i-1} \not\approx 1$. At this point $d^i = \Pr[c^{i-1} = c^i] \approx \Pr[c^{i-1} = c] = e^{i-1} \not\approx 1$. □

Recall that in the impossibility result of Cleve cited above the basis for the proof is that when $e^i \neq d^i$ then the party $\mathcal{P}$ to send m^i can bias the coin. If it withholds the message m^i the output is c^{i-1}. If it sends it the output will be the output of an honest run. By $e^i \neq d^i$ these distributions are different. We cannot use this attack as we do not necessarily know c^i when we are to withhold m^i, for instance due to the use of time-lock puzzles. It is crucial in the proof of Cleve that c^i is known when m^i is about to be sent. This is why it is impossible to circumvent Cleve's result using time-lock puzzles in the stand alone model. However, maybe surprisingly, when security is defined via simulation we can reestablish impossibility using a variant of Cleve's techniques. Instead of mounting Cleve's witholding attack, the environment will after the execution of the protocol test whether the witholding attack would have biased the count. In the real world it sometimes will. In the ideal world it cannot. This will allow to distinguish.

Consider the following strategy $\mathcal{Z}_0$ of the environment when the $\mathcal{P}$ to send m^i in round i is corrupted. Run a copy $\mathcal{P}$ of the honest $\mathcal{P}$ and instruct the corrupted dummy adversary $\widehat{\mathcal{P}}$ to send the same messages as $\mathcal{P}$ in all rounds. Record the checkpoint σ^i, call it σ_0^i. At the end of the protocol compute the default c^i of $\mathcal{P}$ in round i and call it c_0^i. This is possible by *eventual revelation*. Let $c_{\mathcal{P}'}$ be the output of the honest party $\mathcal{P}'$. In the UC model the environment sees this value. Output $(c_0^i, c_0 = c_{\mathcal{P}'})$. By definition we have that $\Pr[c_0^i = c_{\mathcal{P}'}] \approx e^i$ in the real world. Therefore this also holds for the output (c_0^i, c_0) in the ideal world or we would have an attack. Note that in the simulation $c_{\mathcal{P}'} = c$, where c is the coin output by $\mathcal{F}_{\mathsf{cf}}$. It follows that in the simulation it holds that

$$\Pr[c_0^i = c] \approx e^i \ .$$

Note that this follows purely from eventual revelation.

Consider now the following strategy $\mathcal{Z}_1$ of the environment when the $\mathcal{P}$ to send m^i in round i is corrupted. Run a copy $\mathcal{P}$ of the honest $\mathcal{P}$ and instruct the corrupted dummy adversary $\widehat{\mathcal{P}}$ to send the same messages as $\mathcal{P}$ in all rounds $j < i$. Compute m^i and record the checkpoint σ^i, call it σ_1^i. Then abort in round i, i.e., do not send m^i. At the end of the protocol compute the default value c^i of $\mathcal{P}$ in round i and call it c_1^i. Let $c_{\mathcal{P}'}$ be the output of the honest party $\mathcal{P}'$. Output $(c_1^i, c_1 = c_{\mathcal{P}'})$. Note that in the real world $c_{\mathcal{P}'} = c^{i-1}$ (by definition). Therefore in the real world the distribution of the output (c_1^i, c_1) is that of (c^i, c^{i-1}). Therefore this also holds for the output (c_1^i, c_1) in the ideal world or we would have an attack. Note that in the simulation $c_{\mathcal{P}'} = c$, where c is the coin output by $\mathcal{F}_{\mathsf{cf}}$. It follows that in the simulation (c_1^i, c) is distributed as (c^i, c^{i-1}) in the real world. In particular $\Pr[c_1^i = c] \approx \Pr[c_1^i = c^{i-1}]$. By definition this gives us that

$$\Pr[c_1^i = c] \approx d^i \ .$$

The above was the crucial point where we use simulatability. Also without simulatability would We find a way to output (c^i, c^{i-1}) in the real world. We then go to the simulation and force $c^{i-1} = c$. This gives us a second correlation between c^i and c.

Consider now the following strategy $\mathcal{Z}$ of the environment when the $\mathcal{P}$ to send m^i in round i is corrupted. Run a copy $\mathcal{P}$ of the honest $\mathcal{P}$ and instruct the corrupted dummy adversary $\widehat{\mathcal{P}}$ to send the same messages as $\mathcal{P}$ in all rounds $j < i$. Compute m^i and record the checkpoint σ^i, call it σ^i. Pause. We record a property of the default value which is crucial to the proof.

Simulator Non-Interference: After an emulated party $\mathcal{P}$ in $\mathcal{Z}$ computed m^i it can by eventual revelation compute c^i from σ^i, we write $c^i = c^i(\sigma^i)$. Furthermore, after m^i was computed there is no way for the simulator to affect the computation $c^i = c^i(\sigma^i)$ anymore. Therefore the probability that $c^i = c$ depends only on the distribution of (σ^i, c). We call this *simulator non-interference*. Note that this would not be that case in the programmable random oracle model where the computation of $c^i = c^i(\sigma^i)$ might involve querying the oracle. But in our simple setting we clearly have simulator non-interference.

Notice now that up until and including the computation of m^i the three environments $\mathcal{Z}_0$, $\mathcal{Z}_1$, and $\mathcal{Z}$ have the exact same behaviour. In fact, if we stripped the continuation after round i, they would be *the same* environment, namely $\mathcal{Z}$. In the UC model the simulator therefore cannot distinguish $\mathcal{Z}_0$, $\mathcal{Z}_1$ and $\mathcal{Z}$ until m^i has been computed, as it only has blackbox access to $\mathcal{Z}$. From this it follows that (σ_0^i, c), (σ_1^i, c), and (σ^i, c) have identical distributions, where c is the coin in $\mathcal{F}_{\mathsf{cf}}$. In particular, if we let $c^i = c^i(\sigma^i)$ then $\Pr[c^i = c] = \Pr[c_b^i = c]$ for $b \in \{0, 1\}$. This follows from simulator non-interference. This gives us that

$$\Pr[c^i = c] = \Pr[c_0^i = c] \approx e^i$$

and

$$\Pr[c^i = c] = \Pr[c_1^i = c] \approx d^i \ .$$

This is a contradiction as it is not the case that $e^i \approx d^i$.

7.2 Excluded Settings

We now describe how to handle non-programmable random oracles as defined in e.g. [35] and other types of setup. It is straight forward to verify that the above proofs goes through as long as we can prove eventual revelation and simulator non-interference. Therefore all settings with eventual revelation and simulator non-interference are excluded, they do not allow to UC securely realize $\mathcal{F}_{\mathsf{cf}}$. We now give some examples.

