CRAFT: Composable Randomness Beacons and Output-Independent Abort MPC From Time

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

[1] Aarhus University, Denmark
{cbaum,jbn,oechsner}@cs.au.dk
[2] IT University of Copenhagen, Denmark
bernardo@bmdavid.com
[3] Monash University, Australia
rafael@dowsley.net

Abstract. Recently, time-based primitives such as time-lock puzzles (TLP) and verifiable delay functions (VDF) have received a lot of attention due to their power as building blocks for cryptographic protocols. However, even though exciting improvements on their efficiency and security (*e.g.* achieving non-malleability) have been made, most of the existing constructions do not offer general composability guarantees and thus have limited applicability. Baum *et al.* (EUROCRYPT 2021) presented in TARDIS the first (im)possibility results on constructing TLPs with Universally Composable (UC) security and an application to secure two-party computation with output-independent abort (OIA-2PC), where an adversary has to decide to abort *before* learning the output. While these results establish the feasibility of UC-secure TLPs and applications, they are limited to the two-party scenario and suffer from complexity overheads. In this paper, we introduce the first UC constructions of VDFs and of the related notion of publicly verifiable TLPs. We use these primitives to prove folklore results on randomness beacons based on VDFs widely used in industry, as well as introducing a more efficient construction based on publicly verifiable TLPs. We also present the first UC-secure construction of multiparty computation with punishable output-independent aborts (POIA-MPC) (*i.e.* MPC with OIA and financial punishment for cheating), which both establishes the feasibility of OIA-MPC and improves on the efficiency of the state-of-the-art in both (non-OIA) UC-secure MPC with punishable aborts and OIA-2PC.

[⋆] Funded by the European Research Council (ERC) under the European Unions' Horizon 2020 program under grant agreement No 669255 (MPCPRO).
[⋆⋆] Supported by the Concordium Foundation and by the Independent Research Fund Denmark grants number 9040-00399B (TrA^{2}C) and number 9131-00075B (PUMA).
[⋆⋆⋆] Partially done while Rafael Dowsley was with Bar-Ilan University and 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.
[†] Partially funded by The Concordium Foundation; The Danish Independent Research Council under Grant-ID DFF-8021-00366B (BETHE); The Carlsberg Foundation under the Semper Ardens Research Project CF18-112 (BCM).
[‡] Supported by the Danish Independent Research Council under Grant-ID DFF-8021-00366B (BETHE) and Concordium Foundation.

1 Introduction

Time has always been an important, although sometimes overlooked, resource in cryptography. Recently, there has been a renewed interest in time-based primitives such as Time-Lock Puzzles (TLPs) [36] and Verifiable Delay Functions (VDFs) [9]. TLPs allow a sender to commit to a message in such a way that it can be obtained by a receiver only after a certain amount of time, during which the receiver must perform a sequence of computation steps. On the other hand, a VDF works as a pseudorandom function that is evaluated by performing a certain number of computation steps (which take time) and then generates both an output as well as a proof that this number of steps has been performed to obtain the output. A VDF guarantees that evaluating a certain number of steps takes at least a certain amount of time and that the proof obtained with the output can be verified in time essentially independent of the number of steps.

Both TLPs and VDFs have been investigated extensively in recent work which focusses on improving their efficiency [8,34,40], obtaining new properties [23] and achieving stronger security guarantees [20,28,24] for these primitives. These works are motivated by the many applications of TLPs and VDFs as building blocks for cryptographic protocols such as randomness beacons [9,10], partially fair secure computation [18] and fair auctions [10]. In particular, all these applications use TLPs and VDFs concurrently composed with other cryptographic primitives and sub-protocols. However, most of current constructions of TLPs [36,10,8,28,24] and all known constructions of VDFs [9,34,40,20,23] do not offer general composability guarantees, meaning it is not possible to securely plug those constructions in more complex protocols in a straightforward manner.

In order to prove security of cryptographic primitives and protocols under general composability, the current default tool is the Universal Composability (UC) framework [12]. 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-based primitives and protocols in UC. Recently, Baum *et al.* introduced in TARDIS [5] the first UC secure construction of TLPs, which is proven secure in the random oracle model under the iterated squaring assumption of [36] modeled according to the TARDIS model. Baum *et al.* show that a programmable random oracle is *necessary* for realizing such time-based primitives in the UC framework.

