CRAFT: Composable Randomness and Almost Fairness from Time

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. Cryptographic protocols often need to encompass time, e.g. for time outs. Modeling time formally is therefore crucial, as security of protocols can then be proven under more realistic assumptions. This is particularly important when considering composition, as protocols are rarely used in a stand-alone setting. This work extends the recent TARDIS model of abstract composable time (ACT) to the case of multiparty functionalities encompassing communication, publicly verifiable time-based primitives and secure computation. We model delayed multiparty communication through an ACT treatment of broadcast channels and public ledgers. Next, we introduce a publicly verifiable time-lock puzzle (TLP) functionality which we realize by showing that the TLP construction from TARDIS is publicly verifiable. Finally, we show that these new primitives can be used as building blocks for obtaining highly efficient composable randomness beacons and MPC with output independent abort and financial fairness.

1 Introduction

Time has always been an important, although sometimes overlooked, resource in cryptography. Interactive protocols make implicit and explicit assumptions about rounds or message delivery time: for example, protocols often consider a party as corrupt if it does not send information within a certain time frame. But

[⋆] 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) and COBRA.

[‡] Supported by the Danish Independent Research Council under Grant-ID DFF-8021-00366B (BETHE) and COBRA.

the concrete assumptions and consequences of events in implementations are often under-specified. This is particularly problematic when building complex protocols using time-related constructions which recently gained more interest. Examples for these are Time-Lock Puzzles (TLPs) [25] and Verifiable Delay Functions (VDFs) [10] .

In order to make provable security guarantees for composed protocols, the current default tool is the Universal Composability (UC) framework [13]. Unfortunately, the UC framework is inherently sequential: it models protocols as communicating Turing Machines, and only one such Turing Machine can be active at a time. This means that a notion of passing time has to be added in order to analyze time-related protocols in UC. To this end, Katz et al. [20] introduced the idea of a Global Clock, which is a global functionality that can be accessed by parties and functionalities. Crucially, it shows the same passing of time to every caller and only progresses if all honest parties permit it to.

Recently, Baum et al. [6] developed a notion of *Abstract Composable Time* (ACT) where functionalities might be aware of centrally-controlled ticks, while parties are oblivious to the progression of time and only interact based on events that occur at ideal functionalities. This allows to analyze the guarantees that a protocol gives under a variety of choices for time-related parameters of the functionalities that it uses. For example, ACT allows to prove statements about *the same protocol* under different models of message delivery.

Baum et al. [6] presented (among other results) constructions in the ACT framework for Time-Lock Puzzles (TLPs) as well as Two-Party Computation with Output-Independent Abort (OIA) and fair coin tossing. However, these results do not seem to leverage the full strength of their ACT framework. As an example, TLPs have found many interesting applications in ledger-based protocols, so constructions in their framework should work with many parties and ledgers as a primitive should be expressed naturally in ACT. Another interesting subject that should be studied with respect to composability are randomness beacons from time-based constructions. Finally, OIA does seem to be an interesting property for secure computation with more than two parties. In particular, combining it with financial incentives would yield a completely new direction for protocols that punish cheating behavior.

1.1 Our Contributions

In this work, we make the following contributions:

Modeling multiparty communication in ACT. We present a formalization of broadcast channels and public ledgers in the ACT framework. While the broadcast is a generalization of the secure message transmission functionality of [6], the ledger functionality adapts the work of [4] to ACT. This is a particularly interesting result, as [4] requires a Global Clock functionality.

Publicly Verifiable Time-Based Primitives in ACT. We introduce the notion of publicly verifiable TLPs and present an ideal functionality as well as a construction for this primitive in the ACT framework. We show that the TLP construction of [6] can already be proven to be publicly verifiable.

Multiparty Computation with Output Independent Abort (OIA-MPC). We realize Multiparty Computation (MPC) with Output-Independent Abort in the ACT framework by combining a synchronous broadcast channel together with a standard MPC functionality and multi-receiver commitments with delayed openings (which are themselves constructed from TLPs). To the best of our knowledge, no such protocol is known at the time of writing.

Publicly Verifiable OIA-MPC with Financial Penalties. We also generalize the idea of financial punishments for cheaters in MPC [8,2,5] and construct the first MPC protocol where adversaries must make the decision to cheat (and be punished) *before the output is known to them.* As an intermediate result we construct publicly verifiable OIA-MPC, which requires an MPC protocol with certain public verifiability properties as in [5] apart from publicly verifiable TLPs.

Composable Randomness Beacons. We provide a construction for unbiasable and publicly verifiable randomness beacons based on delayed broadcast channels or public ledgers plus publicly verifiable TLPs. To the best of our knowledge, this is the first unbiasable randomness beacon using time-based primitives to be proven secure, and the first to achieve Universal Composability. These constructions and proofs require not only publicly verifiable TLPs but careful modeling of delays in broadcast channels/public ledgers versus the TLP delays.

1.2 Related Work

The recent work of Baum et al. [6] introduced the first construction of a composable Time-Lock Puzzle. This is in comparison to previous constructions such as [25,11,9] that were only proven to be stand-alone secure. The related notion of verifiable delay functions has been investigated in [10,23,28]. Also for these constructions, composition guarantees have so far not been shown. The lack of composability guarantees for both of these primitives is a problem in protocol design, as they are used as building blocks in more complex protocols. Their security is not guaranteed when they are composed with other primitives.

Randomness beacons that resist adversarial bias have been constructed based on verifiable delay functions [10] and on publicly verifiable secret sharing [21,15], although neither of these constructions is composable. Universally composable randomness beacons based on verifiable random functions [18,3] can on the other hand be biased by adversaries.

Fair secure computation, where honest parties always obtain the output if the adversary learns it, is known to be impossible in the standard communication model and with dishonest majority [16], which in particular includes the two-party setting. Recently, Couteau et al. [17] presented a secure two-party protocol for the "best possible" alternative, meaning where an adversary can decide to withhold the output from an honest party but must make this decision independently of the actual output. Baum et al. [6] showed how to construct such a protocol with composition guarantees. Neither of these works considers the important multiparty setting.

An alternative, recently popularized idea is to circumvent the impossibility result of [16] by imposing financial fairness. There, cheating behavior is punished using cryptocurrencies and smart contracts and rational adversaries have a financial incentive to act fair. Works that achieve fair output delivery with penalties such as [1,8,22,5] always allow the adversary to make the abort decision *after* he sees the output. Therefore financial incentives must be chosen according to the worst-case gain of an adversary. By forcing the adversary to make the decision before seeing the output, he has to make it based on an expected gain instead.

1.3 Our Techniques

Multiparty Communication. We define a simultaneous broadcast with a delay that can be adjusted by the adversary. The functionality is a generalization of the Secure Message Transmission functionality of [6], although in the multi-receiver setting. As a second definition for multiparty communication we define a composable public ledger in ACT. We adapt the ledger functionality of Badertscher et al. [4] to get a ticked public ledger functionality that do not need access to a global clock functionality. Instead of the functionality internally keeping track of diverse relevant times and interacting with the global clock in order to take the necessary actions, it only keeps track of a few counters that are decreased when a tick happens and that directly trigger the necessary actions when they get to zero.

Publicly Verifiable TLPs. We define the notion of publicly verifiable TLPs, which allow for a prover who performs all the computational steps needed to solve a TLP to further convince any verifier that the TLP solution it obtained is valid, while requiring the verifier to perform a constant number of computational steps. This public verifiability property turns out to have interesting applications to constructing and improving the efficiency of randomness beacons and MPC with output-independent abort. We show that this notion can be realized by the TLP construction of [6], since it has tags that encode both the initial and final computational states of the TLP as well as a trapdoor that can be used to solve the TLP in constant time. While these values are normally not revealed by the tags, a party who solves the TLP can retrieve these values in order to verify its own solution. We observe that this verification procedure can be consistently repeated by any verifier who receives the information contained in the tags from the party who solved the TLP.

Composable Randomness Beacon. We realize a guaranteed output delivery (G.O.D.) coin tossing functionality that works like a randomness beacon from publicly verifiable TLPs and delayed multiparty communication (through either a delayed broadcast or a public ledger). The idea builds on the standard commit-then-reveal coin tossing protocol but substitutes the commitments with TLPs: 1.

each party broadcasts (or posts on the public ledger) a TLP containing a random value as input that can be solved in δ ticks (*i.e.* computational steps), 2. after δ ticks (counted by solving its own TLP), each party reveals the publicly verifiable solution to its TLP (showing its random input) and stops considering new TLPs received after this point, 3. if any party fails to reveal the valid solution to its own TLP, the other parties can solve it by themselves and retrieve that party's random input, 4. the output is obtained by XORing all random inputs from the valid TLPs. An adversary cannot make this protocol execution abort because any party can solve all valid TLPs. Moreover, by choosing δ larger than the maximum delay of the broadcast channel (or the maximum delay to get into the stable state of the public ledger that was already received by all honest parties), we guarantee that the adversary cannot bias the output, since it cannot have solved the other parties' TLPs before it has to send its own (which is not considered in the protocol if it is received after the other parties have solved their TLPs). This construction is particularly interesting in a setting with financial incentives as proposed by the popular Ethereum-based biased randomness beacon RANDAO [24]. In this scenario, our protocol can achieve very good efficiency if adversaries who fail to reveal valid solutions to their own TLPs are financially punished for causing the extra work needed from the other parties to solve their TLPs (and perhaps the punishment can be distributed among the parties who perform the extra work to solve TLPs that were not revealed by their owners).

MPC with (Punishable) Output-Independent Abort. In our work, we want to achieve that parties agree on the set of cheaters in case cheating occurs. Moreover, we want to ensure that an adversary does not learn the output of a secure computation before deciding that it will cause an abort. This property, called output-independent abort, allows honest parties to agree on which parties they may exclude from future secure computations. While this property is trivial in the two-party setting [6], it becomes much more complex with multiple parties. We observe that the use of a synchronous primitive such as synchronous broadcast seems crucial and use it to generalize the work of [6]. At the same time, we are able to optimize their consistency check by using weaker, non-homomorphic commitments in the process which also dramatically reduces the protocol complexity. We then achieve punishable output-independent abort by using publicly verifiable primitives and a smart contract in a way similar to [5]. While the modifications may appear to be straightforward, care must be taken in the proof process as no ideal functionalities have previously been designed for such composable timed primitives.

2 Preliminaries

Throughout this work, we will use λ for the statistical and τ for the computational security parameter.

2.1 Universal Composability

We use the (Global) Universal Composability or (G)UC model [13,14] for analyzing security and refer interested readers to the original works for more details.

In UC protocols are run by interactive Turing Machines (iTMs) called *parties*. A protocol π will have n parties which we denote as $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$. The *adversary* $\mathcal{A}$, which also is an iTM, will be able to corrupt a subset $I \subset \mathcal{P}$ as defined by the security model. The parties will be able to exchange messages via resources, called *ideal functionalities* (which themselves are iTMs) and which are denoted by $\mathcal{F}$.

As usual, we define security with respect to an iTM $\mathcal{Z}$ called *environment*. The environment provides inputs to and receives outputs from the parties $\mathcal{P}$. Furthermore, the adversary $\mathcal{A}$ will control the corrupt parties I in the name of $\mathcal{Z}$ and thus gain control over these parties. To define security, let $\pi^{\mathcal{F}_1,\cdots} \circ \mathcal{A}$ be the distribution of the output of an arbitrary $\mathcal{Z}$ when interacting with $\mathcal{A}$ in a real protocol instance π using resources $\mathcal{F}_1, \ldots$. Furthermore, let $\mathcal{S}$ denote an *ideal world adversary* and $\mathcal{F} \circ \mathcal{S}$ be the distribution of the output of $\mathcal{Z}$ when interacting with parties which run with $\mathcal{F}$ instead of π and where $\mathcal{S}$ takes care of adversarial behavior.

Definition 1. *We say that $\mathcal{F}$ UC-securely implements π if for every iTM $\mathcal{A}$ there exists an iTM $\mathcal{S}$ (with black-box access to $\mathcal{A}$) such that no environment $\mathcal{Z}$ can distinguish $\pi^{\mathcal{F}_1,\cdots} \circ \mathcal{A}$ from $\mathcal{F} \circ \mathcal{S}$ with non-negligible probability.*

In the security experiment $\mathcal{Z}$ may arbitrarily activate parties or $\mathcal{A}$, though *only one iTM (including $\mathcal{Z}$) is active at each point of time.*

2.2 Public Verifiability

Several functionalities in this work provide *public verifiability*. To model this, we follow the approach of Badertscher et al. [4] and allow the set of verifiers $\mathcal{V}$ to be dynamic by adding register and de-register instructions as well as instructions that allow $\mathcal{S}$ to obtain the list of registered verifiers. All functionalities with public verifiability include the following interfaces (which are omitted henceforth for simplicity):

Register: Upon receiving $(\mathsf{Register}, \mathsf{sid})$ from some verifier $\mathcal{V}_i$, set $\mathcal{V} = \mathcal{V} \cup \mathcal{V}_i$ and return $(\mathsf{Registered}, \mathsf{sid}, \mathcal{V}_i)$ to $\mathcal{V}_i$.

Deregister: Upon receiving $(\mathsf{Deregister}, \mathsf{sid})$ from some verifier $\mathcal{V}_i$, set $\mathcal{V} = \mathcal{V} \backslash \mathcal{V}_i$ and return $(\mathsf{Deregistered}, \mathsf{sid}, \mathcal{V}_i)$ to $\mathcal{V}_i$.

Is Registered: Upon receiving $(\mathsf{Is-Registered}, \mathsf{sid})$ from $\mathcal{V}_i$, return $(\mathsf{Is-Registered}, \mathsf{sid}, b)$ to $\mathcal{V}_i$, where $b = 1$ if $\mathcal{V}_i \in \mathcal{V}$ and $b = 0$ otherwise.

Get Registered: Upon receiving $(\mathsf{Get-Registered}, \mathsf{sid})$ from the ideal adversary $\mathcal{S}$, the functionality returns $(\mathsf{Get-Registered}, \mathsf{sid}, \mathcal{V})$ to $\mathcal{S}$.

The above instructions can also be used by other functionalities to register as a verifier of a publicly verifiable functionality, which means that the verified functionality must be global.

2.3 Abstract Composable Time

ACT expresses abstract time within the GUC framework in such a way that protocols can be made oblivious to clock ticks. To achieve this, ACT provides a global ticker functionality $\mathcal{G}_{\mathsf{ticker}}$ as depicted in Fig. 1. This global ticker can provide "ticks" to ideal functionalities in the name of the environment. A tick represents a discrete unit of time which can only be advanced, and moreover only one unit at a time. "Ticked" ideal functionalities have a ticking interface through which they are contacted by $\mathcal{G}_{\mathsf{ticker}}$. Parties may observe events triggered by elapsed time, but not the time itself. Ticked functionalities can freely interpret these ticks and perform arbitrary internal state changes.

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

Initialize a set of registered parties $\mathcal{P} = \emptyset$, a set of registered functionalities $F = \emptyset$, a set of activated parties $L_P = \emptyset$, and a set of functionalities $L_F = \emptyset$ that have been informed about the current tick.

Party registration: Upon receiving (`register`, pid) from honest party P with pid pid, add pid to $\mathcal{P}$ and send (`registered`) to P.
Functionality registration: Upon receiving (`register`) from functionality $\mathcal{F}$, add $\mathcal{F}$ to F and send (`registered`) to $\mathcal{F}$.
Tick: Upon receiving (`tick`) from the environment, do the following:
1. If $\mathcal{P} = L_P$, reset $L_P = \emptyset$ and $L_F = \emptyset$, and send (`ticked`) to the adversary $\mathcal{S}$.
2. Else, send (`notticked`) to the environment.

Ticked request: Upon receiving (`ticked`?) from functionality $\mathcal{F} \in F$, do the following:
- If $\mathcal{F} \notin L_F$, add $\mathcal{F}$ to L_F and send (`ticked`) to $\mathcal{F}$.
- If $\mathcal{F} \in L_F$, send (`notticked`) to $\mathcal{F}$.

Record party activation: Upon receiving (`activated`) from party P with pid $\mathsf{pid} \in \mathcal{P}$, add pid to L_P and send (`recorded`) to P.

Fig. 1. Global ticker functionality $\mathcal{G}_{\mathsf{ticker}}$(from [6]).

To ensure that all honest parties have a chance of observing all relevant timing-related events, $\mathcal{G}_{\mathsf{ticker}}$ only progresses if all honest parties have signaled to it that they have been activated (in arbitrary order). To further control the observable side effects of ticks, protocols and ideal functionalities are restricted to interact in the "pull model", precluding functionalities from implicitly providing communication channels between parties: parties are required to actively query functionalities in order to obtain new messages, and they obtain the activation token back upon completion. During each "tick" from the environment, a functionality may perform arbitrary computations or contact the adversary, but we restrict ourselves to adversaries which return the activation token to these functionalities afterwards and require that ideal functionalities after each tick return the token to $\mathcal{G}_{\mathsf{ticker}}$. We refer to [6] for a more detailed description.

How we use ACT. In comparison to [6] we require that the adversary per "tick" gets activated by each functionality $\mathcal{F}$ and can provide an optional $(\mathsf{Schedule}, \mathsf{sid}, \mathcal{D})$ message parameterized by a queue $\mathcal{D}$. This queue contains commands to $\mathcal{F}$ which specify if the adversary wants to abort $\mathcal{F}$ or how it will schedule message delivery to individual parties in $\mathcal{P}$.

The reason for this requirement is twofold: first, it simplifies the specification of $\mathcal{F}$. Second and more importantly, any adversary that could influence message delivery between two "ticks" might interfere with simultaneous message delivery that is necessary for some of our solutions. The latter might be true if an adversary would be allowed to reschedule messages at any point.

As mentioned above, an adversary *does not have* to send $(\mathsf{Schedule}, \mathsf{sid}, \mathcal{D})$ - each $\mathcal{F}$ can take care of guaranteed delivery itself. On the other hand, $\mathcal{D}$ *can depend on information that the adversary learns upon* $(\mathsf{Tick}, \mathsf{sid})$.

Ticked Functionalities. We will explicitly mention when a functionality $\mathcal{F}$ is "ticked" - which means that it reacts to the $(\mathsf{Tick}, \mathsf{sid})$ command from $\mathcal{G}_{\mathsf{ticker}}$. Each such $\mathcal{F}$ internally has two lists $\mathcal{M}, \mathcal{Q}$ which are initially empty. The functionality will use these to store messages that the parties ought to obtain.

$\mathcal{Q}$ contains those messages to parties that are currently buffered. Actions by honest parties can add new messages to $\mathcal{Q}$, while actions of the adversary can change the content of $\mathcal{Q}$ in certain restricted ways or move messages from $\mathcal{Q}$ to $\mathcal{M}$. In comparison, $\mathcal{M}$ contains all the "output-ready" messages that can be read by the parties directly. On the other hand, the content of $\mathcal{M}$ cannot be changed by $\mathcal{A}$ and he cannot prevent parties from reading it. "Messages" from $\mathcal{F}$ is here meant in a general sense - these must not only be messages that have been sent between parties, but could also be delayed responses from $\mathcal{F}$ to a request from an honest party.

We assume that each ticked functionality $\mathcal{F}$ has two interfaces. One, as mentioned above, is called **Tick** and should respond as outlined before. The second is called **Fetch Messages**. This latter interface allows parties to obtain entries of $\mathcal{M}$. The code for this is actually identical across all ticked functionalities, so we specify it here for conciseness:

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

Macros. A recurring pattern in ticked ideal functionalities in [6] is that the functionality $\mathcal{F}$, upon receiving a request $(\mathsf{Request}, \mathsf{sid}, m)$ by party $\mathcal{P}_i$ must first internally generate unique message IDs mid to balance message delivery with the adversarial option to delay messages. $\mathcal{F}$ then internally stores the message to be delivered together with the mid in a list $\mathcal{Q}$ and finally hands out i, mid to the ideal adversary $\mathcal{S}$ as well as potentially also m. This will later allow $\mathcal{S}$ to influence message delivery of m by $\mathcal{F}$ at will. We will now define macros that simplify the aforementioned process. When using the macros we will sometimes leave out certain options if their choice is clear from the context.

Macro "*Notify the parties $T \subseteq \mathcal{P}$ about a message with prefix* Request *from $\mathcal{P}_i$ via $\mathcal{Q}$ with delay Δ*" expands to

1. Let $T = \{\mathcal{P}_{i_1}, \dots, \mathcal{P}_{i_k}\}$. Sample unused message IDs $\mathsf{mid}_{i_1}, \dots, \mathsf{mid}_{i_k}$.
2. Add $(\Delta, \mathsf{mid}_{i_j}, \mathsf{sid}, \mathcal{P}_{i_j}, (\mathsf{Request}, i))$ to $\mathcal{Q}$ for each $\mathcal{P}_{i_j} \in T$.

Macro "*Send message m with prefix* Request *received from party $\mathcal{P}_i$ to the parties $T \subseteq \mathcal{P}$ via $\mathcal{Q}$ with delay Δ*" expands to

1. Let $T = \{\mathcal{P}_{i_1}, \dots, \mathcal{P}_{i_k}\}$. Sample unused message IDs $\mathsf{mid}_{i_1}, \dots, \mathsf{mid}_{i_k}$.
2. Add $(\Delta, \mathsf{mid}_{i_j}, \mathsf{sid}, \mathcal{P}_{i_j}, (\mathsf{Request}, i, m))$ to $\mathcal{Q}$ for each $\mathcal{P}_{i_j} \in T$.

Macro "*Notify $\mathcal{S}$ about a message with prefix* Request" expands to

– Send $(\mathsf{Request}, \mathsf{sid}, i, , \mathsf{mid}_{i_1}, \dots, \mathsf{mid}_{i_k})$ to $\mathcal{S}$.

Macro "*Send m with prefix* Request *and the IDs to $\mathcal{S}$*" expands to

– Send $(\mathsf{Request}, \mathsf{sid}, i, m, , \mathsf{mid}_{i_1}, \dots, \mathsf{mid}_{i_k})$ to $\mathcal{S}$.

These macros are useful whenever honest parties send messages that can arrive at different times at the recipients. If they send these via simultaneous broadcast (ensuring simultaneous arrival), then we will instead only choose *one* message ID for all messages. As the adversary can later influence delivery on message ID-basis, this ensures simultaneous delivery. We indicate the use of synchronized broadcast by using the prefix "simultaneously" in the first two macros.

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

$\mathcal{F}^{\Delta}_{\mathsf{BC,delay}}$ is parameterized by maximal delay $\Delta > 0$, parties $\mathcal{P} = \{\mathcal{P}_1, \dots, \mathcal{P}_n\}$ with a special sender $\mathcal{P}_S \in \mathcal{P}$ and adversary $\mathcal{S}$. $\mathcal{S}$ may corrupt a strict subset $I \subset \mathcal{P}$.