Non-programmable Random Oracles. Adding a random oracle does not violate eventual revelation. It does, however, violate simulator non-interference, as it is the simulator which simulates the random oracle to the environment. We can, however, add a restricted observable global random oracle [17]. Since the restricted observable global random oracle cannot be programmed by the simulator it does not violate simulator non-interference. As a special case a global CRS is not enough to bring us out of the excluded settings.

2PC with Abort. We can also add a special type of ideal functionalities $\mathcal{F}_{\mathsf{2pca}}$ working as follows. In each round it has a direction, sending from Alice to Bob in odd rounds and from Bob to Alice in even rounds. We describe a round i from Alice to Bob. First both parties give an input x^i_{Alice} and x^i_{Bob}. If a party does not give an input, the ideal functionality uses 0. Then $\mathcal{F}_{\mathsf{2pca}}$ computes the corresponding outputs y^i_{Alice} and y^i_{Bob} given by a randomised function f^i being part of the description of $\mathcal{F}_{\mathsf{2pca}}$. The outputs are only allowed to depend on the current inputs x^i_{Alice} and x^i_{Bob} and fresh randomness, i.e., the rounds are run independently. Then it outputs y_{Alice} to Alice who gives an input m^i. If $m^i = \mathsf{abort}$, the output to Bob is abort. Otherwise the output to Bob is y_{Bob}. If a party aborts in any round, the other party aborts in all future rounds and proceed as if the other party aborted in all subsequent rounds rounds, no matter what values the other party sends. As before we can define the check point σ^i of Alice to be her state after computing m^i. Rounds with direction from Bob to Alice are the same except that it is Bob who gets to choose whether to abort by inputting m^i. It is easy to see that this setting has eventual revelation and simulator non-interference.

As for eventual revelation, consider a party $\mathcal{P}$ in the check point σ^i. If the other party would abort in the next round, then $\mathcal{P}$ will receive no more information from $\mathcal{F}_{\mathsf{2pca}}$. Hence in σ^i the party $\mathcal{P}$ has all the information needed to compute $c^i(\sigma^i)$. As for simulator non-interference, note that when computing $c^i(\sigma^i)$ one does not access $\mathcal{F}_{\mathsf{2pca}}$. Therefore $c^i(\sigma^i)$ depends only on the local state of the environment. Again, this would have been different in the programmable random oracle model.

Note that the above leaves a pretty pessimistic picture. Even with a non-programmable random oracles, time-lock puzzles, and access to unlimited 2PC with abort, one cannot UC securely implement even a simple task as fair coin-flip, which is arguable one of the simplest tasks where abort is an issue. The only

way to UC securely implement fair coin-flip is therefore to cheat in the model in a strong way, by for instance using a programmable random oracle.

.

References

1. M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Fair two-party computations via bitcoin deposits. In R. Böhme, M. Brenner, T. Moore, and M. Smith, editors, *FC 2014 Workshops*, volume 8438 of *LNCS*, pages 105–121. Springer, Heidelberg, Mar. 2014.
2. M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Secure multi-party computations on bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 443–458. IEEE Computer Society Press, May 2014.
3. M. Backes, D. Hofheinz, J. Müller-Quade, and D. Unruh. On fairness in simulatability-based cryptographic systems. In V. Atluri, P. Samarati, R. Küsters, and J. C. Mitchell, editors, *Proceedings of the 2005 ACM workshop on Formal methods in security engineering, FMSE 2005, Fairfax, VA, USA, November 11, 2005*, pages 13–22. ACM, 2005.
4. M. Backes, P. Manoharan, and E. Mohammadi. TUC: time-sensitive and modular analysis of anonymous communication. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*, pages 383–397. IEEE Computer Society, 2014.
5. C. Badertscher, P. Gazi, A. Kiayias, A. Russell, and V. Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *ACM CCS 2018*, pages 913–930. ACM Press, Oct. 2018.
6. C. Baum, B. David, and R. Dowsley. Insured mpc: Efficient secure computation with financial penalties. In *Financial Cryptography 2020 (To Appear)*, 2020. `https://eprint.iacr.org/2018/942`.
7. C. Baum, E. Orsini, and P. Scholl. Efficient secure multiparty computation with identifiable abort. In M. Hirt and A. D. Smith, editors, *TCC 2016-B, Part I*, volume 9985 of *LNCS*, pages 461–490. Springer, Heidelberg, Oct. / Nov. 2016.
8. M. Bellare, R. Dowsley, B. Waters, and S. Yilek. Standard security does not imply security against selective-opening. In D. Pointcheval and T. Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 645–662. Springer, Heidelberg, Apr. 2012.
9. 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.
10. N. Bitansky, S. Goldwasser, A. Jain, O. Paneth, V. Vaikuntanathan, and B. Waters. Time-lock puzzles from randomized encodings. In M. Sudan, editor, *ITCS 2016*, pages 345–356. ACM, Jan. 2016.
11. D. Boneh, J. Bonneau, B. Bünz, and B. Fisch. Verifiable delay functions. In H. Shacham and A. Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 757–788. Springer, Heidelberg, Aug. 2018.
12. D. Boneh and M. Naor. Timed commitments. In M. Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 236–254. Springer, Heidelberg, Aug. 2000.
13. J. Camenisch, M. Drijvers, T. Gagliardoni, A. Lehmann, and G. Neven. The wonderful world of global random oracles. In J. B. Nielsen and V. Rijmen, editors,