Send: Upon receiving an input $(\mathsf{Send}, \mathsf{sid}, m)$ from party $\mathcal{P}_S$:
1. Simultaneously send message m to the parties $\mathcal{P}$ via $\mathcal{Q}$ with delay Δ.
2. Send m and mid to $\mathcal{S}$.

Tick:
1. Remove each $(0, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$ from $\mathcal{Q}$ and instead add $(\mathcal{P}_i, \mathsf{sid}, m)$ to $\mathcal{M}$.
2. Replace each $(\mathtt{cnt}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$ in $\mathcal{Q}$ with $(\mathtt{cnt} - 1, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$.

Upon receiving $(\mathsf{Schedule}, \mathsf{sid}, \mathcal{D})$ from $\mathcal{S}$:
- If $(\mathsf{Deliver}, \mathsf{sid}, \mathsf{mid}) \in \mathcal{D}$ then remove each $(\mathtt{cnt}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$ from $\mathcal{Q}$ and add $(\mathcal{P}_i, \mathsf{sid}, m)$ to $\mathcal{M}$.
- If $(\mathsf{Abort}, \mathsf{sid}) \in \mathcal{D}$ then add $(\mathcal{P}_i, \mathsf{sid}, \mathsf{Abort})$ to $\mathcal{M}$ for each $i \in [n]$ and ignore all further messages with this sid except to **Fetch Message**.

Fig. 2. Ticked ideal functionality $\mathcal{F}^{\Delta}_{\mathsf{BC,delay}}$ for authenticated broadcast with maximal message delay Δ.

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

$\mathcal{F}_{\mathsf{Ledger}}$ is parameterized by the algorithms $\mathsf{Validate}, \mathsf{ExtendPolicy}$ and the parameters $\mathsf{slackWindow}, \mathsf{qualityWindow}, \mathsf{delaySync}, \mathsf{maxTXDelay}, \mathsf{maxEmpty} \in \mathbb{N}$. It manages variables $\mathsf{state}, \mathsf{nextBlock}, \mathsf{buffer}, \mathsf{emptyBlocks}$, which are initially set to $\perp, \perp$, $\emptyset$, and $\mathsf{maxEmpty}$ respectively. The functionality maintains a list $\mathsf{recentQuality}$ that keeps track of the quality (i.e., generated using the honest procedures or not) of the last $\mathsf{qualityWindow}$ blocks proposed by $\mathcal{S}$ that were used to extend the state state. The functionality maintains the set of registered parties $\mathcal{P}$, and the subsets of synchronized honest parties $\mathcal{H}$ and of de-synchronized honest parties $\mathcal{D}$. Each party $\mathcal{P}_i$ has a current-state view state_i that is initially set to $\perp$. Whenever an honest party $\mathcal{P}_i$ is registered during the execution, it is added to the subset $\mathcal{D}$, an entry $(\mathcal{P}_i, \mathsf{delaySync})$ is added to the delayed entry table DE and the de-synchronized state state'_i is set to $\perp$.

Tick: 1. For each entry $(\mathcal{P}_i, \mathtt{cnt}) \in \mathsf{DE}$, if $\mathtt{cnt} = 1$, set $\mathcal{H} \leftarrow \mathcal{H} \cup \mathcal{P}_i, \mathcal{D} \leftarrow \mathcal{D} \setminus \mathcal{P}_i$ and remove $(\mathcal{P}_i, \mathtt{cnt})$ from DE; otherwise decrease the counter value $\mathtt{cnt}$ by 1.

2. For each entry transaction $\mathsf{BTX} = (\mathsf{tx}, \mathsf{txid}, \mathcal{P}_i, \mathtt{cnt}) \in \mathsf{buffer}$, decrease the counter value $\mathtt{cnt}$ by 1. Remove from buffer all transaction with counter value equal 0, and create a list $\mathsf{mandatoryInclusion}$ with them.
3. Set $\mathsf{state} \leftarrow \mathsf{ExtendPolicy}(\mathsf{state}, \mathsf{nextBlock}, \mathsf{buffer}, \mathsf{mandatoryInclusion}, \mathsf{recentQuality}, \mathsf{emptyBlocks})$. If $\mathsf{nextBlock} = (\mathsf{hFlag}, \mathsf{listTX})$ was used to extend state, then update the list $\mathsf{recentQuality}$ using hFlag. If a block was added to state, set $\mathsf{emptyBlocks} \leftarrow \mathsf{maxEmpty}$; otherwise decrease $\mathsf{emptyBlocks}$ by 1.
4. Remove from buffer all transactions that were added into state. Set $\mathsf{nextBlock} \leftarrow \perp$. For each entry transaction $\mathsf{BTX} \in \mathsf{buffer}$, if $\mathsf{Validate}(\mathsf{BTX}, \mathsf{state}, \mathsf{buffer}) = 0$, then remove BTX from buffer.

Read: Upon receiving $(\mathsf{Read}, \mathsf{sid})$ from $\mathcal{P}_i \in \mathcal{P}$: if $\mathcal{P}_i \in \mathcal{D}$, return $(\mathsf{Read}, \mathsf{sid}, \mathsf{state}'_i)$; otherwise return $(\mathsf{Read}, \mathsf{sid}, \mathsf{state}_i)$.

Read Buffer: Upon receiving $(\mathsf{ReadBuffer}, \mathsf{sid})$ from $\mathcal{S}$, return $(\mathsf{ReadBuffer}, \mathsf{sid}, \mathsf{buffer})$.

Submit a Transaction: Upon receiving $(\mathsf{Submit}, \mathsf{sid}, \mathsf{tx})$ from $\mathcal{P}_i$, choose a unique transaction ID txid and set $\mathsf{BTX} \leftarrow (\mathsf{tx}, \mathsf{txid}, \mathcal{P}_i, \mathsf{maxTXDelay})$. If $\mathsf{Validate}(\mathsf{BTX}, \mathsf{state}, \mathsf{buffer}) = 1$, then set $\mathsf{buffer} \leftarrow \mathsf{buffer} \cup \{\mathsf{BTX}\}$. Send $(\mathsf{Submit}, \mathsf{sid}, \mathsf{BTX})$ to $\mathcal{S}$.

Propose a Block: Upon receiving $(\mathsf{Propose}, \mathsf{sid}, \mathsf{hFlag}, (\mathsf{txid}_1, \ldots, \mathsf{txid}_\ell))$ from $\mathcal{S}$, create the list of transactions listTX by concatenating the eventual transactions contained in buffer that have transaction IDs $\mathsf{txid}_1, \ldots, \mathsf{txid}_\ell$. Then set $\mathsf{nextBlock} \leftarrow (\mathsf{hFlag}, \mathsf{listTX})$ and return $(\mathsf{Propose}, \mathsf{sid}, ok)$ to $\mathcal{S}$.

Set State-Slackness: Upon receiving $(\mathsf{SetSlack}, \mathsf{sid}, \mathcal{P}_i, t)$ from $\mathcal{S}$, proceed as follows: if $t \geq |\mathsf{state}| - \mathsf{slackWindow}$ and $t > |\mathsf{state}_i|$, then set state_i to contain the first t blocks of state and return $(\mathsf{SetSlack}, \mathsf{sid}, ok)$; otherwise, set $\mathsf{state}_i \leftarrow \mathsf{state}$ and return $(\mathsf{SetSlack}, \mathsf{sid}, fail)$.

Set State of De-synchronized Parties: Upon receiving $(\mathsf{DeSyncState}, \mathsf{sid}, \mathcal{P}_i, s)$ from $\mathcal{S}$ for $\mathcal{P}_i \in \mathcal{D}$, set $\mathsf{state}'_i \leftarrow s$ and return $(\mathsf{DeSyncState}, \mathsf{sid}, ok)$.

Fig. 3. Ledger Functionality $\mathcal{F}_{\mathsf{Ledger}}$.

3 Multi-Party Message Delivery

In this section we will model two common multi-party messaging primitives in the ACT framework, namely authenticated broadcast and public ledgers.

3.1 Ticked Authenticated Broadcast

In Fig. 2 we describe an authenticated broadcast functionality with abstract ticks and maximal delay Δ. The functionality uses a scheduling-based template similar to [6] as this seems much easier to handle than an extension of $\mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$.

Observe that this functionality delivers the broadcast message to all honest parties in the same "tick round" (including the sender). This will be necessary later as a point of synchronization between honest parties to allow them to make identical decisions within protocols that use $\mathcal{F}^{\Delta}_{\mathsf{BC,delay}}$. This property is modeled as the functionality uses the same ID mid for all parties and will permit $\mathcal{S}$ only to schedule earlier delivery than Δ simultaneously for all queued messages with this mid.

A weaker formulation of this broadcast can e.g. model that all honest parties obtain the message within a certain maximal number of ticks Δ, but where the precise round at which each individual party obtains it can be decided by $\mathcal{S}$.

3.2 Ticked Public Ledger

In order to define a ledger functionality $\mathcal{F}_{\mathsf{Ledger}}$, we adapt ideas from Badertscher et al. [4]. The ledger functionality $\mathcal{F}_{\mathsf{Ledger}}$ is presented in Fig. 3. It is parameterized by the algorithms $\mathsf{Validate}, \mathsf{ExtendPolicy}$ and the parameters $\mathsf{slackWindow}$, $\mathsf{qualityWindow}, \mathsf{delaySync}, \mathsf{maxTXDelay}, \mathsf{maxEmpty} \in \mathbb{N}$. These parameters can depend on the protocol used to realize the ledger. At any point the ledger has a stable state, which is eventually received by all honest parties (but there is no guarantee that they will receive it immediately, or even at the same time). The parameter $\mathsf{slackWindow}$ is an upper bound on the number of the most recent blocks in the current stable state that are still not received by all honest parties.

Any party can submit a transaction, which will be added to the buffer if it is valid. $\mathsf{Validate}$ is used to validate the transactions, and should at least guarantee that no transaction waiting in the buffer contradicts the stable state of the ledger (the validity of the transactions waiting in the buffer needs to be tested again once a new block is added to the stable state). The adversary is responsible for proposing the potential next blocks. It can choose such blocks using the procedures of an honest miner or not, but the functionality keeps track of that. It can also propose to have no new block in the next tick. Whenever the functionality is ticked, it runs the algorithm $\mathsf{ExtendPolicy}$ to decide if a block will be added, and what its content would be. $\mathsf{ExtendPolicy}$ normally accepts the block proposed by the adversary, but it also enforces liveness and chain quality properties. $\mathsf{maxTXDelay}$ defines the maximum number of ticks that a valid transaction will stay in the buffer. After $\mathsf{maxTXDelay}$ ticks without inclusion, $\mathsf{ExtendPolicy}$ will force the inclusion of the valid transaction in the next block. $\mathsf{maxEmpty}$

defines the maximum number of consecutive suggestions of not adding a new block by the adversary will be accepted by $\mathsf{ExtendPolicy}$. After that many ticks without adding a new block, a new block insertion is forced. $\mathsf{ExtendPolicy}$ also analyzes how many of the last $\mathsf{qualityWindow}$ blocks were honestly generated, and force an honest behavior if the number of honest blocks do not meet the chain quality properties.

Note that a good simulator acts in such way that it never forces an action from $\mathsf{ExtendPolicy}$, as a forced action may lead to a distinguishing advantage for the environment. As the set of parties registered in the ledger is dynamic, the ledger functionality $\mathcal{F}_{\mathsf{Ledger}}$ includes registration interfaces similar to those for public verifiers described in Section 2, and these are omitted for conciseness. $\mathsf{delaySync}$ defines how long it takes for honest parties that just joined to become synchronized (until that point, the adversary can arbitrarily set the state that the de-synchronized parties view).

In the ledger functionality of Badertscher et al. [4] keeps track of many relevant times and interacts with a global clock in order to take actions at the appropriate time. Our ledger functionality, on the other hand, only keeps track of a few counters. The counters are updated during the ticks, and the appropriate actions are done if they reach zero. However, our algorithm $\mathsf{ExtendPolicy}$ also enforces liveness and chain quality properties, and our ledger functionality can also be realized by the same protocols as in [4].

4 Publicly Verifiable Time-Lock Puzzle

In this section, we extend the UC treatment of time-lock puzzles from [6] to the case of publicly verifiable TLPs. Intuitively, a publicly verifiable TLP allows a party who solves a TLP by performing a certain number of computational steps to later convince a third party that the solution is correct while only requiring a constant amount of work from the third party. First, we introduce an ideal functionality $\mathcal{F}_{\mathsf{tlp}}$ modeling a publicly verifiable TLP by extending the functionality presented in [6]. Functionality $\mathcal{F}_{\mathsf{tlp}}$ is presented in Figure 4 and has an extra interface for any third party verifier to check whether a certain solution to a given TLP is correct.

4.1 Construction

We show that the protocol proposed in [6] can be augmented to realize the new public verification interface. In the original protocol, a global random oracle $\mathcal{G}_{\mathsf{rpoRO}}$ and an ideal functionality $\mathcal{F}_{\mathsf{rsw}}$ that captures the hardness assumption used by Rivest et al. [25] are used as setup (we refer the reader to Supplementary Material A for full descriptions). The main observation is that a puzzle solution $\mathtt{el}, m$ for a puzzle $\mathtt{puz}$ can be verified to be valid by repeating the steps of the **Get Message** interface with $\mathtt{puz}, \mathtt{el}$ as input and checking that the output obtained is equal to m. The reason this procedure works is that each puzzle $\mathtt{puz} = (\mathtt{el}_0, \Gamma, \mathtt{tag})$ encodes in its $\mathtt{tag}$ both the final state $\mathtt{el}_\Gamma$ obtained after

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

$\mathcal{F}_{\mathsf{tlp}}$ is parameterized by a set of parties $\mathcal{P}$, a set of verifiers $\mathcal{V}$, 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 $\mathtt{tag} \overset{\$}{\leftarrow} \mathcal{TAG}$ and $\Gamma + 1$ random distinct states $\mathtt{st}_j \overset{\$}{\leftarrow} \{0,1\}^\tau$ for $j \in \{0, \dots, \Gamma\}$. 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 $j \in \{0, \dots, \Gamma - 1\}$, output $(\mathsf{CreatedPuzzle}, \mathsf{sid}, \mathsf{puz} = (\mathtt{st}_0, \Gamma, \mathtt{tag}), \mathtt{st}_\Gamma)$ to $\mathcal{P}_o$ and $(\mathsf{CreatedPuzzle}, \mathsf{sid}, \mathsf{puz} = (\mathtt{st}_0, \Gamma, \mathtt{tag}))$ to $\mathcal{S}$. $\mathcal{F}_{\mathsf{tlp}}$ 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 $\mathtt{st}' \overset{\$}{\leftarrow} \mathcal{ST}$.
- 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 $\mathcal{Q}$.

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 $\mathcal{P}_i$.

Fetch State: Upon receiving $(\mathsf{Fetch}, \mathsf{sid})$ from $\mathcal{P}_i \in \mathcal{P}$ retrieve the set L_i of all entries $(\mathcal{P}_i, \mathsf{sid}, \cdot, \cdot)$ in $\mathcal{M}$, remove L_i from $\mathcal{M}$ and send $(\mathsf{Fetch}, \mathsf{sid}, L_i)$ to $\mathcal{P}_i$.

Public Verification: Upon receiving $(\mathsf{Verify}, \mathsf{sid}, \mathsf{puz}, \mathtt{st}, m)$ from a party $\mathcal{V}_i \in \mathcal{V}$, parse $\mathsf{puz} = (\mathtt{st}_0, \Gamma, \mathtt{tag})$ and, if there exists $(\mathtt{st}_0, \mathtt{tag}, \mathtt{st}, m) \in \mathsf{omsg}$, set $b = 1$, else set $b = 0$. Output $(\mathsf{Verified}, \mathsf{sid}, \mathsf{puz}, \mathtt{st}, m, b)$ to $\mathcal{V}_i$.

Tick: Set $\mathcal{M} \leftarrow \mathcal{Q}$ and $\mathcal{Q} = \emptyset$.

Fig. 4. Functionality $\mathcal{F}_{\mathsf{tlp}}$ for publicly verifiable time-lock puzzles.

Γ computational steps as well as the trapdoor $\mathtt{td}$ that can be used to compute $\mathtt{el}_\Gamma$ from $\mathtt{el}_0$ in constant time. Given a candidate solution $\mathtt{el}, m$ for puz, the verifier can confirm that $\mathtt{tag}$ does encode $\mathtt{el}$ as $\mathtt{el}_\Gamma$ and recompute m all in constant time, since it also uses $\mathtt{el}$ to recover $\mathtt{td}$. The use of a global $\mathcal{F}_{\mathsf{rsw}}$ for the computation and of a global random oracle $\mathcal{G}_{\mathsf{rpoRO}}$ for generating $\mathtt{tag}$ guarantee that any verifier $\mathcal{V}_i \in \mathcal{V}$ obtains the same result as any party $\mathcal{P}_i \in \mathcal{P}$. Hence, if the verifier obtains a message $m' = m$ when executing the **Get Message** procedure on input $\mathtt{puz}, \mathtt{st}$ claimed to have an associated message m, we have a guarantee that all other parties executing the protocol will obtain the same message and that a verifier can check this message is valid with respect to $\mathtt{puz}$ in constant time. We describe Protocol π_{tlp} based on [6] in Figure 5 and formally state its security in Theorem 1.

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 a set of parties $\mathcal{P}$, an owner $\mathcal{P}_o \in \mathcal{P}$ and a set of verifiers $\mathcal{V}$ interacting among themselves and with functionalities $\mathcal{F}_{\mathsf{rsw}}$, $\mathcal{G}_{\mathsf{rpoRO1}}$ (an instance of $\mathcal{G}_{\mathsf{rpoRO}}$ with domain $\{0,1\}^{2\cdot\tau}$ and output size $\{0,1\}^{2\cdot\tau}$) and $\mathcal{G}_{\mathsf{rpoRO2}}$ (an instance of $\mathcal{G}_{\mathsf{rpoRO}}$ with domain $\{0,1\}^{3\cdot\tau}$ 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 $(\mathsf{CreatedPuzzle}, \mathsf{sid}, \mathtt{puz} = (\mathtt{el}_0, \Gamma, \mathtt{tag}), \mathtt{el}_{\Gamma})$.

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 $(\mathsf{GetMsg}, \mathsf{sid}, \mathtt{puz}, \mathtt{el})$ 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 $(\mathsf{GetMsg}, \mathsf{sid}, \mathtt{el}_0, \mathtt{tag}, \mathtt{el}, \perp)$.

Public Verification: On input $(\mathsf{Verify}, \mathsf{sid}, \mathtt{puz}, \mathtt{st}, m)$, a verifier $\mathcal{V}_i$ executes the steps of **Get Message** with input $(\mathsf{GetMsg}, \mathtt{puz}, \mathtt{el})$ in order to obtain $(\mathsf{GetMsg}, \mathsf{sid}, \mathtt{el}_0, \mathtt{tag}, \mathtt{el}, m')$. If $m = m'$, $\mathcal{V}_i$ sets $b = 1$, else it sets $b = 0$. Finally, $\mathcal{V}_i$ outputs $(\mathsf{Verified}, \mathsf{sid}, \mathtt{puz}, \mathtt{st}, m, b)$.

Output: Upon receiving $(\mathsf{Fetch}, \mathsf{sid})$ 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. 5. Protocol π_{tlp} realizing publicly verifiable time-lock puzzle functionality $\mathcal{F}_{\mathsf{tlp}}$ in the $\mathcal{F}_{\mathsf{rsw}}, \mathcal{G}_{\mathsf{rpoRO}}$-hybrid model.

Theorem 1. *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 $\mathcal{F}_{\mathsf{tlp}}$.*

Proof. In order to prove this theorem, we construct a simulator $\mathcal{S}$ that interacts with an internal copy $\mathcal{A}$ of the adversary forwarding messages between $\mathcal{A}$ and $\mathcal{G}_{\mathsf{rpoRO1}}, \mathcal{G}_{\mathsf{rpoRO2}}, \mathcal{F}_{\mathsf{rsw}}$ unless otherwise stated. It forwards messages from $\mathcal{A}$ and simulated hybrid functionalities to $\mathcal{G}_{\mathsf{ticker}}$. For any environment $\mathcal{Z}$, we show that an execution with $\mathcal{S}$ and $\mathcal{F}_{\mathsf{rsw}}$ is indistinguishable from an execution with $\mathcal{A}$ in the $\mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{rsw}}$-hybrid model. First, we observe that, apart from the **Public Verification** interface, $\mathcal{S}$ can be constructed exactly as the simulator in Theorem 2 of [6], which we reproduce in Supplementary Material B. In order to simulate the **Public Verification** procedure, $\mathcal{S}$ executes exactly the same steps of an honest verifier in π_{tlp} and, in case $\mathcal{A}$ performs public verification, forwards all messages between $\mathcal{A}$ and $\mathcal{G}_{\mathsf{rpoRO1}}, \mathcal{G}_{\mathsf{rpoRO2}}, \mathcal{F}_{\mathsf{rsw}}$. Notice that the **Public Verification** procedure is executed locally by a verifier who has received $\mathtt{puz}, \mathtt{el}, m$ and that it corresponds exactly to executing the steps of the **Get Message** procedure with input $\mathtt{puz}, \mathtt{el}$ and verifying the output is equal to m. Hence, since an execution of **Get Message** with $\mathcal{S}$ and $\mathcal{F}_{\mathsf{tlp}}$ is indistinguishable from an execution with $\mathcal{A}$ in the $\mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{rsw}}$-hybrid model as proven in [6], the output of the simulated **Public Verification** procedure with input $\mathtt{puz} = (\mathtt{el}_0, \Gamma, \mathtt{tag} = (\mathtt{tag}_1, \mathtt{tag}_2)), \mathtt{el}', m'$ where $\mathtt{puz}$ was generated with message m and trapdoor $\mathtt{td}$ will only differ from that of $\mathcal{F}_{\mathsf{tlp}}$ if $\mathcal{A}$ finds $\mathtt{el}', m' \neq m$ such that querying $\mathcal{G}_{\mathsf{rpoRO1}}$ with $(\textsc{Hash-Query}, (\mathtt{el}_0|\mathtt{el}'))$ yields $h_1 = \mathtt{tag}_1$ where $(m'|\mathtt{td}) = \mathtt{tag}_1 \oplus h_1$ and that querying $\mathcal{G}_{\mathsf{rpoRO2}}$ with $(\textsc{Hash-Query}, (h_1|m'|\mathtt{td}))$ yields $h_2 = \mathtt{tag}_2$, which only happens with negligible probability in τ since $\mathcal{A}$ can only make a polynomial number of queries to $\mathcal{G}_{\mathsf{rpoRO1}}, \mathcal{G}_{\mathsf{rpoRO2}}$. □

5 Composable Randomness Beacons

We model a randomness beacon as a publicly verifiable coin tossing functionality $\mathcal{F}_{\mathsf{RB}}$ presented in Figure 6. Even though this functionality does not periodically produce new random values on its own as in some notions of randomness beacons, it can be periodically queried by the honest parties when they need to obtain new randomness.