EUROCRYPT 2018, Part I, volume 10820 of *LNCS*, pages 280–312. Springer, Heidelberg, Apr. / May 2018.
14. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. `http://eprint.iacr.org/2000/067`.
15. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, Oct. 2001.
16. 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.
17. 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 2014*, pages 597–608. ACM Press, Nov. 2014.
18. I. Cascudo, I. Damgård, B. David, N. Döttling, R. Dowsley, and I. Giacomelli. Efficient UC commitment extension with homomorphism for free (and applications). In S. D. Galbraith and S. Moriai, editors, *ASIACRYPT 2019, Part II*, volume 11922 of *LNCS*, pages 606–635. Springer, Heidelberg, Dec. 2019.
19. I. Cascudo, I. Damgård, B. David, N. Döttling, and J. B. Nielsen. Rate-1, linear time and additively homomorphic UC commitments. In M. Robshaw and J. Katz, editors, *CRYPTO 2016, Part III*, volume 9816 of *LNCS*, pages 179–207. Springer, Heidelberg, Aug. 2016.
20. R. Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *18th ACM STOC*, pages 364–369. ACM Press, May 1986.
21. G. Couteau, B. Roscoe, and P. Ryan. Partially-fair computation from timed-release encryption and oblivious transfer. Cryptology ePrint Archive, Report 2019/1281, 2019. `https://eprint.iacr.org/2019/1281`.
22. B. David, P. Gazi, A. Kiayias, and A. Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In J. B. Nielsen and V. Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 66–98. Springer, Heidelberg, Apr. / May 2018.
23. C. Dwork, M. Naor, and A. Sahai. Concurrent zero-knowledge. In *30th ACM STOC*, pages 409–418. ACM Press, May 1998.
24. M. Fischlin, A. Lehmann, T. Ristenpart, T. Shrimpton, M. Stam, and S. Tessaro. Random oracles with(out) programmability. In M. Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 303–320. Springer, Heidelberg, Dec. 2010.
25. O. Goldreich. Concurrent zero-knowledge with timing, revisited. In *34th ACM STOC*, pages 332–340. ACM Press, May 2002.
26. C. Hazay, P. Scholl, and E. Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. In T. Takagi and T. Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 598–628. Springer, Heidelberg, Dec. 2017.
27. D. Hofheinz and V. Shoup. GNUC: A new universal composability framework. *Journal of Cryptology*, 28(3):423–508, July 2015.
28. 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.
29. Y. T. Kalai, Y. Lindell, and M. Prabhakaran. Concurrent general composition of secure protocols in the timing model. In H. N. Gabow and R. Fagin, editors, *37th ACM STOC*, pages 644–653. ACM Press, May 2005.

30. 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.
31. A. Kiayias, H.-S. Zhou, and V. Zikas. Fair and robust multi-party computation using a global transaction ledger. In M. Fischlin and J.-S. Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 705–734. Springer, Heidelberg, May 2016.
32. 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 2014*, pages 30–41. ACM Press, Nov. 2014.
33. R. Kumaresan, T. Moran, and I. Bentov. How to use bitcoin to play decentralized poker. In I. Ray, N. Li, and C. Kruegel, editors, *ACM CCS 2015*, pages 195–206. ACM Press, Oct. 2015.
34. U. Maurer. Constructive cryptography - A new paradigm for security definitions and proofs. In S. Mödersheim and C. Palamidessi, editors, *Theory of Security and Applications - Joint Workshop, TOSCA 2011, Saarbrücken, Germany, March 31 - April 1, 2011, Revised Selected Papers*, volume 6993 of *Lecture Notes in Computer Science*, pages 33–56. Springer, 2011.
35. J. B. Nielsen. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In M. Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 111–126. Springer, Heidelberg, Aug. 2002.
36. B. Pfitzmann and M. Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *2001 IEEE Symposium on Security and Privacy*, pages 184–200. IEEE Computer Society Press, May 2001.
37. K. Pietrzak. Simple verifiable delay functions. In A. Blum, editor, *ITCS 2019*, volume 124, pages 60:1–60:15. LIPIcs, Jan. 2019.
38. R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. 1996.
39. L. Rotem, G. Segev, and I. Shahaf. Generic-group delay functions require hidden-order groups. In *Eurocrypt 2020 (To Appear)*, 2020. `https://eprint.iacr.org/2020/225`.
40. V. Shoup. Lower bounds for discrete logarithms and related problems. In W. Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 256–266. Springer, Heidelberg, May 1997.
41. H. Wee. Zero knowledge in the random oracle model, revisited. In M. Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 417–434. Springer, Heidelberg, Dec. 2009.
42. B. Wesolowski. Efficient verifiable delay functions. In Y. Ishai and V. Rijmen, editors, *EUROCRYPT 2019, Part III*, volume 11478 of *LNCS*, pages 379–407. Springer, Heidelberg, May 2019.

Appendix A Additional Functionalities

Functionality $\mathcal{G}_{\mathsf{rpoRO}}$

$\mathcal{G}_{\mathsf{rpoRO}}$ is parameterized by an output size function ℓ and a security parameter τ, and keeps initially empty lists $\mathsf{List}_{\mathcal{H}}$,$\mathsf{prog}$.

Query: On input (HASH-QUERY, m) from party $(\mathcal{P}, \mathsf{sid})$ or $\mathcal{S}$, parse m as (s, m') and proceed as follows:

1. Look up h such that $(m, h) \in \mathsf{List}_{\mathcal{H}}$. If no such h exists, sample $h \xleftarrow{\$} \{0,1\}^{\ell(\tau)}$ and set $\mathsf{List}_{\mathcal{H}} = \mathsf{List}_{\mathcal{H}} \cup \{(m, h)\}$.
2. If this query is made by $\mathcal{S}$, or if $s \neq \mathsf{sid}$, then add (s, m', h) to the (initially empty) list of illegitimate queries $\mathcal{Q}_s$.
3. Send (HASH-CONFIRM, h) to the caller.

Observe: On input (OBSERVE, sid) from $\mathcal{S}$, if $\mathcal{Q}_{\mathsf{sid}}$ does not exist yet, set $\mathcal{Q}_{\mathsf{sid}} = \emptyset$. Output (LIST-OBSERVE, $\mathcal{Q}_{\mathsf{sid}}$) to $\mathcal{S}$.

Program: On input (PROGRAM-RO, m, h) with $h \in \{0,1\}^{\ell(\tau)}$ from $\mathcal{S}$, ignore the input if there exists $h' \in \{0,1\}^{\ell(\tau)}$ where $(m, h') \in \mathsf{List}_{\mathcal{H}}$ and $h \neq h'$. Otherwise, set $\mathsf{List}_{\mathcal{H}} = \mathsf{List}_{\mathcal{H}} \cup \{(m, h)\}$, $\mathsf{prog} = \mathsf{prog} \cup \{m\}$ and send (PROGRAM-CONFIRM) to $\mathcal{S}$.

IsProgrammed: On input (ISPROGRAMMED, m) from a party $\mathcal{P}$ or $\mathcal{S}$, if the input was given by $(\mathcal{P}, \mathsf{sid})$ then parse m as (s, m') and, if $s \neq \mathsf{sid}$, ignore this input. Set $b = 1$ if $m \in \mathsf{prog}$ and $b = 0$ otherwise. Then send (ISPROGRAMMED, b) to the caller.

Fig. 12. Restricted observable and programmable global random oracle functionality $\mathcal{G}_{\mathsf{rpoRO}}$ from [13].

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

$\mathcal{F}_{\mathsf{ct}}$ interacts with two parties $\mathcal{P}_1, \mathcal{P}_2$ and an adversary $\mathcal{S}$. The functionality will internally have two lists $\mathcal{M}, \mathcal{Q}$.

Toss: Upon receiving $(\mathsf{Toss}, \mathsf{sid}, m)$ from $\mathcal{P}_i \in \{\mathcal{P}_1, \mathcal{P}_2\}$ where $m \in \mathbb{N}$:

1. Add $(\Delta, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, (\mathsf{Toss}, m))$ to $\mathcal{Q}$ for an unused mid. Then send $(\mathsf{Toss}, \mathsf{sid}, \mathcal{P}_i, \mathsf{mid})$ to $\mathcal{S}$.
2. If both parties sent $(\mathsf{Toss}, \mathsf{sid}, m)$:
   (a) Uniformly sample m random elements $x_1, \ldots, x_m \xleftarrow{\$} \mathbb{F}$ and send $(\mathsf{Tossed}, \mathsf{sid}, m, \mathbb{F}, x_1, \ldots, x_m)$ to $\mathcal{S}$.
   (b) If $\mathcal{S}$ sends $(\mathsf{DeliverCoins}, \mathsf{sid})$ and $\mathcal{P}_j$ is corrupted then add $(\Delta, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_{3-j}, (\mathsf{Coins}, x_1, \ldots, x_m))$ for a fresh mid to $\mathcal{Q}$.
   (c) Send $(\mathsf{DeliverCoins}, \mathsf{sid}, \mathcal{P}_{3-j}, \mathsf{mid})$ to $\mathcal{S}$.

Fetch Message: Upon receiving $(\mathsf{FetchMsg}, \mathsf{sid})$ by $\mathcal{P} \in \{\mathcal{P}_1, \mathcal{P}_2\}$ retrieve the set L of all entries $(\mathcal{P}, \mathsf{sid}, \cdot)$ in $\mathcal{M}$, remove L from $\mathcal{M}$ and return $(\mathsf{Output}, \mathsf{sid}, L)$ to $\mathcal{P}$.

Scheduling: On input of $\mathcal{S}$:

- If $\mathcal{S}$ sent $(\mathsf{Deliver}, \mathsf{sid}, \mathsf{mid})$ then remove each $(\mathtt{cnt}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}, m)$ from $\mathcal{Q}$ and add $(\mathcal{P}, \mathsf{sid}, m)$ to $\mathcal{M}$.
- If $\mathcal{S}$ sent (Abort) add $(\mathcal{P}_S, \mathsf{sid}, \mathsf{Abort})$, $(\mathcal{P}_R, \mathsf{sid}, \mathsf{Abort})$ to $\mathcal{M}$ and ignore all further calls to the functionality except to **Fetch Message**.

Tick: Upon receiving $(\mathsf{Tick}, \mathsf{sid})$ from $\mathcal{S}$:

1. For each query $(0, \mathsf{mid}, \mathsf{sid}, \mathcal{P}, m) \in \mathcal{Q}$:
   (a) Remove $(0, \mathsf{mid}, \mathsf{sid}, \mathcal{P}, m)$ from $\mathcal{Q}$.
   (b) Add $(\mathcal{P}, \mathsf{sid}, m)$ to $\mathcal{M}$.
2. Replace each $(\mathtt{cnt}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}, m)$ in $\mathcal{Q}$ with $(\mathtt{cnt} - 1, \mathsf{mid}, \mathsf{sid}, \mathcal{P}, m)$.

Fig. 13. Functionality $\mathcal{F}_{\mathsf{ct}}$ for Coin Tossing.

Appendix B Additively Homomorphic Commitments with Delayed Opening

We present a protocol π_{ahcom} for additively homomorphic commitments realizing functionality $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ from Section 6.1 in Figures 14, 15 and 16. Protocol π_{ahcom} is based on a protocol for additively homomorphic commitments with multiple verifiers presented in [18]. As it is the case in [18], we actually construct a protocol for commitments to random messages rather than a protocol for commitments to arbitrary messages. As observed in [19,18], such a protocol for additively homomorphic commitments to random messages can be trivially transformed into a protocol that realizes a functionality supporting arbitrary message. In order to obtain π_{ahcom} we add a delayed opening phase to the protocol of [18] and modify it in the following way:

Delayed Communication via $\mathcal{F}_{\mathsf{smt,delay}}^{\Delta}$: All messages between sender $\mathcal{P}_S$ and $\mathcal{P}_R$ are exchange through secure channels with delays modeled by $\mathcal{F}_{\mathsf{smt,delay}}^{\Delta}$

with a maximum message delay Δ, whereas the protocol of [18] employs standard secure channels with no delay. This is necessary in order to capture the fact that our additively homomorphic commitment functionality with delayed openings $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ allows for adversarially controlled delays in the delivery of commitments and openings to honest parties.

Support for only one receiver: Protocol π_{ahcom} only realizes additively homomorphic commitments with delayed opening for one receiver $\mathcal{P}_R$ instead of supporting multiple receivers as in the protocol of [18]. While it would be possible to obtain a multiple receiver version of our protocol with delayed openings, this would require a broadcast functionality with adversarialy controlled message delays (*i.e.* a generalization of $\mathcal{F}_{\mathsf{smt,delay}}^{\Delta}$ for multiple receivers) in order to ensure that all messages exchanged are eventually received by all receivers. Such a multiple receiver version of π_{ahcom} is not necessary for realizing two party computation with output independent abort as defined in Section 6 and is thus left as future work.

Hardcoded global random oracle commitments: Instead of basing our construction on a generic (non-homomorphic) commitment functionality as in [18], we explicitly use canonical random oracle commitments based on a restricted observable and programmable global random oracle $\mathcal{G}_{\mathsf{rpoRO}}$ as presented in [13]. This concrete construction of non-homomorphic commitments works by having the sender sample some randomness w and compute a commitment to a message m by querying the global random oracle on $(m|w)$ and sending the output $\mathtt{scom}$ to the receiver as a commitment. Later on, the receiver can check that an opening $(m'|w')$ is valid by querying the global random oracle on $(m'|w')$ and verifying that the output $\mathtt{scom}'$ is equal to $\mathtt{scom}$. This allows us to obtain time lock puzzles containing an opening $(m'|w')$ of such commitments $\mathtt{scom}$ through $\mathcal{F}_{\mathsf{tlp}}$ in such a way that this opening can be verified locally by the receiver once the time lock puzzle is solved (*i.e.* after receiving and solving the time lock puzzle containing $(m'|w')$, the receiver can simply query the global random oracle in order to verify it without any further communication with the sender).