5.1 Randomness Beacons from TLPs

In order to construct a UC-secure randomness beacon from TLPs and a broadcast channel or public ledger, we depart from a simple commit-then-open protocol for n parties realizing a simple coin tossing functionality with all messages posted to the public ledger or broadcast channel. Such a protocol involves each

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

$\mathcal{F}_{\mathsf{RB}}$ is parameterized by delay $\varDelta_{\mathsf{TLP-RB}}$ and interacts with a set of parties $\mathcal{P} = \{\mathcal{P}_1, \dots, \mathcal{P}_n\}$, a set of verifiers $\mathcal{V}$ and an adversary $\mathcal{S}$ through the following interfaces:

Toss: Upon receiving $(\textsc{Toss}, \mathsf{sid})$ from all honest parties in $\mathcal{P}$, sample $x \xleftarrow{\$} \{0,1\}^{\tau}$ and send $(\textsc{Tossed}, \mathsf{sid}, x)$ to all parties in $\mathcal{P}$ and $\mathcal{V}$ via $\mathcal{Q}$ with delay $\varDelta_{\mathsf{TLP-RB}}$.

Verify: Upon receiving $(\textsc{Verify}, \mathsf{sid}, x)$ from $\mathcal{V}_j \in \mathcal{V}$, if $(\textsc{Tossed}, \mathsf{sid}, x)$ has been sent to all parties in $\mathcal{P}$ set $f = 1$, otherwise, set $f = 0$. Output $(\textsc{Verified}, \mathsf{sid}, x, f)$ to $\mathcal{V}_j$.

Tick:

1. Remove each $(0, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$ from $\mathcal{Q}$ and instead add $(\mathcal{P}_i, \mathsf{sid}, m)$ to $\mathcal{M}$.
2. Replace each $(\mathtt{cnt}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$ in $\mathcal{Q}$ with $(\mathtt{cnt} - 1, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$.

Fig. 6. Functionality $\mathcal{F}_{\mathsf{RB}}$ for G.O.D. Publicly Verifiable Coin Tossing.

party $\mathcal{P}_i$ posting a commitment com_i containing a random value $r_i \xleftarrow{\$} \{0,1\}^{\tau}$ to the public ledger, waiting for enough time so that all $\mathsf{com}_1, \dots, \mathsf{com}_n$ can be posted on the ledger and then opening com_i once all $\mathsf{com}_1, \dots, \mathsf{com}_n$ have appeared on the ledger. The output of the protocol is defined as $r = r_1 \oplus \dots \oplus r_n$. However, an adversary can bias the output by aborting the opening.

The issue of adversarial bias in coin flipping protocols has been addressed by relying on publicly verifiable secret sharing schemes (PVSS) [15] with $O(n^2)$ communication complexity for achieving guaranteed output delivery. Recently, it has been suggested that a computational vs. communication complexity tradeoff can be achieved by leveraging Verifiable Delay Functions [10] (VDFs) and avoiding the commit-then-open template. This solution achieves $O(n)$ communication complexity at the cost of the high computational complexity incurred by computing VDFs, which are constructed from sequential computation similarly to TLPs. However, these constructions are proposed as folklore and there is no evidence that they can achieve any composability guarantees nor other standard notions of security.

We observe that the commit-then-open template for randomness beacons can be still salvaged from bias without resorting to PVSS by substituting the commitments in simple commit-then-open coin tossing with TLPs as captured in $\mathcal{F}_{\mathsf{tlp}}$. To see why the adversary cannot bias the output of the protocol, first notice that after $t+1$ TLPs are posted to the ledger (where t is the number of corrupted parties), at least one is generated by an honest party and all parties can solve the TLPs. Moreover, if the time to solve the TLP δ is large enough that TLPs of the adversary must be broadcast (or posted to the ledger) before the first honest TLP opens, the adversary does not get any information about honest parties' random values r_i, so it cannot choose its own random values in any way that biases the output. The resulting protocol also achieves $O(n)$ communication complexity while sacrificing computational complexity in the worst case (where TLPs must be solved). Using our UC-secure TLPs and a public ledger [4] or broadcast functionality with ticks (which we assume to be authenticated by *e.g.* digital signatures), it is possible to construct a simulator for this protocol and prove it UC-secure.

An interesting optimization of this construction can be obtained by observing that Protocol π_{tlp} is intrinsically publicly verifiable, allowing any third party to verify a solution to a given TLP in constant time. Notice that a solution $(\mathtt{el}, m)$ for a puzzle $\mathtt{puz}$ generated by $\mathcal{F}_{\mathsf{tlp}}$ can be publicly verified by any third party. Leveraging this property, we can optimize the simple randomness beacon construction described above by having honest parties post solutions to their own TLPs after the commitment phase is completed. In this case, if all parties who act in the commitment phase are honest, the protocol terminates faster without the need for all parties to solve each other's TLPs. Otherwise, the honest parties only have to solve the TLPs provided by corrupted parties (who do not post a valid solution to these TLPs after the commitment phase).

This construction is particularly interesting in a setting with financial incentives as proposed by the popular Ethereum-based biased randomness beacon RANDAO [24]. The core idea behind RANDAO is to leverage a smart contract that collects a security deposit from all parties who participate in a protocol execution before it starts. If corrupted parties misbehave, the smart contract redistributes their security deposits among the parties who successfully completed the protocol. The rationale of this approach is that corrupted parties have no financial incentive to introduce bias to the final output by selectively aborting their execution. However, it is always possible for corrupted parties to bias the beacon's output if they are willing to forfeit their security deposits. Applying a similar approach to our optimized randomness beacon protocol yields a beacon that cannot be biased even by an adversary willing to pay the price of forfeiting security deposits. In our case, parties would be required to provide a security deposit in order to participate in a protocol execution and would forfeit this deposit if they fail to send a valid TLP or to provide a valid solution for their TLP. Notice that an adversary would only be able to slow down protocol execution by forcing honest parties to solve its unopened TLPs but it wouldn't be able to bias the final output.

We describe protocol $\pi_{\mathsf{TLP-RB}}$ in Figure 7 and 8 and formally state its security in Theorem 2. We prove this theorem in the semi-synchronous model where $\mathcal{F}^{\Delta}_{\mathsf{BC,delay}}$ has a finite but unknown maximum delay Δ considering an honest majority. However, if we have synchronous broadcast allowing us to detect when parties do not act on time, we could prove it considering a dishonest majority by proceeding to the **Opening Phase** after the first TLP is solved, since we would have a guarantee that at least the one honest party would have broadcast its TLP by then.

Theorem 2. *There exists a delay parameter $\delta > \Delta$ such that Protocol $\pi_{\mathsf{TLP-RB}}$ UC-realizes $\mathcal{F}_{\mathsf{RB}}$ in the $\mathcal{F}_{\mathsf{tlp}}, \mathcal{F}^{\Delta}_{\mathsf{BC,delay}}$-hybrid model with computational security against a static adversary corrupting $t < \frac{n}{2}$ parties in $\mathcal{P}$. Formally, there exists a simulator $\mathcal{S}$ such that for every static adversary $\mathcal{A}$, and any environment $\mathcal{Z}$, the*

Protocol $\pi_{\mathsf{TLP-RB}}$

Protocol $\pi_{\mathsf{TLP-RB}}$ is parameterized by a delay δ and executed between a set of parties $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ out of which $t < n/2$ are corrupted and a set of verifiers $\mathcal{V}$ who interact with $\mathcal{F}^{\Delta}_{\mathsf{BC,delay}}$ and an instance $\mathcal{F}^{i}_{\mathsf{tlp}}$ of $\mathcal{F}_{\mathsf{tlp}}$ for which they act as $\mathcal{P}_o$:

Toss: On input ($\textsc{Toss}, \mathsf{sid}$), all parties in $\mathcal{P}$ proceed as follows:

1. **Commitment Phase:** For $i \in \{1, \ldots, n\}$, party $\mathcal{P}_i$ proceeds as follows:
   (a) Sample $r_i \xleftarrow{\$} \{0,1\}^{\tau}$ and send $(\mathsf{CreatePuzzle}, \mathsf{sid}, \delta, r_i)$ to $\mathcal{F}^{i}_{\mathsf{tlp}}$, receiving $(\mathsf{CreatedPuzzle}, \mathsf{sid}, \mathtt{puz}_i = (\mathtt{st}^i_0, \delta, \mathtt{tag}_i))$ in response.
   (b) Send $(\mathsf{Send}, \mathsf{sid}, \mathtt{puz}_i)$ to $\mathcal{F}^{\Delta}_{\mathsf{BC,delay}}$.
   (c) Send $(\mathsf{Fetch}, \mathsf{sid})$ to $\mathcal{F}^{\Delta}_{\mathsf{BC,delay}}$, receiving $(\mathsf{Fetch}, \mathsf{sid}, L)$. Check that $t+1$ messages of the form $(\mathcal{P}_j, \mathsf{sid}, \mathtt{puz}_j)$ from different parties are in L (we call the set of such parties $\mathcal{C}$). Once this check succeeds, record $\mathcal{C}$ and proceed.
2. **Opening Phase:** All parties $\mathcal{P}_i \in \mathcal{C}$ proceed as follows:
   (a) Send $(\mathsf{Send}, \mathsf{sid}, \mathtt{st}_\delta, r_i)$ to $\mathcal{F}^{\Delta}_{\mathsf{BC,delay}}$.
   (b) Wait for other parties to broadcast a solution to their TLPs by performing one iteration of the following loop every time it is activated:
      i. Send $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st}^i_{\mathtt{cst}_i})$ to $\mathcal{F}^{i}_{\mathsf{tlp}}$.
      ii. Send $(\mathsf{Fetch}, \mathsf{sid})$ to $\mathcal{F}^{i}_{\mathsf{tlp}}$ and check that there is an entry $(\mathcal{P}_i, \mathtt{st}^i_{\mathtt{cst}_i}, \mathtt{st}^i_{\mathtt{cst}_i+1})$ in L_i. If yes, increment $\mathtt{cst}_i$.
      iii. If $\mathtt{cst}_i = \delta$, exit the loop and proceeds to the next step.
   (c) Send $(\mathsf{Fetch}, \mathsf{sid})$ to $\mathcal{F}^{\Delta}_{\mathsf{BC,delay}}$, receiving $(\mathsf{Fetch}, \mathsf{sid}, L)$. Check that each message of the form $(\mathcal{P}_j, \mathsf{sid}, \mathtt{st}_\delta, r_j)$ from $\mathcal{P}_j \in \mathcal{C}$ is a valid solution to $\mathtt{puz}_j$ by sending $(\mathsf{Verify}, \mathsf{sid}, \mathtt{puz}_j, \mathtt{st}_\delta, r_j)$ to $\mathcal{F}_{\mathsf{tlp}}$ and checking that the answer is $(\mathsf{Verified}, \mathsf{sid}, \mathtt{puz}_j, \mathtt{st}_\delta, r_j, 1)$. If this check passes for all $\mathtt{puz}_j$ from $\mathcal{P}_j \in \mathcal{C}$, output $r = \bigoplus_{r_i \in \mathcal{V}} r_i$ and skip **Recovery Phase**. Otherwise, proceed.
3. **Recovery Phase:** For $i \in \{1, \ldots, n\}$, party $\mathcal{P}_i$ proceeds as follows:
   (a) For each j such that $\mathcal{P}_j \in \mathcal{C}$ did not send a valid solution of $\mathtt{puz}_j$ in the opening phase, solve $\mathtt{puz}_j = (\mathtt{st}^j_0, \delta, \mathtt{tag}_j)$ by setting $\mathtt{cst}_j = 0$ and performing one iteration of the following loop in parallel for all $\mathtt{puz}_j$ every time it is activated:
      i. Send $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st}^j_{\mathtt{cst}_j})$ to $\mathcal{F}^{j}_{\mathsf{tlp}}$.
      ii. Send $(\mathsf{Fetch}, \mathsf{sid})$ to $\mathcal{F}^{j}_{\mathsf{tlp}}$ and check that there is an entry $(\mathcal{P}_i, \mathtt{st}^j_{\mathtt{cst}_j}, \mathtt{st}^j_{\mathtt{cst}_j+1})$ in L_i. If yes, increment $\mathtt{cst}_j$.
      iii. If $\mathtt{cst}_j = \delta$, send $(\mathsf{GetMsg}, \mathsf{sid}, \mathtt{puz}_j, \mathtt{st}^j_{\mathtt{cst}_j})$ to $\mathcal{F}^{j}_{\mathsf{tlp}}$, obtaining $(\mathsf{GetMsg}, \mathsf{sid}, \mathtt{puz}_j, \mathtt{st}^j_{\mathtt{cst}_j}, r_j)$ in response. If all $r_1, \ldots, r_{|\mathcal{C}|}$ have been obtained, $\mathcal{P}_i$ exits the loop and proceeds to the next step.
   (b) Let $\mathcal{V}$ be the set of all $r_i \in \{r_1, \ldots, r_{|\mathcal{C}|}\}$ such that $r_i \neq \perp$ (*i.e.* $\mathcal{V}$ is the set of values r_i obtained from valid TLPs posted in the commitment phase). Send $(\mathsf{Send}, \mathsf{sid}, (\mathtt{st}_{j,\delta}, r_j)_{j | r_j \in \mathcal{V}})$ to $\mathcal{F}^{\Delta}_{\mathsf{BC,delay}}$. Output $r = \bigoplus_{r_i \in \mathcal{V}} r_i$.

Fig. 7. Protocol $\pi_{\mathsf{TLP-RB}}$

environment cannot distinguish and execution of $\pi_{\mathsf{TLP-RB}}$ *by* $\mathcal{A}$ *composed with* $\mathcal{F}_{\mathsf{tlp}}, \mathcal{F}^{\Delta}_{\mathsf{BC,delay}}$ *from an ideal execution with* $\mathcal{S}$ *and* $\mathcal{F}_{\mathsf{RB}}$.

Protocol $\pi_{\mathsf{TLP-RB}}$ (Verification)

Verify: On input $(\textsc{Verify}, \mathsf{sid}, x)$, send $(\mathsf{Fetch}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{BC,delay}}^{\Delta}$, receiving $(\mathsf{Fetch}, \mathsf{sid}, L)$ and determining $\mathcal{C}$ for sid from L. Check that each message of the form $(\mathcal{P}_j, \mathsf{sid}, \mathtt{st}_\delta, r_j)$ from $\mathcal{P}_j \in \mathcal{C}$ is a valid solution to $\mathtt{puz}_j$ by sending $(\mathsf{Verify}, \mathsf{sid}, \mathtt{puz}_j, \mathtt{st}_\delta, r_j)$ to $\mathcal{F}_{\mathsf{tlp}}$ and checking that the answer is $(\mathsf{Verified}, \mathsf{sid}, \mathtt{puz}_j, \mathtt{st}_\delta, r_j, 1)$. Output $r = \bigoplus_{r_i \in \mathcal{V}} r_i$.

Fig. 8. Protocol $\pi_{\mathsf{TLP-RB}}$

Proof. This theorem is proven in Supplementary Material C. $\square$

5.2 Using a Public Ledger $\mathcal{F}_{\mathsf{Ledger}}$

Instead of using a delayed broadcast $\mathcal{F}_{\mathsf{BC,delay}}^{\Delta}$, we can instantiate Protocol $\pi_{\mathsf{TLP-RB}}$ using a public ledger $\mathcal{F}_{\mathsf{Ledger}}$ for communication. In this case, we must parameterize the TLPs with a delay δ that is large enough to guarantee that all honest parties (including desynchronized ones) agree on the set of the first $t+1$ TLPs that are posted on the ledger before proceeding to the **Opening Phase**. We describe an alternative Protocol $\pi_{\mathsf{TLP-RB-LEDGER}}$ that behaves exactly as Protocol $\pi_{\mathsf{TLP-RB}}$ but leverages $\mathcal{F}_{\mathsf{Ledger}}$ for communication.

Protocol $\pi_{\mathsf{TLP-RB-LEDGER}}$: This protocol is exactly the same as $\pi_{\mathsf{TLP-RB}}$ except for using $\mathcal{F}_{\mathsf{Ledger}}$ for communication instead of $\mathcal{F}_{\mathsf{BC,delay}}^{\Delta}$ in the following way:

– At every point of $\pi_{\mathsf{TLP-RB}}$ where parties send $(\mathsf{Send}, \mathsf{sid}, m)$ to $\mathcal{F}_{\mathsf{BC,delay}}^{\Delta}$, instead they send $(\mathsf{Submit}, \mathsf{sid}, m)$ to $\mathcal{F}_{\mathsf{Ledger}}$.
– At every point of $\pi_{\mathsf{TLP-RB}}$ where parties send $(\mathsf{Fetch}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{BC,delay}}^{\Delta}$ and check for messages in $(\mathsf{Fetch}, \mathsf{sid}, L)$, instead they send $(\mathsf{Read}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{Ledger}}$ and check for messages in $(\mathsf{Read}, \mathsf{sid}, \mathsf{state}_i)$.

Theorem 3. *There exists a delay parameter $\delta > \mathsf{maxTXDelay} + \mathsf{emptyBlocks} \cdot \mathsf{slackWindow}$ such that Protocol $\pi_{\mathsf{TLP-RB-LEDGER}}$ UC-realizes $\mathcal{F}_{\mathsf{RB}}$ in the $\mathcal{F}_{\mathsf{tlp}}, \mathcal{F}_{\mathsf{Ledger}}$-hybrid model with computational security against a static adversary corrupting $t < \frac{n}{2}$ parties in $\mathcal{P}$. Formally, there exists a simulator $\mathcal{S}$ such that for every static adversary $\mathcal{A}$, and any environment $\mathcal{Z}$, the environment cannot distinguish an execution of $\pi_{\mathsf{TLP-RB-LEDGER}}$ by $\mathcal{A}$ composed with $\mathcal{F}_{\mathsf{tlp}}, \mathcal{F}_{\mathsf{Ledger}}$ from an ideal execution with $\mathcal{S}$ and $\mathcal{F}_{\mathsf{RB}}$.*

Proof. This theorem is proven in Supplementary Material C. $\square$

6 MPC with (Punishable) Output-Independent Abort

In this section we will describe how to construct a protocol that achieves MPC with output-independent abort and subsequently outline how to financially penalize cheating behavior in the protocol. The starting point of this construction will be MPC with secret-shared output, which is a strictly weaker primitive, as well as (synchronized) broadcast as modeled in $\mathcal{F}_{\mathsf{BC,delay}}^{\Delta}$ and TLPs.

6.1 Functionalities for Output-Independent Abort

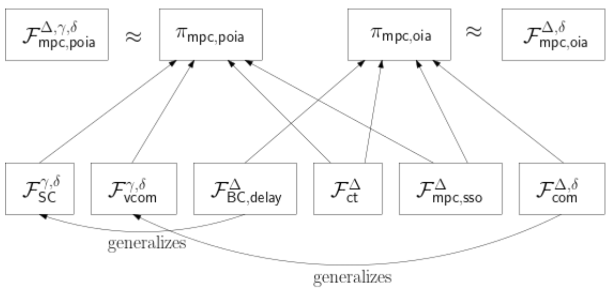
We begin by mentioning the functionalities that are used in our construction and which have not appeared in previous work (when modeled with respect to time). These functionalities are:

1. $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$ (Fig. 10 and Fig. 11) for secure MPC with secret-shared output.
2. $\mathcal{F}^{\Delta,\delta}_{\mathsf{com}}$ (Fig. 12) for commitments with delayed non-interactive openings.
3. $\mathcal{F}^{\Delta}_{\mathsf{ct}}$ (Fig. 17) for coin-flipping with abort.
4. $\mathcal{F}^{\Delta,\delta}_{\mathsf{mpc,oia}}$ (Fig. 13 and Fig. 14) for MPC with output-independent abort.

In the Supplementary Material, we will additionally introduce the following functionalities:

1. $\mathcal{F}^{\gamma,\delta}_{\mathsf{vcom}}$ (Fig. 22 and Fig. 23) for commitments with verifiable delayed non-interactive openings.
2. $\mathcal{F}^{\gamma,\delta}_{\mathsf{SC}}$ (Fig. 25 and Fig. 26) which is an abstraction of a smart contract.
3. $\mathcal{F}^{\Delta,\gamma,\delta}_{\mathsf{mpc,poia}}$ (Fig. 27 and Fig. 28) for MPC with Punishable Output-Independent Abort.

Before formally introducing these functionalities and explaining them in more detail, we will show how they are related in our construction.



$\mathcal{F}^{\Delta,\gamma,\delta}_{\mathsf{mpc,poia}}$
$\approx$
$\pi_{\mathsf{mpc,poia}}$
$\pi_{\mathsf{mpc,oia}}$
$\approx$
$\mathcal{F}^{\Delta,\delta}_{\mathsf{mpc,oia}}$
$\mathcal{F}^{\gamma,\delta}_{\mathsf{SC}}$
$\mathcal{F}^{\gamma,\delta}_{\mathsf{vcom}}$
$\mathcal{F}^{\Delta}_{\mathsf{BC,delay}}$
$\mathcal{F}^{\Delta}_{\mathsf{ct}}$
$\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$
$\mathcal{F}^{\Delta,\delta}_{\mathsf{com}}$
generalizes
generalizes

Fig. 9. How Punishable MPC is constructed.

As can be seen in Figure 9 our approach is twofold. First, we will realize $\mathcal{F}^{\Delta,\delta}_{\mathsf{mpc,oia}}$ via the protocol $\pi_{\mathsf{mpc,oia}}$ relying on $\mathcal{F}^{\Delta}_{\mathsf{BC,delay}}, \mathcal{F}^{\Delta}_{\mathsf{ct}}, \mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$ and $\mathcal{F}^{\Delta,\delta}_{\mathsf{com}}$. Then, we will show how to implement $\mathcal{F}^{\Delta,\gamma,\delta}_{\mathsf{mpc,poia}}$ via the protocol $\pi_{\mathsf{mpc,poia}}$ (a generalization of $\pi_{\mathsf{mpc,oia}}$) which uses $\mathcal{F}^{\gamma,\delta}_{\mathsf{SC}}, \mathcal{F}^{\Delta}_{\mathsf{ct}}, \mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$ as well as $\mathcal{F}^{\gamma,\delta}_{\mathsf{vcom}}$. As mentioned in Fig. 9, $\mathcal{F}^{\gamma,\delta}_{\mathsf{vcom}}$ and $\mathcal{F}^{\gamma,\delta}_{\mathsf{SC}}$ can be thought of as generalizations of $\mathcal{F}^{\Delta,\delta}_{\mathsf{com}}$ and $\mathcal{F}^{\Delta}_{\mathsf{BC,delay}}$. We will now describe the functionalities for $\pi_{\mathsf{mpc,oia}}$ in more detail.

MPC with Secret-Shared Output. The functionality $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$ is formally introduced in Fig. 10 and Fig. 11. It directly translates an MPC protocol with

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

The ticked functionality interacts with n parties $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ and an adversary $\mathcal{S}$ which may corrupt a strict subset $I \subset \mathcal{P}$.

Init: On first input $(\mathsf{Init}, \mathsf{sid}, C)$ by $\mathcal{P}_i \in \mathcal{P}$:
1. Send message C to the parties $\mathcal{P} \setminus \{\mathcal{P}_i\}$ via $\mathcal{Q}$ with delay Δ.
2. If each party sent $(\mathsf{Init}, \mathsf{sid}, C)$ then store C locally.
3. Send C and the IDs to $\mathcal{S}$.

Input: On first input $(\mathsf{Input}, \mathsf{sid}, i, x_i)$ by $\mathcal{P}_i \in \mathcal{P}$:
1. Notify parties $\mathcal{P} \setminus \{\mathcal{P}_i\}$ via $\mathcal{Q}$ with delay Δ.
2. Accept x_i as input for $\mathcal{P}_i$.
3. Send m and the IDs to $\mathcal{S}$ if $\mathcal{P}_i \in I$, otherwise notify $\mathcal{S}$ about a message with prefix Input.

Computation: On first input $(\mathsf{Compute}, \mathsf{sid})$ by $\mathcal{P}_i \in \mathcal{P}$ and if $x_1, \ldots, x_n$ were accepted:
1. Notify parties $\mathcal{P} \setminus \{\mathcal{P}_i\}$ via $\mathcal{Q}$ with delay Δ.
2. If all parties sent $(\mathsf{Compute}, \mathsf{sid})$ compute and store $(y_1, \ldots, y_m) \leftarrow C(x_1, \ldots, x_n)$.
3. Notify $\mathcal{S}$ about a message with prefix $\mathsf{Compute}$.

Tick:
1. Remove each $(0, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$ from $\mathcal{Q}$ and instead add $(\mathcal{P}_i, \mathsf{sid}, m)$ to $\mathcal{M}$.
2. Replace each $(\mathtt{cnt}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$ in $\mathcal{Q}$ with $(\mathtt{cnt} - 1, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$.

Upon receiving $(\mathsf{Schedule}, \mathsf{sid}, \mathcal{D})$ from $\mathcal{S}$:
- If $(\mathsf{Deliver}, \mathsf{sid}, \mathsf{mid}) \in \mathcal{D}$ then remove each $(\mathtt{cnt}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$ from $\mathcal{Q}$ and add $(\mathcal{P}_i, \mathsf{sid}, m)$ to $\mathcal{M}$.
- If $(\mathsf{Abort}, \mathsf{sid}) \in \mathcal{D}$ then add $(\mathcal{P}_i, \mathsf{sid}, \mathsf{Abort})$ to $\mathcal{M}$ for each $i \in [n]$ and ignore all further messages with this sid except to **Fetch Message**.

Fig. 10. Ticked Functionality $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$ for MPC with Secret-Shared Output and Linear Secret Share Operations.

secret-shared output into the ACT model, but does not make use of any tick-related properties beyond scheduling of message transmission. The functionality supports computations on secret input where the output of the computation is additively secret-shared among the participants. Additionally, it allows parties to sample random values, compute linear combinations of outputs and those random values and allows to reliably but unfairly open secret-shared values.

$\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$ can be instantiated from many different MPC protocols, such as those based on secret-sharing [7] or multiparty BMR [19].

Commitments with Delayed Openings. In Fig. 12 we describe the functionality $\mathcal{F}^{\Delta,\delta}_{\mathsf{com}}$ for commitments with verifiable delayed non-interactive openings. The functionality distinguishes between a sender $\mathcal{P}_{\mathsf{Send}}$, which is allowed to make commitments, and a set of receivers, which will obtain the openings. In comparison to regular commitments with a normal **Open** that simply reveals the output to all parties, the sender is also allowed to perform a **Delayed Open**. This means

Functionality $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$ (Computation on Outputs)

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

1. Notify parties $\mathcal{P} \setminus \{\mathcal{P}_i\}$ via $\mathcal{Q}$ with delay Δ.
2. If all parties sent $\mathsf{ShareOutput}$:
   (a) Send $(\mathsf{RequestShares}, \mathsf{sid}, \mathcal{T})$ to $\mathcal{S}$, which replies with $(\mathsf{OutputShares}, \mathsf{sid}, \{s_{j,\mathsf{cid}}\}_{\mathsf{cid} \in \mathcal{T}, \mathcal{P}_j \in I})$. Then for each $\mathcal{P}_j \in \mathcal{P} \setminus I, h \in [m]$ sample $s_{j,\mathsf{cid}_h} \leftarrow \mathbb{F}$ uniformly random conditioned on $y_h = \bigoplus_{k \in [n]} s_{k,\mathsf{cid}_h}$.
   (b) For $\mathsf{cid} \in \mathcal{T}$ store $(\mathsf{cid}, s_{1,\mathsf{cid}}, \ldots, s_{n,\mathsf{cid}})$ and for each $\mathcal{P}_j \in \mathcal{P} \setminus I$ send $s_{j,\mathsf{cid}}$ with prefix $\mathsf{OutputShares}$ to party $\mathcal{P}_j$ via $\mathcal{Q}$ with delay Δ. Finally notify $\mathcal{S}$ about the message with prefix $\mathsf{OutputShares}$.
3. Notify $\mathcal{S}$ about a message with the prefix $\mathsf{ShareOutput}$.

Share Random Value: Upon input $(\mathsf{ShareRandom}, \mathsf{sid}, \mathcal{T})$ by all parties with fresh identifiers $\mathcal{T}$:

1. Notify parties $\mathcal{P} \setminus \{\mathcal{P}_i\}$ via $\mathcal{Q}$ with delay Δ.
2. If all parties sent $\mathsf{ShareRandom}$:
   (a) Send $(\mathsf{RequestShares}, \mathsf{sid}, \mathcal{T})$ to $\mathcal{S}$, which replies with $(\mathsf{RandomShares}, \mathsf{sid}, \{s_{j,\mathsf{cid}}\}_{\mathsf{cid} \in \mathcal{T}, \mathcal{P}_j \in I})$. Then for each $\mathcal{P}_j \in \mathcal{P} \setminus I, \mathsf{cid} \in \mathcal{T}$ sample $s_{j,\mathsf{cid}} \leftarrow \mathbb{F}$ uniformly at random.
   (b) For $\mathsf{cid} \in \mathcal{T}$ store $(\mathsf{cid}, s_{1,\mathsf{cid}}, \ldots, s_{n,\mathsf{cid}})$ and for each $\mathcal{P}_j \in \mathcal{P} \setminus I$ send $s_{j,\mathsf{cid}}$ with prefix $\mathsf{RandomShares}$ to party $\mathcal{P}_j$ via $\mathcal{Q}$ with delay Δ. Finally notify $\mathcal{S}$ about the message with prefix $\mathsf{RandomShares}$.
3. Notify $\mathcal{S}$ about a message with the prefix $\mathsf{ShareRandom}$.

Linear Combination: Upon input $(\mathsf{Linear}, \mathsf{sid}, \{(\mathsf{cid}, \alpha_{\mathsf{cid}})\}_{\mathsf{cid} \in \mathcal{T}}, \mathsf{cid}')$ from all parties: If all $\alpha_{\mathsf{cid}} \in \mathbb{F}$, all $(\mathsf{cid}, s_{1,\mathsf{cid}}, \ldots, s_{n,\mathsf{cid}})$ have been stored and cid' is unused, set $s'_i \leftarrow \sum_{\mathsf{cid} \in \mathcal{T}} \alpha_{\mathsf{cid}} \cdot s_{i,\mathsf{cid}}$ and record $(\mathsf{cid}', s'_1, \ldots, s'_n)$.

Reveal: Upon input $(\mathsf{Reveal}, \mathsf{sid}, \mathcal{T})$ by $\mathcal{P}_i \in \mathcal{P}$ for identifiers $\mathcal{T}$ and if $(\mathsf{cid}, s_1, \ldots, s_n)$ is stored for each $\mathsf{cid} \in \mathcal{T}$:

1. Notify the parties $\mathcal{P} \setminus \{\mathcal{P}_i\}$ via $\mathcal{Q}$ with delay Δ. Then notify $\mathcal{S}$ about a message with prefix Reveal.
2. If all parties sent $(\mathsf{Reveal}, \mathsf{sid}, \mathcal{T})$ then send $(\mathsf{Reveal}, \mathsf{sid}, \{(\mathsf{cid}, s_{1,\mathsf{cid}}, \ldots, s_{n,\mathsf{cid}})\}_{\mathsf{cid} \in \mathcal{T}})$ to $\mathcal{S}$.
3. If $\mathcal{S}$ sends $(\mathsf{DeliverReveal}, \mathsf{sid}, \mathcal{T})$ then send message $\{(\mathsf{cid}, s_{1,\mathsf{cid}}, \ldots, s_{n,\mathsf{cid}})\}_{\mathsf{cid} \in \mathcal{T}}$ with prefix $\mathsf{DeliverReveal}$ to parties $\mathcal{P}$ via $\mathcal{Q}$ with delay Δ and notify $\mathcal{S}$ about a message with prefix $\mathsf{DeliverReveal}$.

Fig. 11. Ticked Functionality $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$ for MPC with Secret-Shared Output and Linear Secret Share Operations, Part 2.

that there is a delay between the choice of a sender to open a commitment (or not) and the actual opening towards the receivers and also the adversary. $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta}$ is a generalization of a similar two-party functionality that can already be found in [6]. At the same time, it is incomparable as it does not provide any homomorphic operations on the secret values.

While both the **Commit** and **Open** directly resemble their counterparts in a normal commitment functionality, the **Delayed Open** logic is not as straight-

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

The ticked functionality is parameterized by $\Delta, \delta \in \mathbb{N}$ and interacts with a set of n parties $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ where $\mathcal{P}_{\mathsf{Send}} \in \mathcal{P}$ is a special party called "the sender" and $\mathcal{P}_{\mathsf{Rec}} = \mathcal{P} \setminus \{\mathcal{P}_{\mathsf{Send}}\}$ are the receivers. An adversary $\mathcal{S}$ may corrupt a strict subset $I \subset \mathcal{P}$ of parties. The functionality internally has an initially empty list $\mathcal{O}$ and a map $\mathsf{commits}$.

Commit: Upon receiving $(\mathsf{Commit}, \mathsf{sid}, \mathsf{cid}, x)$ from $\mathcal{P}_{\mathsf{Send}}$ where cid is an unused identifier and x is a bit-string proceed as follows:

1. Set $\mathsf{commits}[\mathsf{cid}] = x$.
2. Send a message cid with prefix Commit to $\mathcal{P}_{\mathsf{Rec}}$ via $\mathcal{Q}$ with delay Δ.
3. Send cid and the IDs to $\mathcal{S}$.

Open: Upon receiving $(\mathsf{Open}, \mathsf{sid}, \mathsf{cid})$ from $\mathcal{P}_{\mathsf{Send}}$, if $\mathsf{commits}[\mathsf{cid}] = x \neq \perp$ then proceed as follows:

1. Send message (cid, x) with prefix Open to $\mathcal{P}_{\mathsf{Rec}}$ via $\mathcal{Q}$ with delay Δ.
2. Send (cid, x) and the IDs to $\mathcal{S}$.

Delayed Open: Upon receiving $(\mathsf{DOpen}, \mathsf{sid}, \mathsf{cid})$ from $\mathcal{P}_{\mathsf{Send}}$, if $\mathsf{commits}[\mathsf{cid}] = x \neq \perp$ then proceed as follows:

1. Simultaneously send message cid with prefix DOpen to all parties in $\mathcal{P}_{\mathsf{Rec}}$ via $\mathcal{Q}$ with delay Δ.
2. Add $(\delta, \mathsf{sid}, \mathcal{P}_j, (\mathsf{cid}, x))$ for each $\mathcal{P}_j \in \mathcal{P}_{\mathsf{Rec}}$ and $(\delta, \mathsf{sid}, \mathcal{S}, (\mathsf{cid}, x))$ to $\mathcal{O}$.
3. Send cid and the ID to $\mathcal{S}$.

Tick:

1. Remove each $(0, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$ from $\mathcal{Q}$ and instead add $(\mathcal{P}_i, \mathsf{sid}, m)$ to $\mathcal{M}$.
2. Replace each $(\overline{\mathtt{cnt}}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$ in $\mathcal{Q}$ with $(\mathtt{cnt} - 1, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$.
3. For each entry $(\mathtt{cnt}, \mathsf{sid}, \mathcal{P}_j, (\mathsf{cid}, x) \in \mathcal{O}$ with $\mathcal{P}_j \in \mathcal{P}_{\mathsf{Rec}}$, if there is no entry $(\mathtt{cnt}, \mathsf{sid}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_j, (\mathsf{DOpen}, \mathsf{cid})) \in \mathcal{Q}$, proceed as follows:
   - If $\mathtt{cnt} = 0$, append $(\mathcal{P}_j, \mathsf{sid}, (\mathsf{DOpened}, (\mathsf{cid}, x)))$ to $\mathcal{M}$.
   - If $\mathtt{cnt} > 0$, replace $(\mathtt{cnt}, \mathsf{sid}, \mathcal{P}_j, (\mathsf{cid}, x))$ with $(\mathtt{cnt} - 1, \mathsf{sid}, \mathcal{P}_j, (\mathsf{cid}, x))$ in $\mathcal{O}$.
4. For each entry $(\mathtt{cnt}, \mathsf{sid}, \mathcal{S}, (\mathsf{cid}, x)) \in \mathcal{O}$, proceed as follows:
   - If $\mathtt{cnt} = 0$, append $(\mathcal{P}_{\mathsf{Send}}, \mathsf{sid}, (\mathsf{DOpened}, \mathsf{cid}))$ to $\mathcal{M}$ and output $(\mathsf{DOpen}, \mathsf{sid}, \mathcal{S}, (\mathsf{cid}, x))$ to $\mathcal{S}$.
   - If $\mathtt{cnt} > 0$, replace $(\mathtt{cnt}, \mathsf{sid}, \mathcal{S}, (\mathsf{cid}, x))$ with $(\mathtt{cnt} - 1, \mathsf{sid}, \mathcal{S}, (\mathsf{cid}, x))$ in $\mathcal{O}$.

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

- If $(\mathsf{Deliver}, \mathsf{sid}, \mathsf{mid}) \in \mathcal{D}$ then remove each $(\mathtt{cnt}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$ from $\mathcal{Q}$ and add $(\mathcal{P}_i, \mathsf{sid}, m)$ to $\mathcal{M}$.
- If $(\mathsf{Abort}, \mathsf{sid}) \in \mathcal{D}$ and $\mathsf{open} = 0$ then add $(\mathcal{P}_i, \mathsf{sid}, \mathsf{Abort})$ to $\mathcal{M}$ for each $i \in [n]$ and ignore all further messages with this sid except to **Fetch Message**.

Fig. 12. Ticked Functionality $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta}$ For Commitments with Delayed Opening.

forward. What happens during such a delayed open is that first all honest parties will simultaneously learn that indeed an opening will happen in the future - this is the DOpen message which they obtain. Additionally, $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta}$ stores the openings in an internal queue $\mathcal{O}$. These openings *can not be rescheduled by the adversary,* and therefore it will take δ ticks before $\mathcal{S}$ actually learns the opening of the commitment. For honest parties, this may even take up to $\Delta+\delta$ ticks depending when DOpen is obtained by the honest parties. As the openings, once triggered by **Tick**, are directly written to the output queue $\mathcal{M}$ and not $\mathcal{Q}$, these can then directly be read by the respective parties. This also means that *all honest parties will learn the delayed opening simultaneously.*

In Supplementary Material D we provide a secure instantiation of a publicly verifiable[4] version of $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta}$. Since we do not require homomorphic operations, this means that it can be realized with a much simpler protocol than the respective two-party functionality in [6].

MPC with Output-Independent Abort. In Fig. 13 and Fig. 14 we describe the functionality $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta}$ for MPC with output-independent abort.

In terms of the actual secure computation, our functionality is identical with $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$, although it does not reveal the concrete shares to the parties and the adversary during the sharing. The output-independent abort property of our functionality is then achieved as follows: in order to reveal the output of the computation, at least one party will first have to send Reveal to $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta}$. Once all honest parties and the verifiers thus learn that the parties indeed are synchronized by seeing that *the first synchronization message arrives at all parties*($\mathtt{st} = \mathtt{sync}$ and $\mathtt{f} = \top$), the internal state of the functionality changes. From this point on, the adversary can, within an additional time-frame of δ ticks, decide whether to reveal its shares or not. Then, once these δ ticks passed, $\mathcal{S}$ will obtain the output y of the computation *after* having provided the set of aborting parties J. If $J = \emptyset$ then $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta}$ will, within δ additional ticks, simultaneously output y to all honest parties, while it otherwise outputs the set J.

The additional up to δ ticks between the adversary learning y and the honest parties learning y or J is due to our protocol and will be more clear later.

Coin Tossing. $\pi_{\mathsf{mpc,oia}}$ additionally requires a functionality for coin tossing $\mathcal{F}_{\mathsf{ct}}^{\Delta}$ as depicted in Fig. 17 in the Supplementary Material. Note that $\mathcal{F}_{\mathsf{ct}}^{\Delta}$ can easily be realized in the $\mathcal{F}_{\mathsf{BC,delay}}^{\Delta}, \mathcal{F}_{\mathsf{com}}^{\Delta,\delta}$-hybrid model.

6.2 Building MPC with Output-Independent Abort

We will now describe how to construct an MPC protocol that guarantees output-independent abort. This generalizes the previous work of [6] to a multiparty setting. Although this might appear like a natural generalization, constructing the protocol is far from trivial as we must take care that all honest parties agree on the exact set of cheaters.

Our protocol, on a high level, works as follows:

[4] See Theorem 5 for more details. In order to adapt the construction to $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta}$ it is sufficient to replace the required bulletin board with our broadcast functionality $\mathcal{F}_{\mathsf{BC,delay}}^{\Delta}$.

Functionality $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta}$(Computation, Sharing)

The ticked functionality runs with n parties $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ and an adversary $\mathcal{S}$ who may corrupt a strict subset $I \subset \mathcal{P}$. $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta}$ is parameterized by $\Delta, \delta \in \mathbb{N}^+$. The functionality internally has an initially empty list $\mathcal{O}$, a state $\mathtt{st}$ initially $\perp$ as well as an initially empty set J.

Init: On first input $(\mathsf{Init}, \mathsf{sid}, C)$ by $\mathcal{P}_i \in \mathcal{P}$:
1. Send message C to the parties $\mathcal{P} \setminus \{\mathcal{P}_i\}$ via $\mathcal{Q}$ with delay Δ.
2. If each party sent $(\mathsf{Init}, \mathsf{sid}, C)$ then store C locally.
3. Send C and the IDs to $\mathcal{S}$.

Input: On first input $(\mathsf{Input}, \mathsf{sid}, i, x_i)$ by $\mathcal{P}_i \in \mathcal{P}$:
1. Notify parties $\mathcal{P} \setminus \{\mathcal{P}_i\}$ via $\mathcal{Q}$ with delay Δ.
2. Accept x_i as input for $\mathcal{P}_i$.
3. Send x_i and the IDs to $\mathcal{S}$ if $\mathcal{P}_i \in I$, otherwise notify $\mathcal{S}$ about a message with prefix Input.

Computation: On first input $(\mathsf{Compute}, \mathsf{sid})$ by $\mathcal{P}_i \in \mathcal{P}$ and if all $x_1, \ldots, x_n$ were accepted:
1. Notify parties $\mathcal{P} \setminus \{\mathcal{P}_i\}$ via $\mathcal{Q}$ with delay Δ.
2. If each party sent $(\mathsf{Compute}, \mathsf{sid})$ compute $y = C(x_1, \ldots, x_n)$ and store y.
3. Notify $\mathcal{S}$ about a message with prefix $\mathsf{Compute}$.

Share: On first input $(\mathsf{Share}, \mathsf{sid})$ by party $\mathcal{P}_i$, if y has been stored and if $\mathtt{st} = \perp$:
1. Notify parties $\mathcal{P} \setminus \{\mathcal{P}_i\}$ via $\mathcal{Q}$ with delay Δ.
2. If all parties sent Share then:
   (a) Send $(\mathsf{Shares?}, \mathsf{sid})$ to $\mathcal{S}$.
   (b) Upon $(\mathsf{DeliverShares}, \mathsf{sid})$ from $\mathcal{S}$ simultaneously send a message with prefix $\mathsf{DeliverShares}$ to each $\mathcal{P}_j \in \mathcal{P} \setminus I$ via $\mathcal{Q}$ with delay Δ. Then notify $\mathcal{S}$ about messages with prefix $\mathsf{DeliverShares}$ and the ID.
   (c) Otherwise, if $\mathcal{S}$ sends $(\mathsf{Abort}, \mathsf{sid})$ then send Abort to all parties
3. Notify $\mathcal{S}$ about a message with prefix Share.

Reveal: Upon first message $(\mathsf{Reveal}, \mathsf{sid}, i)$ by party $\mathcal{P}_i \in \mathcal{P}$, if **Share** has finished, if no $\mathsf{DeliverShare}$ message is in $\mathcal{Q}$ and if $\mathtt{st} = \perp$ or $\mathtt{st} = \mathtt{sync}$:
1. Simultaneously send a message i with prefix Reveal to parties $\mathcal{P} \setminus \{\mathcal{P}_i\}$ via $\mathcal{Q}$ with delay Δ.
2. Set $\mathtt{st} = \mathtt{sync}$.
3. Notify $\mathcal{S}$ about a message with prefix Reveal.

Fig. 13. The $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta}$ Functionality for MPC with Output-Independent Abort.