While these modifications allow us to obtain delayed openings, π_{ahcom}'s procedures for committing, performing addition of commitments and opening essentially work as in the protocol of [18] for the case where there is only one verifier. In this case, the only difference between π_{ahcom} and [18] is that π_{ahcom} specifically uses global random oracle commitments (as proven UC-secure in [13]) instead of a non-homomorphic commitment functionality as in [18]. Hence, all the arguments in the security proof for the protocol of [18] carry on to π_{ahcom} and we only need to prove security of the delayed opening procedure in order to prove Theorem 4.

B.1 Security Analysis

The security of π_{ahcom} is formally expressed in Theorem 4.

Protocol π_{ahcom}

Let C be a systematic binary linear $[n, k, s]$ code, where s is the statistical security parameter and n is $k + O(s)$. Let τ be a computational security parameter. Let $\mathcal{H}$ be a family of linear almost universal hash functions $\mathbf{H} : \{0,1\}^m \to \{0,1\}^l$. Let $\mathsf{PRG} : \{0,1\}^\ell \to \{0,1\}^{m+l}$ be a pseudorandom generator. Protocol π_{ahcom} is run by a sender $\mathcal{P}_S$ and a receiver $\mathcal{P}_R$, who interact with $\mathcal{G}_{\mathsf{rpoRO}}$ (having output size τ), instances of $\mathcal{F}_{\mathsf{tlp}}$ and instances of $\mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$ with message delay parameter Δ, proceeding as follows:

Commitment Phase

1. On input $(\mathsf{commit}, \mathsf{sid}, \mathsf{ssid}_1, \ldots, \mathsf{ssid}_m, \mathcal{P}_S, \mathcal{P}_R)$, $\mathcal{P}_S$ proceeds as follows:
   (a) For $i \in [n]$ and $j \in \{0,1\}$ $\mathcal{P}_S$ commits to $\vec{s}_{i,j}$ by performing the following steps: (1) Sample $w_{i,j} \leftarrow \{0,1\}^\tau$ and $\vec{s}_{i,j} \leftarrow \{0,1\}^\ell$; (2) Send $(\text{HASH-QUERY}, (\vec{s}_{i,j}|w_{i,j}))$ to $\mathcal{G}_{\mathsf{rpoRO}}$, obtaining $(\text{HASH-CONFIRM}, \mathtt{scom}_{i,j})$; (3) Send $\mathtt{scom}_{i,j}$ to $\mathcal{P}_R$ via $\mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$.
   (b) Compute $\mathbf{R_j}[i, \cdot] = \mathsf{PRG}(\vec{s}_{i,j})$ and set $\mathbf{R} = \mathbf{R_0} + \mathbf{R_1}$ so that $\mathbf{R_0}, \mathbf{R_1}$ forms an additive secret sharing of $\mathbf{R}$.
   (c) Adjust the bottom $n - k$ rows of $\mathbf{R}$ so that all columns are codewords in C by constructing a matrix $\mathbf{W}$ with dimensions as $\mathbf{R}$ and 0s in the top k rows, such that $\mathbf{A} := \mathbf{R} + \mathbf{W} \in \mathsf{C}^{\odot \mathsf{m+l}}$ (recall that C is systematic). Set $\mathbf{A_0} = \mathbf{R_0}, \mathbf{A_1} = \mathbf{R_1} + \mathbf{W}$ and send to $\mathcal{P}_R$ via $\mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$ $(\mathsf{sid}, \mathsf{ssid}_1, \ldots, \mathsf{ssid}_m, \mathbf{W})$ (only sending the bottom $n - k = O(s)$ rows).
2. Upon receiving all messages $\mathtt{scom}_{i,j}$ and $(\mathsf{sid}, \mathsf{ssid}_1, \ldots, \mathsf{ssid}_m, \mathbf{W})$ from $\mathcal{P}_S$, $\mathcal{P}_R$ proceeds as follows:
   (a) Sample $\vec{r}' \leftarrow \{0,1\}^\ell$, and send it to $\mathcal{P}_S$ via $\mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$.
3. Upon receiving $\vec{r}'$ from $\mathcal{P}_R$ via $\mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$, $\mathcal{P}_S$ proceeds as follows:
   (a) Use $\vec{r}'$ as a seed for a random function $\mathbf{H} \in \mathcal{H}$ (note that we identify the function with its matrix and all functions in $\mathcal{H}$ are linear).
   (b) Set matrices $\mathbf{P}$, $\mathbf{P_0}$ and $\mathbf{P_1}$ as the first l columns of $\mathbf{A}$, $\mathbf{A_0}$ and $\mathbf{A_1}$, respectively, and remove these columns from $\mathbf{A}$, $\mathbf{A_0}$ and $\mathbf{A_1}$. Renumber the remaining columns of $\mathbf{A}$, $\mathbf{A_0}$ and $\mathbf{A_1}$ from 1 and associate each $\mathtt{scom}_{i,j}$ (for $i \in [n]$ in step 1) with a different column index in these matrices. Notice that $\mathbf{P} = \mathbf{P_0} + \mathbf{P_1}$.
   (c) For $i \in \{0,1\}$, compute $\mathbf{T}_i = \mathbf{A}_i\mathbf{H} + \mathbf{P}_i$ and send $(\mathsf{sid}, \mathsf{ssid}_1, \ldots, \mathsf{ssid}_m, \mathbf{T_0}, \mathbf{T_1})$ to $\mathcal{P}_R$ via $\mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$. Note that $\mathbf{AH} + \mathbf{P} = \mathbf{A_0H} + \mathbf{P_0} + \mathbf{A_1H} + \mathbf{P_1} = \mathbf{T_0} + \mathbf{T_1}$, and $\mathbf{AH} + \mathbf{P} \in \mathsf{C}^{\odot \mathsf{l}}$.

Fig. 14. Modified version of the Commit phase for the protocol π_{ahcom} of [18] with delayed opening.