1. The parties provide their inputs x_i to $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$, perform the computation using $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$ and obtain secret shares $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n$ of the output $\boldsymbol{y}$. Additionally, they sample a blinding value $\boldsymbol{r}_i \in \mathbb{F}^{\lambda}$ for each party $\mathcal{P}_i$ inside $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$. The values $\boldsymbol{y}_i, \boldsymbol{r}_i$ are sent to each $\mathcal{P}_i$.
2. Next, the parties commit to both $\boldsymbol{y}_i, \boldsymbol{r}_i$ using $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta}$ towards all parties. Dishonest parties may commit to a different value than the one they obtained from $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$ and consistency must therefore be checked.
3. All parties use the coin-flipping functionality to sample a uniformly random matrix $\mathbf{A} \in \mathbb{F}^{\lambda \times m}$. This matrix is used to perform the consistency check.

```
Functionality $\mathcal{F}^{\Delta,\delta}_{\mathsf{mpc,oia}}$ (Timing)

Tick:
    1. Set $\mathtt{f} \leftarrow \bot$.
    2. Remove each $(0, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$ from $\mathcal{Q}$ and instead add $(\mathcal{P}_i, \mathsf{sid}, m)$ to $\mathcal{M}$.
       If $m = (\mathsf{Reveal}, i)$ then set $\mathtt{f} \leftarrow \top$.
    3. Replace each $(\mathtt{cnt}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$ in $\mathcal{Q}$ with $(\mathtt{cnt} - 1, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$.
    4. If $\mathtt{st} = \mathtt{wait}(x)$ & $x \geq 0$:
       If $x \geq 0$: Set $\mathtt{st} = \mathtt{wait}(x-1)$.
       If $x = 0$:
           (a) Send $(\mathsf{Abort?}, \mathsf{sid})$ to $\mathcal{S}$ and wait for response $(\mathsf{Abort}, \mathsf{sid}, J)$ with
               $J \subseteq I$.
           (b) If $J = \emptyset$ then send message $y$ with prefix $\mathsf{Output}$ to each party
               $\mathcal{P} \setminus I$ via $\mathcal{Q}$ with delay $\delta$.
           (c) If $J \neq \emptyset$ then send message $J$ with prefix $\mathsf{Abort}$ to each party $\mathcal{P} \setminus I$
               via $\mathcal{Q}$ with delay $\delta$.
           (d) Send $(\mathsf{Output}, \mathsf{sid}, y)$ and the IDs to $\mathcal{S}$.
    5. If $\mathtt{st} = \mathtt{sync}$ and $f = \top$:
       (a) Set $\mathtt{st} = \mathtt{wait}(\delta)$.
       (b) Send $(\mathsf{RevealStart}, \mathsf{sid})$ to $\mathcal{S}$
  Upon receiving $(\mathsf{Schedule}, \mathsf{sid}, \mathcal{D})$ from $\mathcal{S}$:
    – If $(\mathsf{Deliver}, \mathsf{sid}, \mathsf{mid}) \in \mathcal{D}$ then remove each $(\mathtt{cnt}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$ from $\mathcal{Q}$ and
      add $(\mathcal{P}_i, \mathsf{sid}, m)$ to $\mathcal{M}$.
    – If $(\mathsf{Abort}, \mathsf{sid}) \in \mathcal{D}$ and $\mathtt{st} = \bot$ then add $(\mathcal{P}_i, \mathsf{sid}, \mathsf{Abort})$ to $\mathcal{M}$ for each
      $\mathcal{P}_i \in \mathcal{P}$ and ignore all further messages with this $\mathsf{sid}$ except to Fetch
      Message.
```

Fig. 14. The Ticked $\mathcal{F}^{\Delta,\delta}_{\mathsf{mpc,oia}}$ Functionality for MPC with Output-Independent Abort.

4. For each $i \in [n]$ the parties compute and open $\boldsymbol{t}_i = \boldsymbol{r}_i + \mathbf{A}\boldsymbol{y}_i$ using $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$. Due to the blinding value $\boldsymbol{r}_i$ opening $\boldsymbol{t}_i$ will not leak any information about $\boldsymbol{y}_i$ of $\mathcal{P}_i \in \mathcal{P} \setminus I$ to the adversary.
5. Now the parties synchronize using $\mathcal{F}^{\Delta}_{\mathsf{BC,delay}}$. Once synchronized, they simultaneously perform a delayed open of both $\boldsymbol{y}_i, \boldsymbol{r}_i$ using their commitments. Parties which don't open their commitments in time or whose opened values do not yield $\boldsymbol{t}_i$ by the same equation as above are considered as cheaters.

Intuitively, our construction has output-independent abort because of the timing of the opening: Until Step 5, the adversary may abort at any time but no such abort will provide it with information about the output. Once the opening phase begins, parties can easily verify if an opening by an adversary is valid or not - because he committed to its shares before $\mathbf{A}$ was chosen and the probability of a collision with $\boldsymbol{t}_i$ for different choices of $\boldsymbol{y}'_i, \boldsymbol{r}'_i$ can be shown to be negligible in λ as this is exactly the same as finding a collision to a universal hash function. The decision to initiate its opening, on the other hand, must arrive at each honest party before the honest party's delayed opening finishes - which will be ensured by the appropriate choice of δ with respect to Δ for honest parties. In turn, an adversary must thus send its opening message before learning the shares of an honest party, which is exactly the property of output-independent abort.

Identifiability of the abort requires that the honest parties agree on the same set of cheaters. Towards achieving this, we need the synchronized broadcast that is used in Step 1 of **Reveal**. If parties would start the opening at different times, then there would not be agreement about a point at which all parties decide that a certain received message timed out. But this agreement is crucial to decide which parties cheated.

Interestingly, our construction does not need homomorphic commitments as was necessary in [6,5] to achieve their verifiable or output-independent abort in UC. Clearly, our solution can also be used to improve these protocols and to simplify their constructions. The full protocol can be found in Fig. 15 and Fig. 16.

Protocol $\pi_{\mathsf{mpc,oia}}$ (Computation, Share)

All parties $\mathcal{P}$ have access to one instance of the functionalities $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$ and $\mathcal{F}^{\Delta}_{\mathsf{ct}}$. Furthermore, each $\mathcal{P}_i \in \mathcal{P}$ has it's own $\mathcal{F}^{\Delta,i}_{\mathsf{BC,delay,}}$ and $\mathcal{F}^{\Delta,\delta,i}_{\mathsf{com}}$ where it acts as the dedicated sender and all other parties of $\mathcal{P}$ are receivers.

Init: Each $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{Init}, \mathsf{sid}, C)$ to $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$ and waits until it obtains messages C with prefix Init from $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$ for every other party $\mathcal{P} \setminus \{\mathcal{P}_i\}$.

Input: Each $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{Input}, \mathsf{sid}, i, x_i)$ to $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$ and waits until it obtains messages j with prefix Input from $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$ for every other party $\mathcal{P}_j \in \mathcal{P} \setminus \{\mathcal{P}_i\}$.

Computation: Each $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{Computation}, \mathsf{sid})$ to $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$ and waits until it obtains messages with prefix $\mathsf{Computation}$ from $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$ for every other party $\mathcal{P} \setminus \{\mathcal{P}_i\}$.

Share:

1. Set $\mathcal{T}_y = \{\mathsf{cid}_{y,j}\}_{j\in[m]}$, $\mathcal{T}_r = \{\mathsf{cid}_{r,k}\}_{k\in[\lambda]}$ and $\mathcal{T}_t = \{\mathsf{cid}_{t,k}\}_{k\in[\lambda]}$.
2. Each $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{ShareOutput}, \mathsf{sid}, \mathcal{T}_y)$ to $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$ and wait until it obtains a message $\{y_{i,\mathsf{cid}}\}_{\mathsf{cid}\in\mathcal{T}_y}$ with prefix $\mathsf{OutputShares}$ from $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$.
3. Each $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{ShareRandom}, \mathsf{sid}, \mathcal{T}_r)$ to $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$ and wait until it obtains a message $\{r_{i,\mathsf{cid}}\}_{\mathsf{cid}\in\mathcal{T}_r}$ with prefix $\mathsf{RandomShares}$ from $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$. Set $\boldsymbol{y}_i = (y_{i,\mathsf{cid}_{y,1}}, \ldots, y_{i,\mathsf{cid}_{y,m}})$ and equivalently define $\boldsymbol{r}_i$.
4. Each $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{Commit}, \mathsf{sid}, \mathsf{cid}_i, (\boldsymbol{y}_i, \boldsymbol{r}_i))$ to $\mathcal{F}^{\Delta,\delta,i}_{\mathsf{com}}$ and then waits for messages $(\mathsf{Commit}, \mathsf{sid}, \mathsf{cid}_j)$ from the $\mathcal{F}^{\Delta,\delta,j}_{\mathsf{com}}$-instances of all other parties $\mathcal{P}_j \in \mathcal{P} \setminus \{\mathcal{P}_i\}$.
5. Each $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{Toss}, \mathsf{sid}, m \cdot \lambda)$. It then waits for the message $(\mathsf{Coins}, \mathsf{sid}, \mathbf{A})$ where $\mathbf{A} \in \mathbb{F}^{\lambda\times m}$.
6. Each $\mathcal{P}_i \in \mathcal{P}$ for $k \in [\lambda]$ sends $(\mathsf{Linear}, \mathsf{sid}, \{(\mathsf{cid}_{v,j}, \mathbf{A}[k,j])\}_{j\in[m]} \cup \{(\mathsf{cid}_{r,k}, 1)\}, \mathsf{cid}_{t,k})$ to $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$.
7. Each $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{Reveal}, \mathsf{sid}, \mathcal{T}_t)$ to $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$ and waits for the message $\{(\mathsf{cid}, t_{1,\mathsf{cid}}, \ldots, t_{n,\mathsf{cid}})\}_{\mathsf{cid}\in\mathcal{T}_t}$ with prefix $\mathsf{DeliverReveal}$ from $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$. Set $\boldsymbol{t}_j = (t_{j,\mathsf{cid}_{t,1}}, \ldots, t_{j,\mathsf{cid}_{t,\lambda}})$ for each $j \in [n]$.

Fig. 15. Protocol $\pi_{\mathsf{mpc,oia}}$ for MPC with Output-Independent Abort.

Protocol $\pi_{\mathsf{mpc,oia}}$ (Reveal)

Reveal: If **Share** completed successfully:

1. Each $\mathcal{P}_i$ sends $(\mathsf{Send}, \mathsf{sid}, \mathsf{sync})$ to $\mathcal{F}^{\Delta,i}_{\mathsf{BC,delay},\cdot}$.
2. If $\mathcal{P}_i$ obtains the first message with prefix Sync from *any* $\mathcal{F}^{\Delta,j}_{\mathsf{BC,delay},}$ then $\mathcal{P}_i$ sends $(\mathsf{DOpen}, \mathsf{sid}, \mathsf{cid}_i)$ to$\mathcal{F}^{\Delta,\delta,i}_{\mathsf{com}}$.
3. Each $\mathcal{P}_i \in \mathcal{P}$ waits until $\mathcal{F}^{\Delta,\delta,i}_{\mathsf{com}}$ returns $(\mathsf{DOpened}, \mathsf{sid}, (\mathsf{cid}_i, (\boldsymbol{y}_i, \boldsymbol{r}_i))$. It then checks if it obtained a message with prefix DOpen from all other $\mathcal{F}^{\Delta,\delta,j}_{\mathsf{com}}$. Let $J_1 \subset \mathcal{P}$ be the set of parties such that $\mathcal{P}_i$ did not obtain DOpen before it received $(\mathsf{DOpened}, \mathsf{sid}, (\mathsf{cid}_i, (\boldsymbol{y}_i, \boldsymbol{r}_i))$.
4. Each $\mathcal{P}_i \in \mathcal{P}$ waits until it obtains $(\mathsf{DOpened}, \mathsf{sid}, (\mathsf{cid}_j, (\boldsymbol{y}_j, \boldsymbol{r}_j))$ for each $\mathcal{P}_j \in \mathcal{P} \setminus (J_1 \cup \{\mathcal{P}_i\})$ from the respective instance of $\mathcal{F}^{\Delta,\delta,j}_{\mathsf{com}}$. It then defines J_2 as the set of all parties $\mathcal{P}_j$ such that $\boldsymbol{t}_j \neq \boldsymbol{r}_j + \mathbf{A}\boldsymbol{y}_j$.
5. If $J_1 \cup J_2 = \emptyset$ then each $\mathcal{P}_i \in \mathcal{P}$ outputs $(\mathsf{Output}, \mathsf{sid}, \boldsymbol{y} = \bigoplus_{j\in[n]} \boldsymbol{y}_j)$ and terminates. Otherwise it outputs $(\mathsf{Abort}, \mathsf{sid}, J_1 \cup J_2)$.

Fig. 16. Protocol $\pi_{\mathsf{mpc,oia}}$ for MPC with Output-Independent Abort.

In Supplementary Material D we show the following Theorem:

Theorem 4. *Let λ be the statistical security parameter and $\delta > \Delta$. Then the protocol $\pi_{\mathsf{mpc,oia}}$ GUC-securely implements the ticked functionality $\mathcal{F}^{\Delta,\delta}_{\mathsf{mpc,oia}}$ in the $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}, \mathcal{F}^{\Delta,\delta}_{\mathsf{com}}, \mathcal{F}^{\Delta}_{\mathsf{ct}}, \mathcal{F}^{\Delta}_{\mathsf{BC,delay}}$-hybrid model against any static adversary corrupting up to $n-1$ of the n parties in $\mathcal{P}$. The transcripts are statistically indistinguishable.*

Output-Independent Abort with Semi-Synchronous Broadcast. Our solution to achieve output-independent abort crucially requires that all honest parties obtain the broadcast messages in the same round. Otherwise they will not be able to agree upon the abort event in the **Output** phase. It is an interesting question if that can be generalized to semi-synchronous broadcast.

One naive solution which requires $O(n \cdot |BA|)$ rounds and where $|BA|$ is the number of rounds for binary byzantine agreement, works as follows: Each $\mathcal{P}_i \in \mathcal{P}$ sends a TLP with delay $(n-i)\cdot(\Delta+\zeta)$ after running agreement that the previous TLP was received by all, where Δ is the maximal delay of the broadcast and ζ is the maximal agreement delay.

Clearly, this resolves the problem of honest parties agreeing on abort, but at the same time requires an honest majority for the byzantine agreement. We leave a more detailed analysis of this as interesting future work.

6.3 Penalizing Cheaters

We will now outline how the idea behind the protocol $\pi_{\mathsf{mpc,oia}}$ can be modified in order to construct MPC with punishable output-independent abort.

First, in order to manage monetary contributions of parties, we will use a smart contract functionality $\mathcal{F}^{\gamma,\delta}_{\mathsf{SC}}$ that accepts deposits of parties and distributes these to parties that do not cheat. As this smart contract will have to act upon messages sent by all parties, we will let $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ replace the broadcast functionality $\mathcal{F}_{\mathsf{BC,delay}}^{\Delta}$. This has the additional advantage that it easily synchronizes honest parties and $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ concerning the abort condition that no DOpen message can be accepted anymore. This is important, as we require that both the honest parties and $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ identify the same set of cheaters J_1, which $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ can then punish.

In order to identify the identical set J_2, $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ must be able to check openings of $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta}$. For this reason we will enhance this functionality to $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta}$ which has verifiability for opened values. Moreover, we ask that the opened values of $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta}$ have to be transferable: if $\mathcal{P}_i$ has found the opening, then by sending it to $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ the smart contract should be able to verify the opening (possibly with less computational overhead).

In the full protocol, parties will then first compute on their inputs and generate shares of the outputs as in $\pi_{\mathsf{mpc,poia}}$, although using the aforementioned different functionalities. Then, before starting the opening phase, each party will send the deposit to $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$. Here all these deposits have to arrive within a δ tick time span. Then, parties start the delayed openings as before, although the timeout to $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ is now longer ($2\delta + \gamma$ instead of 2δ). This is because we now require that honest parties, once they find an opening to (possibly adversarial) commitments, post these to $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$, which may take additional γ time to complete. Due to the length of the time span, all commitments from parties in $\mathcal{P} \setminus J_1$ will have been posted at that time, so that the set J_2 is identical for honest parties and $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$. At the same time, as $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta}$ is a commitment functionality the adversary cannot send "incorrect" openings for commitments of honest parties to $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$. We refer to Supplementary Material D for the concrete construction.

Penalizing Non-Optimistic Protocol Runs. An interesting optimization would be an optimistic version of $\pi_{\mathsf{mpc,poia}}$. There, each party $\mathcal{P}_i$, after having determined the set of parties that do not open the commitments to their output shares, would itself post the opening to its own commitment on $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$. This opening can then be verified by $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ and the other parties. Now a dishonest party $\mathcal{P}_j$ could always refrain from doing this and this would not be considered per se as cheating, but honest parties would in such a setting have to invest computational resources to perform the delayed opening of $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta,j}$. $\mathcal{P}_j$ has no incentive to follow the optimistic path of the protocol.

As $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ could in this case observe which verifiable openings are posted on time (and which had to be generated by other parties and are therefore posted only later), it can reimburse honest parties for this extra work. This incentivizes $\mathcal{A}$ not just to provide outputs to the honest parties, but also to do this without them having to work extra in order to solve TLPs. We leave a concrete construction as future work.

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 multiparty computations on bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 443–458. IEEE Computer Society Press, May 2014.
3. 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.
4. C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas. Bitcoin as a transaction ledger: A composable treatment. In J. Katz and H. Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 324–356. Springer, Heidelberg, Aug. 2017.
5. C. Baum, B. David, and R. Dowsley. Insured mpc: Efficient secure computation with financial penalties. In *Financial Cryptography 2020*, 2020.
6. C. Baum, B. David, R. Dowsley, J. B. Nielsen, and S. Oechsner. Tardis: Time and relative delays in simulation. Cryptology ePrint Archive, Report 2020/537, 2020. https://eprint.iacr.org/2020/537.
7. R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-homomorphic encryption and multiparty computation. In K. G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 169–188. Springer, Heidelberg, May 2011.
8. 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.
9. 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.
10. 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.
11. D. Boneh and M. Naor. Timed commitments. In M. Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 236–254. Springer, Heidelberg, Aug. 2000.
12. 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.
13. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, Oct. 2001.
14. 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.
15. I. Cascudo and B. David. SCRAPE: Scalable randomness attested by public entities. In D. Gollmann, A. Miyaji, and H. Kikuchi, editors, *ACNS 17*, volume 10355 of *LNCS*, pages 537–556. Springer, Heidelberg, July 2017.

16. 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.
17. 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.
18. 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.
19. 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.
20. 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.
21. A. Kiayias, A. Russell, B. David, and R. Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In J. Katz and H. Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 357–388. Springer, Heidelberg, Aug. 2017.
22. 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.
23. K. Pietrzak. Simple verifiable delay functions. In A. Blum, editor, *ITCS 2019*, volume 124, pages 60:1–60:15. LIPIcs, Jan. 2019.
24. randao.org. RANDAO: Verifiable random number generation, 2017. https://www.randao.org/whitepaper/Randao_v0.85_en.pdf accessed on 20/02/2020.
25. R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. 1996.
26. L. Rotem, G. Segev, and I. Shahaf. Generic-group delay functions require hidden-order groups. In *Eurocrypt 2020*, 2020. https://eprint.iacr.org/2020/225.
27. 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.
28. 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.

Supplementary Material

A Additional Functionalities

In our work, we need two additional functionalities which we deferred to this Supplementary Material, namely one for coin tossing and for a restricted observable and programmable random oracle. Those are given in Fig. 17 and Fig. 18.

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

The ticked functionality $\mathcal{F}_{\mathsf{ct}}^{\Delta}$ interacts with the n parties $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ and an adversary $\mathcal{S}$.

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

1. Send m with prefix Toss to the parties $\mathcal{P} \setminus \{\mathcal{P}_i\}$ via $\mathcal{Q}$ with delay Δ.
2. Send m and the IDs to $\mathcal{S}$.
3. If all parties sent $(\mathsf{Toss}, \mathsf{sid}, m)$:
   (a) Uniformly sample m random elements $x_1, \ldots, x_m \overset{\$}{\leftarrow} \mathbb{F}$ and send $(\mathsf{Tossed}, \mathsf{sid}, m, \mathbb{F}, x_1, \ldots, x_m)$ to $\mathcal{S}$.
   (b) If $\mathcal{S}$ sends $(\mathsf{DeliverCoins}, \mathsf{sid})$ then send the message $x_1, \ldots, x_m$ with prefix Coins to the parties $\mathcal{P}$ via $\mathcal{Q}$ with delay Δ. Otherwise send the message $\perp$ with prefix Coins to the parties $\mathcal{P}$ via $\mathcal{Q}$ with delay Δ.
   (c) Notify $\mathcal{S}$ about the message with prefix Coins.

Tick:

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

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

– If $(\mathsf{Deliver}, \mathsf{sid}, \mathsf{mid}) \in \mathcal{D}$ then remove each $(\mathtt{cnt}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$ from $\mathcal{Q}$ and add $(\mathcal{P}_i, \mathsf{sid}, m)$ to $\mathcal{M}$.
– If $(\mathsf{Abort}, \mathsf{sid}) \in \mathcal{D}$ and $\mathsf{open} = 0$ then add $(\mathcal{P}_i, \mathsf{sid}, \mathsf{Abort})$ to $\mathcal{M}$ for each $i \in [n]$ and ignore all further messages with this sid except to **Fetch Message**.

Fig. 17. Functionality $\mathcal{F}_{\mathsf{ct}}^{\Delta}$ for Multiparty Coin Tossing.

A.1 Modeling Rivest *et al.*'s Time-Lock Assumption [25]

We describe in Fig. 19 the ideal functionality $\mathcal{F}_{\mathsf{rsw}}$ from [6] that captures the hardness assumption used by Rivest et al. [25] to build a time-lock puzzle protocol. Later on, we will use this functionality as setup for realizing UC-secure publicly verifiable TLPs. Essentially, this functionality treats group $(\mathbb{Z}/N\mathbb{Z})^{\times}$ as in the generic group model [27], giving unique handles to the group elements (but not their descriptions) to all parties. In order to perform group operations, the parties interact with the functionality but only receive the result of the operation (*i.e.* the handle of the resulting group element) after the next computational tick occurs. As pointed out in [6], this definition of $\mathcal{F}_{\mathsf{rsw}}$ is corroborated by a recent result [26] showing that delay functions (such as a TLP) based on cyclic groups that do not exploit any particular property of the underlying group cannot be constructed if the order is known. Moreover, we cannot reveal the group structure to the environment, since it could use it across multiple sessions to solve TLPs quicker than the regular parties.

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 $(\textsc{Hash-Query}, m)$ 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 $h \overset{\$}{\leftarrow} \{0,1\}^{\ell(\tau)}$ 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 $(\textsc{Hash-Confirm}, h)$ to the caller.

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

Program: On input $(\textsc{Program-RO}, m, h)$ 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 $(\textsc{Program-Confirm})$ to $\mathcal{S}$.

IsProgrammed: On input $(\textsc{IsProgrammed}, m)$ 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 $(\textsc{IsProgrammed}, b)$ to the caller.

Fig. 18. Restricted observable and programmable global random oracle functionality $\mathcal{G}_{\mathsf{rpoRO}}$ from [12].

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: Set $\mathsf{out} \leftarrow \mathsf{in}$ and $\mathsf{in} = \emptyset$.

Fig. 19. Functionality $\mathcal{F}_{\mathsf{rsw}}$ from [6] capturing the time lock assumption of [25].

B Additional Details of π_{tlp} Proof - Theorem 1

We reproduce verbatim the simulator from [6] in Figure 20 (Corrupted $\mathcal{P}_o$) and in Figure 21 (Honest $\mathcal{P}_o$). This simulator covers all interfaces of $\mathcal{F}_{\mathsf{tlp}}$ but the **Public Verification** one, which is simulated by following the exact steps of an honest party in π_{tlp} as argued in the proof of Theorem 1.

Simulator $\mathcal{S}$ for 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 $(\textsc{Observe}, \mathsf{sid})$ 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 $\mathsf{tag}\text{-}\overline{\mathsf{tag}}, \mathsf{el}\text{-}\mathsf{st}, \mathsf{omsg}$.

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 $(\textsc{Hash-Query}, (h_1|m|\mathtt{td}))$ from $\mathcal{A}$ to $\mathcal{G}_{\mathsf{rpoRO2}}$ for which there was a response $(\textsc{Hash-Confirm}, \mathtt{tag}_2)$.
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 $(\textsc{Hash-Query}, (\mathtt{el}_{\Gamma}|\mathtt{el}_{\Gamma}))$ from $\mathcal{A}$ to $\mathcal{G}_{\mathsf{rpoRO1}}$ for which there was a response $(\textsc{Hash-Confirm}, h_1)$.
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 $j \in \{0, \dots, \Gamma\}$, sample $\mathtt{st}_j \xleftarrow{\$} \{0,1\}^{\tau}$, add $(\mathtt{el}_j, \mathtt{st}_j)$ to $\mathsf{el}\text{-}\mathsf{st}$ 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 $\mathsf{tag}\text{-}\overline{\mathsf{tag}}$ 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 $\mathtt{st}_0, \dots, \mathtt{st}_{\Gamma}, \mathtt{tag}$.

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

– If there is $(\mathtt{el}, \mathtt{st}) \in \mathsf{el}\text{-}\mathsf{st}$, 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 $(\mathtt{el}, \mathtt{st}) \in \mathsf{el}\text{-}\mathsf{st}$, 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 $\mathsf{el}\text{-}\mathsf{st}$. 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 $\mathsf{el}\text{-}\mathsf{st}$ and $(\mathtt{tag}, \overline{\mathtt{tag}})$ in $\mathsf{tag}\text{-}\overline{\mathsf{tag}}$, 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 $(\textsc{Hash-Query}, (\mathtt{el}_0|\mathtt{el}))$ to $\mathcal{G}_{\mathsf{rpoRO1}}$, obtain$(\textsc{Hash-Confirm}, h_1)$, compute $(m|\mathtt{td}) = \mathtt{tag}_1 \oplus h_1$, send $(\textsc{Hash-Query}, (h_1|m|\mathtt{td}))$ to $\mathcal{G}_{\mathsf{rpoRO2}}$, obtain $(\textsc{Hash-Confirm}, h_2)$, 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 $\mathcal{F}_{\mathsf{tlp}}$.

Fig. 20. Simulator $\mathcal{S}$ for the case of a corrupted $\mathcal{P}_o$ in π_{tlp} from [6].

Simulator $\mathcal{S}$ for 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 $(\textsc{IsProgrammed}, m)$ to $\mathcal{G}_{\mathsf{rpoRO1}}$ or $\mathcal{G}_{\mathsf{rpoRO2}}$, $\mathcal{S}$ always answers with $(\textsc{IsProgrammed}, 0)$. $\mathcal{S}$ 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 **Get Message**. However, for every query $(\textsc{IsProgrammed}, m)$ to $\mathcal{G}_{\mathsf{rpoRO1}}$ or $\mathcal{G}_{\mathsf{rpoRO2}}$, $\mathcal{S}$ always answers with $(\textsc{IsProgrammed}, 0)$.

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 **Get Message** 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 **Get Message** 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 $(\textsc{Program-RO}, (\mathtt{el}_0|\mathtt{el}_{\Gamma}), h_1)$ 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 $(\textsc{Program-RO}, (h_1|m|\mathtt{td})), h_2)$ 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. 21. Simulator $\mathcal{S}$ for the case of an honest $\mathcal{P}_o$ in π_{tlp} from [6].

C Proofs for Theorems 2 and 3 from Section 5

We present the proofs for Theorems 2 and 3 from Section 5 below.

C.1 Proofs for Theorems 2

In order to prove this theorem, we construct a simulator $\mathcal{S}$ that interacts with an internal copy $\mathcal{A}$ of the adversary simulating $\mathcal{F}_{\mathsf{tlp}}$ and $\mathcal{F}^{\Delta}_{\mathsf{BC,delay}}$ towards $\mathcal{A}$. For any environment $\mathcal{Z}$, we show that an execution with $\mathcal{S}$ and $\mathcal{F}_{\mathsf{RB}}$ is indistinguishable from an execution with $\mathcal{A}$ in the $\mathcal{F}_{\mathsf{tlp}}, \mathcal{F}^{\Delta}_{\mathsf{BC,delay}}$-hybrid model.

First of all, we observe that there exists such $\delta > \Delta$ since Δ is finite. Without loss of generality, in the remainder of this proof we assume that the parties in $\mathcal{P}$ receive their $(\textsc{Toss}, \mathsf{sid})$ inputs and start the **Commitment Phase** at the same time (*i.e.* at the same tick). However, notice that, if this is not the case and there's a delay of δ_{act} ticks between the first party in $\mathcal{P}$ receiving $(\textsc{Toss}, \mathsf{sid})$ and the last party in $\mathcal{P}$ receiving this input, we can adjust for that by increasing the delay parameter δ by δ_{act} ticks, which makes sure that the last party's message $(\mathcal{P}_j, \mathsf{sid}, \mathtt{puz}_j)$ is received by all the other parties in $\mathcal{P}$ before the first party's TLP is solved.

We focus on constructing $\mathcal{S}$ for the worst case where t of the parties are corrupted by $\mathcal{A}$. In this case, $\mathcal{S}$ proceeds in the **Commitment Phase** by executing the exact instructions of an honest party in $\pi_{\mathsf{TLP-RB}}$. Notice that this will ensure that a simulated party is in the set $\mathcal{C}$ and that the protocol proceeds to the **Opening Phase**, since only $t+1$ TLPs must be received before proceeding and we are guaranteed that this happens because at least $t+1$ parties are not corrupted. We denote one simulated honest party in $\mathcal{C}$ by $\mathcal{P}_h$ and $\mathcal{S}$ will use it to force the output of the protocol to be equal to that of $\mathcal{F}_{\mathsf{RB}}$. After the **Commitment Phase** is complete, $\mathcal{S}$ waits for $(\mathcal{P}_i, \mathsf{sid}, x)$ from $\mathcal{F}_{\mathsf{RB}}$. $\mathcal{S}$ executes the rest of the steps of an honest party in $\pi_{\mathsf{TLP-RB}}$ for the simulated parties in $\mathcal{C}$ with the following exceptions:

- For each $\mathcal{P}_j \in \mathcal{C}$, $\mathcal{S}$ checks that the TLP $\mathtt{puz}_j$ in $(\mathcal{P}_j, \mathsf{sid}, \mathtt{puz}_j)$ broadcast by $\mathcal{P}_j$ is valid according to $\mathcal{F}_{\mathsf{tlp}}$ and extracts all r_j values from the valid $\mathtt{puz}_j$, obtaining a set $\mathcal{C}_{\mathcal{V}}$ of parties $\mathcal{P}_j$ that broadcast a valid r_j, which will be either opened in the **Opening Phase** or recovered in the **Recovery Phase**.
- $\mathcal{S}$ sends $(\textsc{Toss}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{RB}}$ on behalf of each corrupted party $\mathcal{P}_j \in \mathcal{C}'$ that broadcast a valid TLP.
- $\mathcal{S}$ equivocates the opening of $\mathtt{puz}_h$ from $\mathcal{P}_h$ in the **Opening Phase** so that it opens to a value r' such that $r' \oplus_{\{j|\mathcal{P}_j \in \mathcal{C}_{\mathcal{V}} \setminus \mathcal{P}_h\}} = x$.

After the simulated execution of $\pi_{\mathsf{TLP-RB}}$ is complete and an $r = x$ is obtained, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs and halts.

Notice that the simulated opening of $\mathtt{puz}_h$ to r' is distributed exactly as in a real world execution of $\pi_{\mathsf{TLP-RB}}$ and that $\mathcal{A}$ obtains the same output x given by $\mathcal{F}_{\mathsf{RB}}$. This holds since only the valid TLPs $\mathtt{puz}_j$ are considered in computing the final output and since r' is computed by $\mathcal{S}$ based on the extracted r_j from the valid TLPs. Moreover, $\mathcal{S}$ sends $(\textsc{Toss}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{RB}}$ for each of the corrupted parties that participated in the simulated execution correctly. Hence, an execution with $\mathcal{S}$ and $\mathcal{F}_{\mathsf{RB}}$ is indistinguishable from an execution with $\mathcal{A}$ in the $\mathcal{F}_{\mathsf{tlp}}, \mathcal{F}^{\Delta}_{\mathsf{BC,delay}}$-hybrid model.

C.2 Proofs for Theorems 3

The proof of this theorem follows from the proof of Theorem 2 by observing the our choice of δ ensures that similar conditions to those of $\pi_{\mathsf{TLP-RB}}$ are also maintained in the end of the **Commitment Phase** in $\pi_{\mathsf{TLP-RB-LEDGER}}$, allowing us to use the same simulation strategy. We observe that there exists such $\delta > \mathsf{maxTXDelay} + \mathsf{emptyBlocks} \cdot \mathsf{slackWindow}$ since $\mathsf{maxTXDelay}, \mathsf{emptyBlocks}$, $\mathsf{slackWindow}$ are finite. Without loss of generality, in the remainder of this proof we assume that the parties in $\mathcal{P}$ are already registered to $\mathcal{F}_{\mathsf{Ledger}}$ and synchronized with respect to $\mathcal{F}_{\mathsf{Ledger}}$ when they receive their $(\textsc{Toss}, \mathsf{sid})$ inputs and that they start the **Commitment Phase** at the same time (*i.e.* at the same tick). However, notice that, if this is not the case and there's a delay of δ_{act} ticks between the first party in $\mathcal{P}$ receiving $(\textsc{Toss}, \mathsf{sid})$ and the last party in $\mathcal{P}$ receiving this input, we can adjust for that by increasing the delay parameter δ by δ_{act} ticks, which makes sure that the last party's message $(\mathcal{P}_j, \mathsf{sid}, \mathsf{puz}_j)$ is received by all the other parties in $\mathcal{P}$ before the first party's TLP is solved. Since we guarantee this condition, we can use the same simulator $\mathcal{S}$ with the difference that it simulates $\mathcal{F}_{\mathsf{Ledger}}$ towards an internal copy $\mathcal{A}$ of the adversary by following the exact instructions of $\mathcal{F}_{\mathsf{Ledger}}$ and executing all the queries to $\mathcal{F}_{\mathsf{Ledger}}$ by $\mathcal{A}$. Hence, we argue that an execution with $\mathcal{S}$ and $\mathcal{F}_{\mathsf{RB}}$ is indistinguishable from an execution of $\pi_{\mathsf{TLP-RB-LEDGER}}$ by $\mathcal{A}$ composed with $\mathcal{F}_{\mathsf{tlp}}, \mathcal{F}_{\mathsf{Ledger}}$.

D MPC with Punishable Output-Independent Aborts, continued

In this Supplementary Material Section, we will first give a full proof of Theorem 4. Then, we show how to extend $\pi_{\mathsf{mpc,oia}}$ from Section 6 to financially punish cheaters. This will be done using a smart contract functionality $\mathcal{F}^{\gamma,\delta}_{\mathsf{SC}}$ as well as a multi-party publicly verifiable delayed commitment $\mathcal{F}^{\gamma,\delta}_{\mathsf{vcom}}$, both of which we will now introduce. The protocol then implements a modification of the previous functionality, which we call $\mathcal{F}^{\Delta,\gamma,\delta}_{\mathsf{mpc,poia}}$.

D.1 Proof of Theorem 4

To prove security, we will construct a PPT simulator $\mathcal{S}$ and then argue indistinguishability of the transcripts of $\pi_{\mathsf{mpc,oia}} \circ \mathcal{A}$ and $\mathcal{F}^{\Delta,\delta}_{\mathsf{mpc,oia}} \circ \mathcal{S}$.

Proof. The simulator will, towards the dishonest parties I that are corrupted by $\mathcal{A}$, simulate honest parties while additionally interacting with $\mathcal{F}^{\Delta,\delta}_{\mathsf{mpc,oia}}$. $\mathcal{S}$ will furthermore simulate the hybrid functionalities $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}, \mathcal{F}^{\Delta,\delta}_{\mathsf{com}} i, \mathcal{F}^{\Delta}_{\mathsf{ct}}$ and $\mathcal{F}^{\Delta}_{\mathsf{BC,delay}} i$ towards $\mathcal{A}$. $\mathcal{S}$ forwards the messages from the hybrid functionalities and $\mathcal{A}$ to $\mathcal{G}_{\mathsf{ticker}}$ honestly.

Init: If an honest party sends $(\mathsf{Init}, \mathsf{sid}, C)$ to $\mathcal{F}^{\Delta,\delta}_{\mathsf{mpc,oia}}$ then $\mathcal{S}$ gets informed by the functionality. It will then simulate sending the same message to $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$. Similarly, if a dishonest party inputs such a message into $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$ then $\mathcal{S}$ will forward this to $\mathcal{F}^{\Delta,\delta}_{\mathsf{mpc,oia}}$. If $\mathcal{A}$ decides to reschedule the arrival of such a message to any of the honest parties in $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$, then $\mathcal{S}$ will forward this to $\mathcal{F}^{\Delta,\delta}_{\mathsf{mpc,oia}}$.

Input: The simulator behaves as during **Init**. For any honest party that provides an input into $\mathcal{F}^{\Delta,\delta}_{\mathsf{mpc,oia}}$ it inputs a dummy value into $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$. For every dishonest party $\mathcal{P}_i$ that provides an input to $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$ it observes that value x_i as $\mathcal{S}$ simulates the functionality and then sends x_i in the name of $\mathcal{P}_i$ to $\mathcal{F}^{\Delta,\delta}_{\mathsf{mpc,oia}}$.

Computation: $\mathcal{S}$ will follow the same strategy as during **Init**.

Share: $\mathcal{S}$ simulates the protocol $\pi_{\mathsf{mpc,oia}}$ as follows:

1. For each correct message $\mathsf{ShareOutput}$ by a dishonest party, send $(\mathsf{Share}, \mathsf{sid})$ to $\mathcal{F}^{\Delta,\delta}_{\mathsf{mpc,oia}}$ in the name of that party. For each message $(\mathsf{Share}, \mathsf{sid})$ by an honest party through $\mathcal{F}^{\Delta,\delta}_{\mathsf{mpc,oia}}$ let that party follow the protocol $\pi_{\mathsf{mpc,oia}}$.
2. In step 4 the simulated honest parties commit to $\boldsymbol{y}_i, \boldsymbol{r}_i$ using $\mathcal{F}^{\Delta,\delta,i}_{\mathsf{com}}$ that were obtained from $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$. For each dishonest party $\mathcal{P}_j$, observe which values $\boldsymbol{y}_j, \boldsymbol{r}_j$ it commits to using $\mathcal{F}^{\Delta,\delta,j}_{\mathsf{com}}$. Set J_2 as the set of parties where $\boldsymbol{y}_j, \boldsymbol{r}_j$ are inconsistent with the outputs of the respective parties from $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$.
3. If the revealing of shares in Step 7 of the protocol succeeds, then $\mathcal{S}$ sends $\mathsf{DeliverShares}$ to $\mathcal{F}^{\Delta,\delta}_{\mathsf{mpc,oia}}$, otherwise it sends Abort. Observe that if $\mathcal{S}$ sends $\mathsf{DeliverShares}$ to $\mathcal{F}^{\Delta,\delta}_{\mathsf{mpc,oia}}$ then message delivery to the honest parties in $\mathcal{F}^{\Delta,\delta}_{\mathsf{mpc,oia}}$ will be synchronized with how the adversary delays the $\mathsf{DeliverShares}$ message in $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$. Abort will also be sent if $\mathcal{A}$ aborts $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$ or $\mathcal{F}^{\Delta}_{\mathsf{ct}}$.

Reveal: $\mathcal{S}$ simulates the protocol $\pi_{\mathsf{mpc,oia}}$ as follows:

1. If an honest party $\mathcal{P}_i$ in $\mathcal{F}^{\Delta,\delta}_{\mathsf{mpc,oia}}$ sends Reveal then broadcast the Send-message in the protocol simulation using $\mathcal{F}^{\Delta,i}_{\mathsf{BC,delay},}$. Any influence on arrival of the broadcast by $\mathcal{A}$ is forwarded to $\mathcal{F}^{\Delta,\delta}_{\mathsf{mpc,oia}}$. Conversely, if a dishonest party broadcasts Send then send a Reveal-message in the name of that party to $\mathcal{F}^{\Delta,\delta}_{\mathsf{mpc,oia}}$.
2. Once the first Send message arrives at all simulated honest parties (which means that each honest party in $\mathcal{F}^{\Delta,\delta}_{\mathsf{mpc,oia}}$ obtains its first Reveal message), let each simulated honest party $\mathcal{P}_i$ send DOpen to its instance $\mathcal{F}^{\Delta,\delta,i}_{\mathsf{com}}$ as in the protocol.
3. $\mathcal{S}$ now waits for δ ticks. When $\mathcal{A}$ sends the tick, it checks from which parties of $\mathcal{P}$ the honest parties did not yet obtain DOpen or for which

commitments they already received an opened value. Let J_1 be that set. Then it sets $J = J_1 \cup J_2$, forwards the tick to $\mathcal{G}_{\mathsf{ticker}}$ and responds with the appropriate $(\mathsf{Abort}, \mathsf{sid}, J)$ to $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta}$.
4. Upon obtaining the output $\boldsymbol{y}$ from $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta}$ $\mathcal{S}$ picks one simulated honest party $\mathcal{P}_i$ uniformly at random. It then reprograms $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta,i}$ to output $\boldsymbol{y}'_i, \boldsymbol{r}'_i$ such that $\boldsymbol{y}'_i = \boldsymbol{y} - \sum_{j\in[n]\setminus\{i\}} \boldsymbol{y}_j$ (where the $\boldsymbol{y}_j$ where committed by all other parties in $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta,j}$) and $\boldsymbol{r}'_i = \boldsymbol{t}_i - \mathbf{A}\boldsymbol{y}'_i$.
5. When the last $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta,j}$ of a party $\mathcal{P}_j \in I$ opens, $\mathcal{S}$ lets $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta}$ deliver the output to the honest parties.

First, the honest parties react upon their inputs from $\mathcal{Z}$ or send outputs to it at the same points of time both during the real protocol and the simulation. For the **Init, Input, Computation** phase that is clear, and aborts are also carried to $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta}$ during **Share** at the same time. Similarly, actions that honest parties take towards $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta}$ lead to equivalent actions in the simulation that can be observed by $\mathcal{A}$ in **Init, Input, Computation, Share**. In **Reveal** they do not have any input from $\mathcal{Z}$, so we have to consider the output that they provide in both cases.

Both in $\mathcal{S}$ and in the real protocol, $\mathcal{A}$ will always get the correct output of the computation. It will also always get messages from the (simulated) honest parties with the same distribution: we only reprogram one commitment $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta,i}$ in $\mathcal{S}$ but this is indistinguishable due to the random choice of $\boldsymbol{r}_i$ that hides the committed share perfectly.

The simulated honest parties in the simulation will always abort if they would get the wrong output of the computation, due to the choice of J: J_1 is determined identically in both the simulation and the real protocol, but J_2 is computed differently and it is computed in the simulation according to incorrect output shares $\boldsymbol{y}_j$ of dishonest parties.

Due to the choice of $\mathbf{A} \in \mathbb{F}^{\lambda\times m}$ we know that $f_{\mathbf{A}}(\boldsymbol{y}, \boldsymbol{r}) := \boldsymbol{r} + \mathbf{A}\boldsymbol{y}$ is a universal hash function, which implies that J_1 differs between $\mathcal{S}$ and $\pi_{\mathsf{mpc,oia}}$ only when $\mathcal{A}$ in the real protocol commits to values $\boldsymbol{y}'_j, \boldsymbol{r}'_j$ such that $f_{\mathbf{A}}(\boldsymbol{y}_j, \boldsymbol{r}_j) = \boldsymbol{t}_j = f_{\mathbf{A}}(\boldsymbol{y}'_j, \boldsymbol{r}'_j)$, which it has to do before $\mathbf{A}$ is known. By the properties of a universal hash function, we then have that $\boldsymbol{t}_j \neq f_{\mathbf{A}}(\boldsymbol{y}'_j, \boldsymbol{r}'_j)$ except with probability that is negligible in λ.

Now if the honest parties do not output y then they output the set J. Honest parties in the protocol will simply output J while those in the ideal setting only output J to $\mathcal{Z}$ that does not contain any honest parties. But because $\delta > \Delta$ all honest messages **DOpen** will always arrive before $\mathsf{DOpened}$ occurs at an honest party, so J_1 never contains an honest party in either the simulation or real protocol. □

D.2 Commitments with Publicly Verifiable Delayed Openings

In Fig. 22 and Fig. 23 we describe the functionality $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta}$ for commitments with publicly verifiable delayed non-interactive openings. The functionality distinguishes between a sender, which is allowed to make commitments, a set of

Functionality $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta}$ (Commit, Opening)

The ticked functionality is parameterized by $\gamma, \delta \in \mathbb{N}$ and interacts with a set of verifiers $\mathcal{V}$ and a set of n parties $\mathcal{P} = \{\mathcal{P}_1, \dots, \mathcal{P}_n\}$ where $\mathcal{P}_{\mathsf{Send}} \in \mathcal{P}$ is a special party called "the sender" and $\mathcal{P}_{\mathsf{Rec}} = \mathcal{P} \setminus \{\mathcal{P}_{\mathsf{Send}}\}$ are the receivers. An adversary $\mathcal{S}$ may corrupt a strict subset $I \subset \mathcal{P}$ of parties. The functionality internally has an initially empty list $\mathcal{O}$ and a map $\mathsf{commits}$.