Theorem 4. *Protocol π_{ahcom} UC-realizes $\mathcal{F}^{\Delta,\delta}_{\mathsf{ahcom}}$ in the $\mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{tlp}}, \mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$-hybrid model with computational security against a static adversary. Formally, there exists a simulator $\mathcal{S}$ such that for every static adversary $\mathcal{A}$, and any environment $\mathcal{Z}$, the environment cannot distinguish π_{ahcom} composed with $\mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{tlp}}, \mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$ and $\mathcal{A}$ from $\mathcal{S}$ composed with $\mathcal{F}^{\Delta,\delta}_{\mathsf{ahcom}}$. That is:*

$$\mathsf{IDEAL}_{\mathcal{F}^{\Delta,\delta}_{\mathsf{ahcom}},\mathcal{S},\mathcal{Z}} \approx_{\mathsf{c}} \mathsf{HYBRID}^{\mathcal{G}_{\mathsf{rpoRO}},\mathcal{F}_{\mathsf{tlp}},\mathcal{F}^{\Delta}_{\mathsf{smt,delay}}}_{\pi_{\mathsf{ahcom}},\mathcal{A},\mathcal{Z}} .$$

Protocol π_{ahcom}

Addition of Commitments

1. On input $(\mathsf{add}, \mathsf{sid}, \mathsf{ssid}_1, \mathsf{ssid}_2, \mathsf{ssid}_3, \mathcal{P}_S, \mathcal{P}_R)$, $\mathcal{P}_S$ finds indexes i and j corresponding to ssid_1 and ssid_2 respectively and check that ssid_3 is unused. $\mathcal{P}_S$ appends the column $\mathbf{A}[\cdot, i] + \mathbf{A}[\cdot, j]$ to $\mathbf{A}$, likewise appends to $\mathbf{A_0}$ and $\mathbf{A_1}$ the sum of their i-th and j-th columns, and associates ssid_3 with the new column index. $\mathcal{P}_S$ sends $(\mathsf{add}, \mathsf{sid}, \mathsf{ssid}_1, \mathsf{ssid}_2, \mathsf{ssid}_3)$ to $\mathcal{P}_R$ via $\mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$.
2. Upon receiving $(\mathsf{add}, \mathsf{sid}, \mathsf{ssid}_1, \mathsf{ssid}_2, \mathsf{ssid}_3)$ from $\mathcal{P}_S$ via $\mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$, $\mathcal{P}_R$ stores the message.

Opening

1. On input $(\mathsf{reveal}, \mathsf{sid}, \mathsf{ssid}_1, \ldots, \mathsf{ssid}_o)$, $\mathcal{P}_S$ finds the set $J = \{j_1, \ldots, j_o\}$ of indexes associated to $\mathsf{ssid}_1, \ldots, \mathsf{ssid}_o$ and sends $(\mathbf{A_0}[\cdot, j], \mathbf{A_1}[\cdot, j])_{j \in J}$ to $\mathcal{P}_R$ via $\mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$.
2. Upon receiving message $(\mathbf{A_0}[\cdot, j], \mathbf{A_1}[\cdot, j])_{j \in J}$ from $\mathcal{P}_S$ via $\mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$, $\mathcal{P}_R$ samples $\vec{r} \leftarrow \{0,1\}^n$ and sets the diagonal matrix $\mathbf{\Delta}$ such that it contains $\vec{r}[1], \ldots, \vec{r}[n]$ in the diagonal. Send $\vec{r}$ to $\mathcal{P}_S$ via $\mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$.
3. Upon receiving $\vec{r}$ from $\mathcal{P}_R$ via $\mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$, $\mathcal{P}_S$ opens commitments $\mathtt{scom}_{i,\vec{r}[i]}$ by sending $(\vec{s}_{i,\vec{r}[i]} | w_{i,\vec{r}[i]})$ to $\mathcal{P}_R$ via $\mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$ and halts.
4. Upon receiving $(\vec{s}_{i,\vec{r}[i]} | w_{i,\vec{r}[i]})$ from $\mathcal{P}_S$ via $\mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$ for $i \in [n]$, $\mathcal{P}_R$ proceeds as follows:
   (a) For $i \in [n]$, check the validity of the openings to $\mathtt{scom}_{i,\vec{r}[i]}$ by sending $(\textsc{Hash-Query}, (\vec{s}_{i,\vec{r}[i]} | w_{i,\vec{r}[i]}))$ to $\mathcal{G}_{\mathsf{rpoRO}}$, obtaining $(\textsc{Hash-Confirm}, \overline{\mathtt{scom}}_{i,\vec{r}[i]})$ and aborting if $\overline{\mathtt{scom}}_{i,\vec{r}[i]} \neq \mathtt{scom}_{i,\vec{r}[i]}$.
   (b) Compute $\mathbf{S}[i, \cdot] = \mathsf{PRG}(\vec{s}_{i,\vec{r}[i]})$, obtaining a matrix $\mathbf{S}$. Set $\mathbf{B} = \mathbf{\Delta W} + \mathbf{S}$. Define the matrix $\mathbf{Q}$ as the first l columns of $\mathbf{B}$ and remove these columns from $\mathbf{B}$, renumbering the remaining columns from 1. Check that $\mathbf{\Delta T_1} + (\mathbf{I} - \mathbf{\Delta})\mathbf{T_0} = \mathbf{BH} + \mathbf{Q}$ and that $\mathbf{T_0} + \mathbf{T_1} \in \mathsf{C}^{\odot \mathsf{I}}$. If any check fails, abort.
   (c) For every message $(\mathsf{add}, \mathsf{sid}, \mathsf{ssid}_1, \mathsf{ssid}_2, \mathsf{ssid}_3)$ received from $\mathcal{P}_S$, append $\mathbf{B}[\cdot, j] + \mathbf{B}[\cdot, i]$ to $\mathbf{B}$, where i and j are the index corresponding to ssid_1 and ssid_2 respectively and associate ssid_3 with the new column index.
   (d) For every $j \in J$, check that $\mathbf{A_0}[\cdot, j] + \mathbf{A_1}[\cdot, j] \in \mathsf{C}$ and that, for $i \in [n]$, it holds that $\mathbf{B}[i, j] = \mathbf{A}_{\vec{\mathbf{r}}[\mathbf{i}]}[i, j]$ (recall that $\vec{r}[i]$ is the i-th entry on the diagonal of $\mathbf{\Delta}$). If all checks succeed, for every $j \in J$, output the first k positions in $\mathbf{A_0}[\cdot, j] + \mathbf{A_1}[\cdot, j]$ as the opened string and halt. Otherwise, abort by outputting $(\mathsf{sid}, \mathsf{ssid}_j, \perp)$.

Fig. 15. Addition of commitments and modified opening phase for the protocol π_{ahcom} of [18] with delayed opening.

Proof. (Sketch) As it is the case in [18], we actually construct a protocol for commitments to random messages rather than a protocol for commitments to arbitrary messages. As observed in [19,18], such a protocol for additively homomorphic commitments to random messages can be trivially transformed into a protocol that realizes a functionality supporting arbitrary message. We omit this straightforward transformation and focus on the case of random messages.