Commit: Upon receiving $(\mathsf{Commit}, \mathsf{sid}, \mathsf{cid}, x)$ from $\mathcal{P}_{\mathsf{Send}}$ where cid is an unused identifier and x is a bit-string proceed as follows:

1. Set $\mathsf{commits}[\mathsf{cid}] = (x, \mathsf{vt})$ where $\mathsf{vt} \xleftarrow{\$} \{0,1\}^{\tau}$.
2. Send a message cid with prefix Commit to $\mathcal{P}_{\mathsf{Rec}} \cup \mathcal{V}$ via $\mathcal{Q}$ with delay γ.
3. Send cid and the IDs to $\mathcal{S}$.

Open: Upon receiving $(\mathsf{Open}, \mathsf{sid}, \mathsf{cid})$ from $\mathcal{P}_{\mathsf{Send}}$, if $\mathsf{commits}[\mathsf{cid}] = (x, \mathsf{vt}) \neq \perp$ then proceed as follows:

1. Send message $(\mathsf{cid}, x, \mathsf{vt})$ with prefix Open to $\mathcal{P}_{\mathsf{Rec}}$ via $\mathcal{Q}$ with delay γ.
2. Send $(\mathsf{cid}, x, \mathsf{vt})$ and the IDs to $\mathcal{S}$.

Delayed Open: Upon receiving $(\mathsf{DOpen}, \mathsf{sid}, \mathsf{cid})$ from $\mathcal{P}_{\mathsf{Send}}$, if $\mathsf{commits}[\mathsf{cid}] = (x, \mathsf{vt}) \neq \perp$ then proceed as follows:

1. Simultaneously send message cid with prefix DOpen to all parties in $\mathcal{P}_{\mathsf{Rec}}$ via $\mathcal{Q}$ with delay γ.
2. Add $(\delta, \mathsf{sid}, \mathcal{P}_j, (\mathsf{cid}, x, \mathsf{vt}))$ for each $\mathcal{P}_j \in \mathcal{P}_{\mathsf{Rec}}$ and $(\delta, \mathsf{sid}, \mathcal{S}, (\mathsf{cid}, x, \mathsf{vt}))$ to $\mathcal{O}$.
3. Send cid and the ID to $\mathcal{S}$.

Public Verification: Upon receiving $(\mathsf{Verify}, \mathsf{sid}, (\mathsf{cid}, x, \mathsf{vt}))$ from $\mathcal{V}_i \in \mathcal{V}$, if $\mathsf{commits}[\mathsf{cid}] = (x, \mathsf{vt})$, set $b = 1$, else set $b = 0$. Output $(\mathsf{Verified}, \mathsf{sid}, (\mathsf{cid}, x, \mathsf{vt}), b)$ to $\mathcal{V}_i$.

Fig. 22. Ticked Functionality $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta}$ For Multiparty Commitments with Verifiable Delayed Opening.

receivers, which will obtain the openings, and a set of verifiers, which will be able to verify that a claimed opening is indeed correct. For **Public Verification** any verifier $\mathcal{V}_i \in \mathcal{V}$ (which does not have to be part of $\mathcal{P}_{\mathsf{Rec}}$) can check whether a certain opening for a commitment cid is indeed valid. This allows parties from $\mathcal{P}_{\mathsf{Rec}}$ to "verifiably transfer" openings to other parties. The string vt_i makes it computationally infeasible for any $\mathcal{V}_i \in \mathcal{V}$ to simply brute-force the value.

We construct a protocol π_{vcom} realizing $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta}$ by combining a standard random oracle-based commitment with a TLP. The core of the protocol is having the sender commit to a message m by sampling some randomness r and broadcasting the commitment c obtained from the random oracle being queried on $(m|r)$, which is revealed later in the opening phase so that the receivers can repeat the query to verify that the output matches the previously received c. This basic scheme can be augmented with a delayed opening procedure by simply generating a TLP containing $(m|r)$ that can be solved in δ steps, so that receivers only learn the message (and verification information) for the commitment after the desired delay δ. In order to make this scheme publicly verifiable,

Functionality $\mathcal{F}^{\gamma,\delta}_{\mathsf{vcom}}$ (Message Handling)

Tick:

1. Remove each $(0, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$ from $\mathcal{Q}$ and instead add $(\mathcal{P}_i, \mathsf{sid}, m)$ to $\mathcal{M}$.
2. Replace each $(\overline{\mathtt{cnt}}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$ in $\mathcal{Q}$ with $(\mathtt{cnt} - 1, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$.
3. For each entry $(\mathtt{cnt}, \mathsf{sid}, \mathcal{P}_j, (\mathsf{cid}, x, \mathsf{vt})) \in \mathcal{O}$ with $\mathcal{P}_j \in \mathcal{P}_{\mathsf{Rec}}$, if there is no entry $(\mathtt{cnt}, \mathsf{sid}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_j, (\mathsf{DOpen}, \mathsf{cid})) \in \mathcal{Q}$, proceed as follows:
   - If $\mathtt{cnt} = 0$, append $(\mathcal{P}_j, \mathsf{sid}, (\mathsf{DOpened}, (\mathsf{cid}, x, \mathsf{vt})))$ to $\mathcal{M}$.
   - If $\mathtt{cnt} > 0$, replace $(\mathtt{cnt}, \mathsf{sid}, \mathcal{P}_j, (\mathsf{cid}, x, \mathsf{vt}))$ with $(\mathtt{cnt} - 1, \mathsf{sid}, \mathcal{P}_j, (\mathsf{cid}, x, \mathsf{vt}))$ in $\mathcal{O}$.
4. For each entry $(\mathtt{cnt}, \mathsf{sid}, \mathcal{S}, (\mathsf{cid}, x, \mathsf{vt})) \in \mathcal{O}$, proceed as follows:
   - If $\mathtt{cnt} = 0$, append $(\mathcal{P}_{\mathsf{Send}}, \mathsf{sid}, (\mathsf{DOpened}, \mathsf{cid}))$ to $\mathcal{M}$ and output $(\mathsf{DOpen}, \mathsf{sid}, \mathcal{S}, (\mathsf{cid}, x, \mathsf{vt}))$ to $\mathcal{S}$.
   - If $\mathtt{cnt} > 0$, replace $(\mathtt{cnt}, \mathsf{sid}, \mathcal{S}, (\mathsf{cid}, x, \mathsf{vt}))$ with $(\mathtt{cnt} - 1, \mathsf{sid}, \mathcal{S}, (\mathsf{cid}, x, \mathsf{vt}))$ in $\mathcal{O}$.

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

- If $(\mathsf{Deliver}, \mathsf{sid}, \mathsf{mid}) \in \mathcal{D}$ then remove each $(\mathtt{cnt}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$ from $\mathcal{Q}$ and add $(\mathcal{P}_i, \mathsf{sid}, m)$ to $\mathcal{M}$.
- If $(\mathsf{Abort}, \mathsf{sid}) \in \mathcal{D}$ and $\mathsf{open} = 0$ then add $(\mathcal{P}_i, \mathsf{sid}, \mathsf{Abort})$ to $\mathcal{M}$ for each $i \in [n]$ and ignore all further messages with this sid except to **Fetch Message**.

Fig. 23. Ticked Functionality $\mathcal{F}^{\gamma,\delta}_{\mathsf{vcom}}$ For Multiparty Commitments with Verifiable Delayed Opening.

we use a bulletin board incorporated into the smart contract functionality $\mathcal{F}^{\gamma,\delta}_{\mathsf{SC}}$ and a global random oracle $\mathcal{G}_{\mathsf{rpoRO}}$, so that any verifier who joins the protocol execution at any point can retrieve commitments, openings and delayed openings from the bulletin board and verify them while obtaining the same results as the parties who participated in the execution so far.

Theorem 5. *Protocol π_{vcom} GUC-realizes $\mathcal{F}^{\gamma,\delta}_{\mathsf{vcom}}$ in the $\mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}^{\gamma,\delta}_{\mathsf{SC}}, \mathcal{F}_{\mathsf{tlp}}$ hybrid model.*

Proof. [Sketch] The fact that the **Commit** and **Open** steps of π_{vcom} realize the corresponding interfaces of the standard commitment functionality in the $\mathcal{G}_{\mathsf{rpoRO}}$ and $\mathcal{F}_{\mathsf{Auth}}$-hybrid model ($\mathcal{F}_{\mathsf{Auth}}$ is the functionality for authenticated channels) is proven in [12]. In our case $\mathcal{F}_{\mathsf{Auth}}$ is substituted by the authenticated bulletin board embedded in $\mathcal{F}^{\gamma,\delta}_{\mathsf{SC}}$ through which messages are sent among parties. We can further extend the simulator $\mathcal{S}$ from [12] to capture the delayed opening and public verification. The delayed opening can be simulated by equivocating the message contained in the simulated TLP with the one received from $\mathcal{F}^{\gamma,\delta}_{\mathsf{vcom}}$ in case $\mathcal{A}$ corrupts parties in $\mathcal{P}$ but not $\mathcal{P}_{\mathsf{Send}}$. In case $\mathcal{A}$ corrupts $\mathcal{P}_{\mathsf{Send}}$, the delayed opening can be simulated by extracting (x, r) from its TLP and checking that these values represent a valid opening, in which case $\mathcal{S}$ instructs $\mathcal{F}^{\gamma,\delta}_{\mathsf{vcom}}$ to start a delayed opening. $\mathcal{S}$ can do this since it simulates $\mathcal{F}_{\mathsf{tlp}}$ towards $\mathcal{A}$, similarly to the strategy of the delayed homomorphic commitment of [6]. Public verification follows in a straightforward manner since verifiers $\mathcal{V}$ receive the same messages as parties $\mathcal{P}$ and perform the exact same procedures of an honest receiver to

Protocol π_{vcom}

Protocol π_{vcom} is parameterized by an opening delay δ and operates with parties $\mathcal{P} = \{\mathcal{P}_1, \dots, \mathcal{P}_n\}$ and verifiers $\mathcal{V}$ that interact with each other and with $\mathcal{G}_{\mathsf{rpoRO}}$ (with output in $\{0,1\}^\tau$), $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ and $\mathcal{F}_{\mathsf{tlp}}$ as follows:

Commit: On input $(\mathsf{Commit}, \mathsf{sid}, \mathsf{cid}, x)$, $\mathcal{P}_{\mathsf{Send}}$ uniformly samples $r \overset{\$}{\leftarrow} \{0,1\}^\tau$ and queries $\mathcal{G}_{\mathsf{rpoRO}}$ on $(\mathsf{cid}, x|r)$ to obtain c. $\mathcal{P}_{\mathsf{Send}}$ sends $(\mathsf{Broadcast}, \mathsf{sid}, (\mathsf{cid}, c))$ to $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$. All parties $\mathcal{P}_j \in \mathcal{P}$ for $j \neq i$ output $(\mathsf{Committed}, \mathsf{sid}, \mathcal{P}_{\mathsf{Send}}, \mathsf{cid})$ upon receiving this message from $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$.

Open: On input $(\mathsf{Open}, \mathsf{sid}, \mathsf{cid})$, $\mathcal{P}_{\mathsf{Send}}$ sends $(\mathsf{Broadcast}, \mathsf{sid}, (\mathsf{cid}, x, r))$ to $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$. Upon receiving (cid, x, r) from $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$, each party $\mathcal{P}_j$ queries $\mathcal{G}_{\mathsf{rpoRO}}$ on $(\mathsf{cid}, x|r)$ and checks that the answer is equal to c and that this output is not programmed by sending $(\textsc{IsProgrammed}, \mathsf{cid}, x|r)$ to $\mathcal{G}_{\mathsf{rpoRO}}$, aborting if the answer is $(\textsc{IsProgrammed}, 1)$. Output $(\mathsf{Open}, \mathsf{sid}, \mathcal{P}_{\mathsf{Send}}, \mathsf{cid}, m)$.

Delayed Open: On input $(\mathsf{DOpen}, \mathsf{sid}, \mathsf{cid})$, $\mathcal{P}_{\mathsf{Send}}$ sends $(\mathsf{CreatePuzzle}, \mathsf{sid}, \delta, (x, r))$, receiving $(\mathsf{CreatedPuzzle}, \mathsf{sid}, \mathsf{puz} = (\mathtt{st}_0, \delta, \mathtt{tag}))$. $\mathcal{P}_{\mathsf{Send}}$ sends $(\mathsf{Broadcast}, \mathsf{sid}, (\mathsf{cid}, \mathsf{puz}))$ to $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$. Upon receiving $(\mathsf{cid}, \mathsf{puz})$ from $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ all parties $\mathcal{P}_i \in \mathcal{P}$ parse $\mathsf{puz} = (\mathtt{st}_0, \delta, \mathtt{tag})$ and solve it by performing one iteration of the following loop at every activation, where $\mathtt{cst} = 0$ in the beginning:

1. Send $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st}_{\mathtt{cst}})$ to $\mathcal{F}_{\mathsf{tlp}}$.
2. Send $(\mathsf{Fetch}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{tlp}}$ and check that there is an entry $(\mathcal{P}_i, \mathtt{st}_{\mathtt{cst}}, \mathtt{st})$ in L_i. If yes, increment $\mathtt{cst}$ and set $\mathtt{st}_{\mathtt{cst}} = \mathtt{st}$.
3. If $\mathtt{cst} = \delta$, $\mathcal{P}_i$ sends $(\mathsf{GetMsg}, \mathsf{sid}, \mathsf{puz}, \mathtt{st}_{\mathtt{cst}})$ to $\mathcal{F}_{\mathsf{tlp}}$, receiving $(\mathsf{GetMsg}, \mathsf{sid}, \mathtt{st}_0, \mathtt{tag}, \mathtt{st}_{\mathtt{cst}}, (x, r))$. $\mathcal{P}_i$ queries $\mathcal{G}_{\mathsf{rpoRO}}$ on $(\mathsf{cid}, x|r)$ and checks that the answer is equal to c and that the output is not programmed by sending $(\textsc{IsProgrammed}, \mathsf{cid}, x|r)$ to $\mathcal{G}_{\mathsf{rpoRO}}$. If any of these checks fail, $\mathcal{P}_i$ aborts. Otherwise, it sends $(\mathsf{Broadcast}, \mathsf{sid}, (\mathsf{cid}, \mathtt{st}, x, r))$ to $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$, outputs $(\mathsf{DOpened}, \mathsf{sid}, (\mathsf{cid}, x, r))$ and exits the loop.

$\mathcal{P}_{\mathsf{Send}}$ executes a similar loop after sending $(\mathsf{Broadcast}, \mathsf{sid}, (\mathsf{cid}, \mathsf{puz}))$ to $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ but when $\mathtt{cst} = \delta$ in Step 3, it sends $(\mathsf{Broadcast}, \mathsf{sid}, (\mathsf{cid}, \mathtt{st}, x, r))$ to $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ and outputs $(\mathsf{DOpened}, \mathsf{sid}, \mathsf{cid})$.

Verify: On input $(\mathsf{Verify}, \mathsf{sid}, (\mathsf{cid}, x, r))$, $\mathcal{V}_j \in \mathcal{V}$ sends $(\mathsf{Fetch} - \mathsf{BB}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$, receives $(\mathsf{Return} - \mathsf{BB}, \mathsf{sid}, \mathcal{B})$ and checks that there exists (cid, c) in $\mathcal{B}$. $\mathcal{V}_j$ queries $\mathcal{G}_{\mathsf{rpoRO}}$ on $(\mathsf{cid}, x|r)$ and checks that the answer is equal to c and that this output is not programmed by sending $(\textsc{IsProgrammed}, \mathsf{cid}, x|r)$ to $\mathcal{G}_{\mathsf{rpoRO}}$, checking that the answer is $(\textsc{IsProgrammed}, 0)$. Moreover, if there is $(\mathsf{cid}, \mathsf{puz}) \in \mathcal{B}$, $\mathcal{V}_j$ checks that there exists a valid $(\mathsf{cid}, \mathtt{st}, x', r') \in \mathcal{B}$ with respect to $\mathcal{F}_{\mathsf{tlp}}$ such that $(x', r') = (x, r)$. If any of these checks fail set $b = 0$, else set $b = 1$. Output $(\mathsf{Verified}, \mathsf{sid}, (\mathsf{cid}, x, r), b)$.

Fetching Messages: At every activation, all parties $\mathcal{P}$ and in $\mathcal{V}$ send $(\mathsf{Fetch}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$, receiving $(\mathsf{Fetch}, \mathsf{sid}, L)$ and parsing L according to the steps above.

Fig. 24. Protocol π_{vcom} for Multiparty Commitments with Verifiable Delayed Opening.

verify the validity of such messages. $\mathcal{S}$ simulates public verification towards $\mathcal{A}$ by also following the exact steps of honest parties. Notice that this would only fail if it was possible to find alternative openings x', r' for a commitment (cid, c), which only happens with negligible probability. Hence, since $\mathcal{G}_{\mathsf{rpoRO}}$ is global the output obtained by $\mathcal{V}$ in the public verification procedure is 1 if and only if the output x was really obtained from a valid opening of the commitment identified by cid. $\square$

D.3 The Smart Contract Functionality

Functionality $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ (Contract Code, Bulletin Board)

The ticked functionality runs with n parties $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ and an adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ is parameterized by the compensation amount q, the security deposit $d = (n-1)q$ and has a state $\mathtt{st}$ initially set to $\perp$ as well as a list $\mathcal{B}$.

Register: On first input $(\mathsf{Register}, \mathsf{sid}, \{\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta,j}\}_{j\in[n]})$ by $\mathcal{P}_i \in \mathcal{P}$:

1. Notify the parties $\mathcal{P} \setminus \{\mathcal{P}_i\}$ via $\mathcal{Q}$ with delay 0.
2. If each party sent $(\mathsf{Register}, \mathsf{sid}, \{\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta,j}\}_{j\in[n]})$ with the same functionalities then set $\mathtt{st} = \mathsf{ready}$, register to all functionalities as verifier and store references to all these functionalities.
3. Send $(\mathsf{Register}, \mathsf{sid}, \mathcal{P}_i, \{\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta,j}\}_{j\in[n]})$ to $\mathcal{S}$.

Broadcast: Upon receiving an input $(\mathsf{Broadcast}, \mathsf{sid}, m)$ from a party $\mathcal{P}_i \in \mathcal{P}$:

1. Simultaneously send message (m, i) to the parties $\mathcal{P}$ via $\mathcal{Q}$ with delay γ.
2. Send (m, i) and the ID to $\mathcal{S}$.

Fetch Bulletin Board: Upon receiving an input $(\mathsf{Fetch}-\mathsf{BB}, \mathsf{sid})$ from a party in $\mathcal{P}$ or $\mathcal{V}$, output $(\mathsf{Return}-\mathsf{BB}, \mathsf{sid}, \mathcal{B})$ to that party.

Deposit: On input $(\mathsf{Deposit}, \mathsf{sid}, \mathtt{coins}(d))$ by $\mathcal{P}_i \in \mathcal{P}$, if for each $\mathcal{P}_j \in \mathcal{P}$ there is the same $(\mathsf{sid}, (\mathsf{Broadcast}, (\mathbf{A}, \boldsymbol{t}_1, \ldots, \boldsymbol{t}_n, j))) \in \mathcal{B}$ and if $\mathtt{st} \in \{\mathsf{ready}, \mathsf{dep}(x)\}$:

1. Simultaneously send a message $(\text{“}\mathtt{coins}(d)\text{''}, i)$ with prefix $\mathsf{Deposit}$ to the parties $\mathcal{P} \setminus \{\mathcal{P}_i\}$ via $\mathcal{Q}$ with delay 0.
2. If $\mathtt{st} = \mathsf{ready}$ then set $\mathtt{st} = \mathsf{dep}(\gamma)$.
3. Send $(\mathsf{Deposit}, \mathsf{sid}, \text{“}\mathtt{coins}(d)\text{''}, i)$ to $\mathcal{S}$.

Fig. 25. Ticked Functionality $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ for Smart Contracts.

The smart contract functionality $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ is depicted in Fig. 25 and Fig. 26. It realizes the coin-handling parts of our protocol. At the same time, it serves in the protocol as a bulletin board (and therefore also broadcast) functionality and is a verifier to $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta}$. Therefore, our construction requires $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta}$ to be a global functionality. This hides details of the commitment verification from the smart contract.

At any point the parties will be able to use the bulletin board property of $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$, where $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ also keeps track about all messages that have been broadcast in an internal list $\mathcal{B}$. All such sent messages can at any point be retrieved using **Fetch Bulletin Board**.

Before being able to use $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ with respect to coins, the parties will have to register the instances of $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta}$ that they want to use. Once this is finished, they

Functionality $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ (Ticks)

```
Tick:
  1. Remove each $(0, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$ from $\mathcal{Q}$ and instead add $(\mathcal{P}_i, \mathsf{sid}, m)$ to $\mathcal{M}$
     and, if there is no other $(\mathtt{cnt}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_j, m) \in \mathcal{Q}$, add $(\mathsf{sid}, m)$ to $\mathcal{B}$.
  2. Replace each $(\mathtt{cnt}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$ in $\mathcal{Q}$ with $(\mathtt{cnt} - 1, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$.
  3. If $\mathtt{st} = \mathtt{wait}(x)$ and $x \geq 0$:
     (a) Set $\mathtt{st} = \mathtt{wait}(x - 1)$.
     (b) If $x = 0$:
         i. Let $L \subset \mathcal{P} \setminus J_1$ be the set of parties such that for $i \in L$ there exists
            $(\mathsf{cid}_i, \boldsymbol{y}_i, \boldsymbol{r}_i, \mathsf{vt}_i)$ in $\mathcal{B}$ such that public verification on $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta,i}$ outputs
            1. Set $J_2$ as the set of all parties $\mathcal{P}_i \in L$ such that $\boldsymbol{t}_i \neq \boldsymbol{r}_i + \mathbf{A}\boldsymbol{y}_i$.
            Set $J \leftarrow J_1 \cup J_2$. If $J = \emptyset$ then set $e_1, \ldots, e_n \leftarrow d$.
         ii. If instead $J \neq \emptyset$ then set $e_i \leftarrow d + |J| \cdot q$ for each party $\mathcal{P}_i \in \mathcal{P} \setminus J$
             and $e_i \leftarrow d - q \cdot (n - |J|)$ for each $\mathcal{P}_i \in J$.
         iii. Send message $\mathtt{coins}(e_i)$ to each party $\mathcal{P}_i \in \mathcal{P}$ via $\mathcal{M}$.
         iv. Set $\mathtt{st} = \bot$ and send $(\mathsf{Coins}, \mathsf{sid}, \{e_i\}_{\mathcal{P}_i \in \mathcal{P}})$ to $\mathcal{S}$.
     (c) If $x = \delta + \gamma$: Set $J_1$ as the set of parties $\mathcal{P}_j$ such that $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta,j}$ did not
         send $\mathsf{DOpen}$ to $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$.
  4. If $\mathtt{st} = \mathtt{dep}(x)$:
     If $x > 0$: Set $\mathtt{st} = \mathtt{dep}(x - 1)$.
     If $x = 0$: If all parties in $\mathcal{P}$ sent $(\mathsf{Deposit}, \mathsf{sid}, \mathtt{coins}(d))$ then set $\mathtt{st} =$
         $\mathtt{wait}(2\delta + \gamma)$ and send a message $\mathsf{AllDeposited}$ to $\mathcal{P}$ and $\mathcal{S}$ via $\mathcal{M}$.
         Otherwise send a message $\mathtt{coins}(d)$ with prefix $\mathsf{Coins}$ to each party that
         sent the deposit via $\mathcal{M}$, set $\mathtt{st} = \mathsf{ready}$ and send a message $\mathsf{Reimbursed}$
         to $\mathcal{P}$ and $\mathcal{S}$ via $\mathcal{M}$.
```

Fig. 26. Ticked Functionality $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ for Smart Contracts.

can then deposit coins to the functionality *if the protocol has actually shared the output to all parties* which is indicated by $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ having obtained $\mathbf{A}$. This way we avoid that the adversary can activate **Deposit** of $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ prematurely. All parties then, once **Deposit** is activated, have time γ to deposit their coins as well, otherwise these will automatically be returned by $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$. If all parties indeed deposited their coins then $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ will notify both the parties and $\mathcal{S}$ about this state change, which will allow them to react to this event by opening their instances of $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta}$. After this, no more coins can be deposited by any party.

Once the coins are locked, $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ will similarly to $\pi_{\mathsf{mpc,oia}}$ wait for the parties to initialize the opening of their commitments for δ ticks. Afterwards it will wait $\delta + \gamma$ ticks where parties in the protocol first obtain the committed values for each commitment (which takes δ ticks) which they then broadcast via $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ (which takes another γ ticks to succeed). $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ can then verify these openings using the respective instances of $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta}$. Honest parties will always succeed in doing this in the respective amount of time.

None of the actions done by $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ rely on any secret information or secret state and all messages that are provided by $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ are provided immediately. In an implementation, this can be implemented with a non-private smart contract.

D.4 MPC with Punishable Output-Independent Abort

Functionality $\mathcal{F}_{\mathsf{mpc,poia}}^{\Delta,\gamma,\delta}$ (MPC)

The ticked functionality runs with n parties $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ and an adversary $\mathcal{S}$ who may corrupt a strict subset $I \subset \mathcal{P}$. $\mathcal{F}_{\mathsf{mpc,poia}}^{\Delta,\gamma,\delta}$ is parameterized by $\Delta, \gamma, \delta \in \mathbb{N}^+$, the compensation amount q and the security deposit $d = (n-1)q$. The computed circuit is defined over $\mathbb{F}$. The functionality has a state $\texttt{st}$ that is initially $\perp$ as well as an initially empty set J.

Init: On first input $(\mathsf{Init}, \mathsf{sid}, C)$ by $\mathcal{P}_i \in \mathcal{P}$:
1. Send message C to the parties $\mathcal{P} \setminus \{\mathcal{P}_i\}$ via $\mathcal{Q}$ with delay Δ.
2. If each party sent $(\mathsf{Init}, \mathsf{sid}, C)$ then store C locally.
3. Send C and the IDs to $\mathcal{S}$.

Input: On first input $(\mathsf{Input}, \mathsf{sid}, i, x_i)$ by $\mathcal{P}_i \in \mathcal{P}$:
1. Notify parties $\mathcal{P} \setminus \{\mathcal{P}_i\}$ via $\mathcal{Q}$ with delay Δ.
2. Accept x_i as input for $\mathcal{P}_i$.
3. Send m and the IDs to $\mathcal{S}$ if $\mathcal{P}_i \in I$, otherwise notify $\mathcal{S}$ about a message with prefix Input.

Computation: On first input $(\mathsf{Compute}, \mathsf{sid})$ by $\mathcal{P}_i \in \mathcal{P}$ and if all $x_1, \ldots, x_n$ were accepted:
1. Notify parties $\mathcal{P} \setminus \{\mathcal{P}_i\}$ via $\mathcal{Q}$ with delay Δ.
2. If each party sent $(\mathsf{Compute}, \mathsf{sid})$ compute $y = C(x_1, \ldots, x_n)$ and store y.
3. Notify $\mathcal{S}$ about a message with prefix $\mathsf{Compute}$.

Share: On first input $(\mathsf{Share}, \mathsf{sid})$ by party $\mathcal{P}_i$, if y has been stored and if $\texttt{st} = \perp$:
1. Notify parties $\mathcal{P} \setminus \{\mathcal{P}_i\}$ via $\mathcal{Q}$ with delay Δ.
2. If all parties sent Share then:
   (a) Send $(\mathsf{Shares?}, \mathsf{sid})$ to $\mathcal{S}$.
   (b) Upon $(\mathsf{DeliverShares}, \mathsf{sid})$ from $\mathcal{S}$ simultaneously send a message with prefix $\mathsf{DeliverShares}$ to each $\mathcal{P}_j \in (\mathcal{P} \cup \mathcal{V}) \setminus I$ via $\mathcal{Q}$ with delay Δ. Then notify $\mathcal{S}$ about messages with prefix $\mathsf{DeliverShares}$ and the ID.
   (c) Otherwise, if $\mathcal{S}$ sends $(\mathsf{Abort}, \mathsf{sid})$ then send Abort to all parties
3. Notify $\mathcal{S}$ about a message with prefix Share.

Deposit: On first input $(\mathsf{Deposit}, \mathsf{sid}, \texttt{coins}(d))$ by $\mathcal{P}_i \in \mathcal{P}$, if **Share** finished, if no $\mathsf{DeliverShare}$ message is in $\mathcal{Q}$ and if $\texttt{st} \in \{\texttt{dep}, \perp\}$:
1. Simultaneously send a message $(i, \text{“}\texttt{coins}(d)''\,)$ to the parties $\mathcal{P} \setminus \{\mathcal{P}_i\}$ via $\mathcal{Q}$ with delay 0.
2. If $\texttt{st} = \perp$ then set $\texttt{st} = \texttt{dep}(\gamma)$.
3. Notify $\mathcal{S}$ about the message.

Fig. 27. Ticked Functionality $\mathcal{F}_{\mathsf{mpc,poia}}^{\Delta,\gamma,\delta}$ for Secure Multiparty Computation with Punishable Output-Independent Abort.

Finally, we now describe the functionality $\mathcal{F}_{\mathsf{mpc,poia}}^{\Delta,\gamma,\delta}$ which provides MPC with punishable output-independent abort as described in Fig. 27 and Fig. 28.

$\mathcal{F}_{\mathsf{mpc,poia}}^{\Delta,\gamma,\delta}$ contains, as previous MPC functionalities, the MPC capabilities for input sharing, computation and output sharing. Any party can, after the output

```
Functionality $\mathcal{F}^{\Delta,\gamma,\delta}_{\mathsf{mpc,poia}}$ (Message Scheduling)

Tick:
    1. Remove each $(0, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$ from $\mathcal{Q}$ and instead add $(\mathcal{P}_i, \mathsf{sid}, m)$ to $\mathcal{M}$.
    2. Replace each $(\mathtt{cnt}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$ in $\mathcal{Q}$ with $(\mathtt{cnt} - 1, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$.
    3. If $\mathtt{st} = \mathtt{wait}(x)$ & $x \geq 0$:
       If $x \geq 0$: Set $\mathtt{st} = \mathtt{wait}(x-1)$.
       If $x = \delta + \gamma$:
          (a) Send $(\mathsf{Abort?}, \mathsf{sid})$ to $\mathcal{S}$ and wait for $(\mathsf{Abort}, \mathsf{sid}, J)$ with $J \subseteq I$.
          (b) If $J = \emptyset$ then send message $y$ with prefix $\mathsf{Output}$ to each party
              $\mathcal{P} \setminus I$ via $\mathcal{Q}$ with delay $\delta$.
          (c) If $J \neq \emptyset$ then send message $J$ with prefix $\mathsf{Abort}$ to each party $\mathcal{P} \setminus I$
              via $\mathcal{Q}$ with delay $\delta$.
          (d) Send $(\mathsf{Output}, \mathsf{sid}, y)$ and the IDs to $\mathcal{S}$.
       If $x = 0$:
          (a) If $J = \emptyset$ then set $e_1, \ldots, e_n \leftarrow d$.
          (b) If $J \neq \emptyset$ set $e_i \leftarrow d + |J| \cdot q$ for each party $\mathcal{P}_i \in \mathcal{P} \setminus J$ and
              $e_i \leftarrow d - q \cdot (n - |J|)$ for each $\mathcal{P}_i \in J$.
          (c) Send message $\mathtt{coins}(e_i)$ with prefix $\mathsf{Coins}$ to each party $\mathcal{P} \setminus I$ via
              $\mathcal{M}$ with delay 0.
          (d) Send $(\mathsf{Coins}, \mathsf{sid}, \{\mathtt{coins}(e_i)\}_{\mathcal{P}_i \in I})$ to $\mathcal{S}$.
    4. If $\mathtt{st} = \mathsf{dep}(x)$:
       (a) Set $\mathtt{st} = \mathsf{dep}(x-1)$.
       (b) If $x = 0$: If all parties in $\mathcal{P}$ sent $(\mathsf{Deposit}, \mathsf{sid}, \mathtt{coins}(d))$ then set $\mathtt{st} =$
           $\mathtt{wait}(2\delta + \gamma)$ and send a message $\mathsf{AllDeposited}$ to $\mathcal{P}$ and $\mathcal{S}$ via $\mathcal{M}$.
           Otherwise send a message $\mathtt{coins}(d)$ with prefix $\mathsf{Coins}$ to each party that
           sent the deposit via $\mathcal{M}$, set $\mathtt{st} \leftarrow \bot$ and send a message $\mathsf{Reimbursed}$
           to $\mathcal{P}$ and $\mathcal{S}$ via $\mathcal{M}$.
   Upon receiving $(\mathsf{Schedule}, \mathsf{sid}, \mathcal{D})$ from $\mathcal{S}$:
    – If $(\mathsf{Deliver}, \mathsf{sid}, \mathsf{mid}) \in \mathcal{D}$ then remove each $(\mathtt{cnt}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_i, m)$ from $\mathcal{Q}$ and
      add $(\mathcal{P}_i, \mathsf{sid}, m)$ to $\mathcal{M}$.
    – If $(\mathsf{Abort}, \mathsf{sid}) \in \mathcal{D}$ and $\mathtt{st} = \bot$ then add $(\mathcal{P}_i, \mathsf{sid}, \mathsf{Abort})$ to $\mathcal{M}$ for each $i \in [n]$
      and ignore all further messages with this sid except to Fetch Message.
```

Fig. 28. Ticked Functionality $\mathcal{F}^{\Delta,\gamma,\delta}_{\mathsf{mpc,poia}}$ for Secure Multiparty Computation with Punishable Output-Independent Abort.

sharing is finished, deposit coins to $\mathcal{F}^{\Delta,\gamma,\delta}_{\mathsf{mpc,poia}}$ which will then also immediately notify all other parties and $\mathcal{S}$ about this event, which if it happens the first time will lead to an internal state-change. Unless all parties then deposit coins within γ, they will be reimbursed by $\mathcal{F}^{\Delta,\gamma,\delta}_{\mathsf{mpc,poia}}$, otherwise it switches to a *waiting state* `wait`.

Similar to $\mathcal{F}^{\Delta,\delta}_{\mathsf{mpc,oia}}$, the functionality $\mathcal{F}^{\Delta,\gamma,\delta}_{\mathsf{mpc,poia}}$ will first remain in the waiting state for δ ticks. Then it asks $\mathcal{S}$ to provide the set J of cheating parties to it. After obtaining J, the functionality will then return the output of the computation y to $\mathcal{S}$. Ultimately, the functionality will wait for another $\delta + \gamma$ ticks during which it either reveals the output y to all honest parties (if $J = \emptyset$) or the set J - exactly as $\mathcal{F}^{\Delta,\delta}_{\mathsf{mpc,oia}}$. In addition, after these $\delta + \gamma$ ticks the functionality will either reimburse all parties with $\texttt{coins}(d)$ if $J = \emptyset$ or share all coins among the non-cheating parties $\mathcal{P} \setminus J$ otherwise. The strategy for calculating this reimbursement is identical to $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$.

Protocol $\pi_{\mathsf{mpc,poia}}$ (Computation, Sharing)

All parties $\mathcal{P}$ have access to one instance of the functionalities $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}, \mathcal{F}_{\mathsf{ct}}^{\Delta}$ and $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$. Furthermore, each $\mathcal{P}_i \in \mathcal{P}$ has it's own $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta,i}$ where it acts as the dedicated sender and all other parties of $\mathcal{P}$ are receivers.

Init:

1. Each $\mathcal{P}_i$ sends $(\mathsf{Register}, \mathsf{sid}, \{\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta,j}\}_{j\in[n]})$ to $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$. Then it waits until it receives $\mathsf{Register}$ from $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ for each $\mathcal{P} \setminus \{\mathcal{P}_i\}$.
2. Each $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{Init}, \mathsf{sid}, C)$ to $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$ and waits until it obtains messages C with prefix Init from $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$ for every other party $\mathcal{P} \setminus \{\mathcal{P}_i\}$.

Input: Each $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{Input}, \mathsf{sid}, i, x_i)$ to $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$ and waits until it obtains messages j with prefix Input from $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$ for every other party $\mathcal{P}_j \in \mathcal{P} \setminus \{\mathcal{P}_i\}$.

Computation: Each $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{Computation}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$ and waits until it obtains messages with prefix $\mathsf{Computation}$ from $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$ for every other party $\mathcal{P} \setminus \{\mathcal{P}_i\}$.

Share:

1. Set $\mathcal{T}_y = \{\mathsf{cid}_{y,j}\}_{j\in[m]}$, $\mathcal{T}_r = \{\mathsf{cid}_{r,k}\}_{k\in[\lambda]}$ and $\mathcal{T}_t = \{\mathsf{cid}_{t,k}\}_{k\in[\lambda]}$.
2. Each $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{ShareOutput}, \mathsf{sid}, \mathcal{T}_y)$ to $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$ and wait until it obtains a message $\{y_{i,\mathsf{cid}}\}_{\mathsf{cid}\in\mathcal{T}_y}$ with prefix $\mathsf{OutputShares}$ from $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$.
3. Each $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{ShareRandom}, \mathsf{sid}, \mathcal{T}_r)$ to $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$ and wait until it obtains a message $\{r_{i,\mathsf{cid}}\}_{\mathsf{cid}\in\mathcal{T}_r}$ with prefix $\mathsf{RandomShares}$ from $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$. Set $\boldsymbol{y}_i = (y_{i,\mathsf{cid}_{y,1}}, \dots, y_{i,\mathsf{cid}_{y,m}})$ and equivalently define $\boldsymbol{r}_i$.
4. Each $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{Commit}, \mathsf{sid}, \mathsf{cid}_i, (\boldsymbol{y}_i, \boldsymbol{r}_i))$ to $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta,i}$ and then waits for messages $(\mathsf{Commit}, \mathsf{sid}, \mathsf{cid}_j)$ from the $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta,j}$-instances of all other parties $\mathcal{P}_j \in \mathcal{P} \setminus \{\mathcal{P}_i\}$.
5. Each $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{Toss}, \mathsf{sid}, m \cdot \lambda)$. It then waits for the message $(\mathsf{Coins}, \mathsf{sid}, \mathbf{A})$ where $\mathbf{A} \in \mathbb{F}^{\lambda\times m}$.
6. Each $\mathcal{P}_i \in \mathcal{P}$ for $k \in [\lambda]$ sends $(\mathsf{Linear}, \mathsf{sid}, \{(\mathsf{cid}_{v,j}, \mathbf{A}[k,j])\}_{j\in[m]} \cup \{(\mathsf{cid}_{r,k}, 1)\}, \mathsf{cid}_{t,k})$ to $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$.
7. Each $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{Reveal}, \mathsf{sid}, \mathcal{T}_t)$ to $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$ and waits for the message $\{(\mathsf{cid}, t_{1,\mathsf{cid}}, \dots, t_{n,\mathsf{cid}})\}_{\mathsf{cid}\in\mathcal{T}_t}$ with prefix $\mathsf{DeliverReveal}$ from $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$. Set $\boldsymbol{t}_j = (t_{j,\mathsf{cid}_{t,1}}, \dots, t_{j,\mathsf{cid}_{t,\lambda}})$ for each $j \in [n]$.
8. Each $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{Broadcast}, \mathsf{sid}, (\mathbf{A}, \boldsymbol{t}_1, \dots, \boldsymbol{t}_n))$ to $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$.
9. Each $\mathcal{P}_i \in \mathcal{P}$ waits until it received n identical broadcasts $(\mathsf{Broadcast}, \mathsf{sid}, (\mathbf{A}, \boldsymbol{t}_1, \dots, \boldsymbol{t}_n))$, one from each $\mathcal{P}_j \in \mathcal{P}$.

Fig. 29. Protocol $\pi_{\mathsf{mpc,poia}}$ for MPC with Punishable Output-Independent Abort.

The Protocol. The full protocol $\pi_{\mathsf{mpc,poia}}$ is depicted in Fig. 29 and Fig. 30. It uses a similar approach as $\pi_{\mathsf{mpc,oia}}$, although the broadcast of $\mathbf{A}, \boldsymbol{t}_1, \dots, \boldsymbol{t}_n$ as well as the inherent broadcasts in $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta}$ must now be done via $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$. In comparison to

Protocol $\pi_{\mathsf{mpc,poia}}$ (Deposit)

Deposit:

1. If $\mathcal{P}_i$ finished **Share**, then it sends $(\mathsf{Deposit}, \mathsf{sid}, \mathtt{coins}(d))$ to $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$.
2. Upon having received $(\mathsf{AllDeposited}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ $\mathcal{P}_i$ sends $(\mathsf{DOpen}, \mathsf{sid}, \mathsf{cid}_i)$ to$\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta,i}$. If the party instead obtains $(\mathsf{Coins}, \mathsf{sid}, \mathtt{coins}(d))$ then it aborts and outputs $(\mathsf{Reimbursed}, \mathsf{sid}, \mathtt{coins}(d))$.
3. Each $\mathcal{P}_i \in \mathcal{P}$ waits until $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta,i}$ returns $(\mathsf{DOpened}, \mathsf{sid}, (\mathsf{cid}_i, (\boldsymbol{y}_i, \boldsymbol{r}_i))$. It then checks if it obtained a message with prefix DOpen from all other $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta,j}$. Let $J_1 \subset \mathcal{P}$ be the set of parties such that $\mathcal{P}_i$ did not obtain DOpen before it received $(\mathsf{DOpened}, \mathsf{sid}, (\mathsf{cid}_i, (\boldsymbol{y}_i, \boldsymbol{r}_i))$.
4. Each $\mathcal{P}_i \in \mathcal{P}$ waits until it obtains $(\mathsf{DOpened}, \mathsf{sid}, (\mathsf{cid}_j, (\boldsymbol{y}_j, \boldsymbol{r}_j))$ for each $\mathcal{P}_j \in \mathcal{P} \setminus (J_1 \cup \{\mathcal{P}_i\})$ from the respective instance of $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta,j}$. $\mathcal{P}_i$ then sends $(\mathsf{Broadcast}, \mathsf{sid}, (\mathsf{cid}_j, \boldsymbol{y}_j, \boldsymbol{r}_j, \mathsf{vt}_j))$ to $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ and defines J_2 as the set of all parties $\mathcal{P}_j$ such that $\boldsymbol{t}_j \neq \boldsymbol{r}_j + \mathbf{A}\boldsymbol{y}_j$.
5. If $J_1 \cup J_2 = \emptyset$ then each $\mathcal{P}_i \in \mathcal{P}$ defines $\boldsymbol{y} = \bigoplus_{j \in [n]} \boldsymbol{y}_j$ and outputs $(\mathsf{Output}, \mathsf{sid}, \boldsymbol{y})$. Otherwise it outputs $(\mathsf{Abort}, \mathsf{sid}, J_1 \cup J_2)$.
6. Each $\mathcal{P}_i$ waits for a message $(\mathsf{Coins}, \mathsf{sid}, \mathtt{coins}(e_i))$ from $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$. It then outputs $(\mathsf{Coins}, \mathsf{sid}, \mathtt{coins}(e_i))$.

Fig. 30. Protocol $\pi_{\mathsf{mpc,poia}}$ for MPC with Punishable Output-Independent Abort.

$\pi_{\mathsf{mpc,oia}}$ we do not have to synchronize the honest parties first, as the $\mathsf{AllDeposited}$ message from $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ now serves as synchronization.

Afterwards, honest parties will now open their commitments $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta}$ as before and wait for DOpen messages from other parties' commitment functionalities $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta,j}$. Once they obtain a solution by $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta,j}$, however, they post it on $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ to allow $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ to verify it. Then, those parties who started the opening at the right time, got their openings on $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ and whose openings are correct will then be reimbursed. Here, observe that honest parties will be able to solve the TLPs and solve the solutions within the time-frame given by $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$, so the set J_2 identified by $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ will be identical with the set determined by each honest party.

For the correctness of the protocol, we see that honest parties can never be framed as cheaters as long as $\delta > \gamma$ i.e. as long as the TLPs do not time out before they succeeded at sending their TLPs to the bulletin board.

Overall, this leads to the following

Theorem 6. *Let λ be the statistical security parameter and $\delta > \gamma$. Then the protocol $\pi_{\mathsf{mpc,poia}}$ GUC-securely implements the ticked functionality $\mathcal{F}_{\mathsf{mpc,poia}}^{\Delta,\gamma,\delta}$ in the $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}, \mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta}, \mathcal{F}_{\mathsf{ct}}^{\Delta}, \mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$-hybrid model against any static adversary corrupting up to $n-1$ of the n parties in $\mathcal{P}$. The transcripts are statistically indistinguishable.*