We construct a simulator $\mathcal{S}$ that interacts with the functionality $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$, the environment $\mathcal{Z}$ an internal copy of the adversary $\mathcal{A}$, towards which it executes π_{ahcom} and simulates $\mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{tlp}}, \mathcal{F}_{\mathsf{smt,delay}}^{\Delta}$. As observed previously, we can use the instructions from the simulator for the protocol of [18] in the case with 1 verifier in order to construct $\mathcal{S}$ for the commitment, addition and opening phases of π_{ahcom}. The only difference is that instead of simulating a non-homomorphic commitment functionality as in [18], $\mathcal{S}$ follows the instructions of a simulator for the global random oracle commitment scheme of [13] (used explicitly by π_{ahcom} in place of a commitment functionality) in order to extract the messages from $\mathcal{A}$ when the simulator of [13] would do so by simulating the commitment functionality towards $\mathcal{A}$. Besides that, $\mathcal{S}$ simulates message delivery through $\mathcal{F}_{\mathsf{smt,delay}}^{\Delta}$ exactly as indicated by $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$.

In order to simulate the delayed opening phase, $\mathcal{S}$ takes advantage of its simulation of $\mathcal{F}_{\mathsf{tlp}}$ towards $\mathcal{A}$. In case the adversary $\mathcal{A}$ corrupts $\mathcal{P}_S$, $\mathcal{S}$ learns $(\vec{s}_{i,\vec{r}[i]}|w_{i,\vec{r}[i]})$ for $i \in [n]$ (resp. $(\mathbf{A_0}[\cdot,j], \mathbf{A_1}[\cdot,j])_{j\in J})|w)$) contained in $\mathtt{puz}_{\vec{s}}$ (resp. $\mathtt{puz}_o$) by observing the queries of $\mathcal{A}$ to the simulated $\mathcal{F}_{\mathsf{tlp}}$. $\mathcal{S}$ checks that these values are valid openings for the global random oracle commitments $\mathtt{scom}_{\vec{s}}, \mathtt{scom}_o$ sent by $\mathcal{A}$ are valid by following the instructions of the global random oracle commitment simulator of [13]. If these are valid openings, $\mathcal{S}$ uses $(\mathbf{A_0}[\cdot,j], \mathbf{A_1}[\cdot,j])_{j\in J})$ and $(\vec{s}_{i,\vec{r}[i]}|w_{i,\vec{r}[i]})$ for $i \in [n]$ to execute the steps of an honest receiver $\mathcal{P}_R$ in order to check that the opening each commitment identified by $\mathsf{ssid}_1, \ldots, \mathsf{ssid}_o$ is valid. If any of these checks fails, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs and aborts. Otherwise, it sends $(\mathsf{DOpen}, \mathsf{sid}, \mathsf{ssid}_1, \ldots, \mathsf{ssid}_o, \delta)$ to $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ and halts. Essentially, upon receiving the messages for a delayed commitment, $\mathcal{S}$ checks that they will result in an honest receiver accepting this delayed opening, either performing a similar delayed opening via $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ if this is the case, or aborting and outputting what $\mathcal{A}$ outputs if the checks fail.

If $\mathcal{A}$ corrupts $\mathcal{P}_R$, upon receiving $(\mathsf{DOpen}, \mathsf{sid}, \mathsf{cid}, \ldots, \mathsf{cid}_o, \delta)$ from $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$, $\mathcal{S}$ simulates the steps of an honest sender for a delayed commitment, with the difference that the puzzles $\mathtt{puz}_{\vec{s}}, \mathtt{puz}_o$ are generated for random messages. Upon receiving $(\mathsf{DOpen}, \mathcal{S}, (\mathsf{ssid}_i, \boldsymbol{x}_i)_{i\in\{1,\ldots,o\}})$, $\mathcal{S}$ simulates the solutions of puzzles in a way that they yield delayed openings to messages $\boldsymbol{x}_i$ for commitments identified by ssid_i. Notice that $\mathcal{S}$ can do that following the instruction of the simulator of [18] for the case of a corrupted receiver in order to obtain $(\mathbf{A_0}[\cdot,j], \mathbf{A_1}[\cdot,j])_{j\in J})$ consistent with $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_o$ obtained from $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ and then make $\mathtt{puz}_o$ open to these simulated values when $\mathcal{A}$ queries the simulated $\mathcal{F}_{\mathsf{tlp}}$ to obtain this solution. □

Protocol π_{ahcom}

Delayed Opening

1. On input $(\mathsf{reveal}, \mathsf{sid}, \mathsf{ssid}_1, \ldots, \mathsf{ssid}_o)$, $\mathcal{P}_S$ finds the set $J = \{j_1, \ldots, j_o\}$ of indexes associated to $\mathsf{ssid}_1, \ldots, \mathsf{ssid}_o$, samples $w_o \xleftarrow{\$} \{0,1\}^\tau$ and commits to $(\mathbf{A_0}[\cdot, j], \mathbf{A_1}[\cdot, j])_{j \in J}$ by sending $(\text{HASH-QUERY}, (((\mathbf{A_0}[\cdot, j], \mathbf{A_1}[\cdot, j])_{j \in J})|w_o))$ to $\mathcal{G}_{\mathsf{rpoRO}}$, obtaining $(\text{HASH-CONFIRM}, \mathtt{scom}_o)$. $\mathcal{P}_S$ commits to $(\mathsf{sid}, \mathsf{ssid}_1, \ldots, \mathsf{ssid}_o, (\mathbf{A_0}[\cdot, j], \mathbf{A_1}[\cdot, j])_{j \in J})$ towards $\mathcal{P}_R$ by sending $\mathtt{scom}_o$ to $\mathcal{P}_R$ via $\mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$.
2. Same as Step 2 of Opening Phase, except that $\mathcal{P}_R$ proceeds upon receiving $\mathtt{scom}_o$ from $\mathcal{P}_S$ via $\mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$.
3. Upon receiving $\vec{r}$ from $\mathcal{P}_R$ via $\mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$, $\mathcal{P}_S$ creates time lock puzzles containing the opening of commitments $\mathtt{scom}_{i,\vec{r}[i]}$ sends $(\mathsf{CreatePuzzle}, \mathsf{sid}, \delta, \big((\vec{s}_{1,\vec{r}[1]}|w_{1,\vec{r}[1]}), \ldots, (\vec{s}_{n,\vec{r}[n]}|w_{n,\vec{r}[n]})\big))$ (resp. $(\mathsf{CreatePuzzle}, \mathsf{sid}, \delta, (((\mathbf{A_0}[\cdot, j], \mathbf{A_1}[\cdot, j])_{j \in J})|w_o))$) to $\mathcal{F}_{\mathsf{tlp}}$, obtaining $(\mathsf{CreatedPuzzle}, \mathsf{sid}, \mathtt{puz}_{\vec{s}})$ (resp. $(\mathsf{CreatedPuzzle}, \mathsf{sid}, \mathtt{puz}_o)$). $\mathcal{P}_S$ sends $\mathtt{puz}_{\vec{s}}, \mathtt{puz}_o$ to $\mathcal{P}_R$ via $\mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$. In order to determine the earliest point when these time lock puzzles can be opened, $\mathcal{P}_S$ parses $\mathtt{puz}_{\vec{s}} = (\mathtt{st} - \vec{s}_0, \Gamma - \vec{s}, \mathtt{tag} - \vec{s})$, sets $\mathtt{cst} - \vec{s} = 0$ and performs the following loop:
   (a) Send $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st} - \vec{s}_{\mathtt{cst}-\vec{s}})$ to $\mathcal{F}_{\mathsf{tlp}}$.
   (b) Send $(\mathsf{Output}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{tlp}}$ and checks that there is an entry $(\mathcal{P}_R, \mathtt{st} - \vec{s}_{\mathtt{cst}-\vec{s}}, \mathtt{st} - \vec{s}_{\mathtt{cst}-\vec{s}+1})$ in L_i. If yes, increment $\mathtt{cst} - \vec{s}$.
   (c) If $\mathtt{cst} - \vec{s} = \Gamma$, $\mathcal{P}_S$ outputs $(\mathsf{DOpened}, \mathsf{sid}, \mathsf{ssid}_1, \ldots, \mathsf{ssid}_o)$ and exits the loop.
4. Upon receiving time lock puzzles $\mathtt{puz}_{\vec{s}}, \mathtt{puz}_o$ via $\mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$, $\mathcal{P}_R$ uses $\mathcal{F}_{\mathsf{tlp}}$ to obtain $(\vec{s}_{i,\vec{r}[i]}|w_{i,\vec{r}[i]})$ (resp. $((\mathbf{A_0}[\cdot, j], \mathbf{A_1}[\cdot, j])_{j \in J})|w_o)$), *i.e.*, $\mathcal{P}_R$ parses $\mathtt{puz}_{\vec{s}} = (\mathtt{st} - \vec{s}_0, \Gamma - \vec{s}, \mathtt{tag} - \vec{s})$ (resp. $\mathtt{puz}_o = (\mathtt{st} - o_0, \Gamma - o, \mathtt{tag} - o)$), sets $\mathtt{cst} - \vec{s} = 0$ (resp. $\mathtt{cst} - o = 0$) and performs the following loop:
   (a) Send $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st} - \vec{s}_{\mathtt{cst}-\vec{s}})$ (resp. $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st} - \vec{s}_{\mathtt{cst}-o})$) to $\mathcal{F}_{\mathsf{tlp}}$.
   (b) Send $(\mathsf{Output}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{tlp}}$ and checks that there is an entry $(\mathcal{P}_R, \mathtt{st} - \vec{s}_{\mathtt{cst}-\vec{s}}, \mathtt{st} - \vec{s}_{\mathtt{cst}-\vec{s}+1})$ (resp. $(\mathcal{P}_R, \mathtt{st} - \vec{s}_{\mathtt{cst}-o}, \mathtt{st} - \vec{s}_{\mathtt{cst}-o+1})$) in L_i. If yes, increment $\mathtt{cst} - \vec{s}$ (resp. $\mathtt{cst} - o$).
   (c) If $\mathtt{cst} - \vec{s} = \Gamma$ (resp. $\mathtt{cst} - o = \Gamma$), send $(\mathsf{GetMsg}, \mathsf{sid}, \mathtt{puz}_{\vec{s}}, \mathtt{st} - \vec{s}_{\mathtt{cst}-\vec{s}})$ $((\mathsf{GetMsg}, \mathsf{sid}, \mathtt{puz}_o, \mathtt{st} - o_{\mathtt{cst}-o}))$, obtaining $(\mathsf{GetMsg}, \mathsf{sid}, \mathtt{puz}_{\vec{s}}, \mathtt{st} - \vec{s}_{\mathtt{cst}-\vec{s}}, \big((\vec{s}_{1,\vec{r}[1]}|w_{1,\vec{r}[1]}), \ldots, (\vec{s}_{n,\vec{r}[n]}|w_{n,\vec{r}[n]})\big))$ (resp. $(\mathsf{GetMsg}, \mathsf{sid}, \mathtt{puz}_o, \mathtt{st} - o_{\mathtt{cst}-o}, (((\mathbf{A_0}[\cdot, j], \mathbf{A_1}[\cdot, j])_{j \in J})|w_o))$). If both $\big((\vec{s}_{1,\vec{r}[1]}|w_{1,\vec{r}[1]}), \ldots, (\vec{s}_{n,\vec{r}[n]}|w_{n,\vec{r}[n]})\big)$ and $(((\mathbf{A_0}[\cdot, j], \mathbf{A_1}[\cdot, j])_{j \in J})|w_o))$ have been obtained, $\mathcal{P}_R$ exits the loop and proceeds to the next step.
5. For each commitment $\mathtt{scom}$ where $\mathtt{scom} = \mathtt{scom}_{i,\vec{r}[i]}$ (resp. $\mathtt{scom}_o$) and message of the form $(m|w)$ where $m = \vec{s}_{i,\vec{r}[i]}$ and $w = w_{i,\vec{r}[i]}$ (resp. $m = ((\mathbf{A_0}[\cdot, j], \mathbf{A_1}[\cdot, j])_{j \in J})$ and $w = w_o$), $\mathcal{P}_R$ checks the validity of the openings obtained from the time lock puzzles by sending $(\text{HASH-QUERY}, (m|w)$ to $\mathcal{G}_{\mathsf{rpoRO}}$, obtaining $(\text{HASH-CONFIRM}, \overline{\mathtt{scom}})$ and aborting if $\overline{\mathtt{scom}} \neq \mathtt{scom}$.
6. $\mathcal{P}_R$ uses $(\vec{s}_{i,\vec{r}[i]}|w_{i,\vec{r}[i]})$ for $i \in [n]$ and $(\mathsf{sid}, \mathsf{ssid}_1, \ldots, \mathsf{ssid}_o, (\mathbf{A_0}[\cdot, j], \mathbf{A_1}[\cdot, j])_{j \in J})|w)$ obtained in the previous step to execute Step 4 of the Opening Phase.

Fig. 16. Delayed Opening phase for the protocol π_{ahcom} of [18] with delayed opening.