Besides analyzing the (im)possibility of constructing UC TLPs, Baum *et al.* [5] showed that these primitives can be used to construct UC-secure Two-Party Computation with Output-Independent Abort (OIA-2PC), where the adversary must decide whether to cause an abort *before* learning the output of the computation. OIA-2PC itself implies fair coin tossing, an important task used in randomness beacons. However, while these results showcase the power of UC-secure TLPs, they are restricted to the two-party setting and incur a high concrete complexity. Moreover, their results do not extend to VDFs. This leaves an important gap, since many applications of TLPs (*e.g.* auctions [10]) are intrinsically multiparty and VDFs have been suggested to be used for practically used randomness beacons [9,39].

1.1 Our Contributions

In this work, we introduce the first UC-secure constructions of VDFs and of the related notion of Publicly Verifiable TLPs, which we introduce. Using these primitives as building blocks, we construct a new efficient randomness beacon and Multiparty Computation with Output-Independent Abort (OIA-MPC). All of our constructions are both practical and proven to be secure under general composition. We discuss our contributions in more detail below.

UC-secure Publicly Verifiable Time-Lock Puzzles. We introduce the notion of UC-secure publicly verifiable TLPs and present an ideal functionality as well as a construction for this primitive. Such TLPs allow the creator to reveal the message in such a way that any third party can verify in constant time that this was indeed the message in the TLP. We show that the TLP of [5] can be proven to be publicly verifiable. Moreover, we generalize their construction to obtain a UC-secure TLP from any trapdoor sequential computation assumption.

UC-secure Verifiable Delay Functions. We introduce the *first* composable definition and matching construction of VDFs [9]. Our construction consists in compiling a generic continuous VDF [23] into a UC-secure continuous VDF in the random oracle model while only increasing the proof size by a small constant.

UC-secure Randomness Beacons. Using our composable VDFs, we give the first security proof of a folklore construction [9] of randomness beacons based on VDFs. Moreover, we introduce a randomness beacons based on publicly verifiable TLPs that achieves better best case scenario efficiency than this VDF-based construction. Our TLP-based construction *requires only $O(n)$ communication to generate a uniformly random output*, where n is the number of parties. However, differently from the VDF-based construction [9], whose execution time is at least the worst case communication channel delay, ours outputs a random value as soon as all messages are delivered, achieving in the optimistic case an *execution as fast as 2 round trip times in the communication channel.* These constructions/proofs require not only UC-secure VDFs and publicly verifiable TLPs but careful consideration of delays in broadcast channels/public ledgers versus the TLP and VDF delays, which we analyze and present in details.

UC-secure Multiparty Computation (MPC) with Punishable Output Independent Abort (POIA-MPC). We construct the first protocol for Multiparty Computation with Punishable Output Independent Abort (POIA-MPC), which is a stronger notion of MPC where the output can be publicly verified and cheaters in the output stage can be identified and financially punished. In particular, this notion implies a multiparty version of the limited OIA-2PC result from [5]. This construction employs our new publicly verifiable TLPs to construct a commitment scheme with delayed opening. In order to use this simple commitment scheme, we improve the currently best [4] techniques for publicly verifiable MPC with cheater identification in the output stage, eliminating the need for homomorphic commitments and achieving a dramatic efficiency improvement.

1.2 Related Work

The recent work of Baum et al. [5] introduced the first construction of a composable TLP. This is in comparison to previous constructions such as [36,10,8] that were only proven to be stand-alone secure. As an intermediate step towards composable TLPs, non-malleable TLPs were constructed in [28,24]. The related notion of VDFss has been investigated in [9,34,40,20,23]. Also for these constructions, composability guarantees have so far not been shown. Hence, issues arise when using these primitives for protocol design, since they are used as building blocks in more complex protocols but their security is not guaranteed when they are composed with other primitives.

Randomness beacons that resist adversarial bias have been constructed based on publicly verifiable secret sharing (PVSS) [30,14] and on VDFs [9], although neither of these constructions is composable. The best UC-secure randomness beacons based on PVSS [15] still require $O(n^2)$ communication where n is the number of parties. Even though they can output $O(n^2)$ random values, they require $O(n^2)$ communication even if only one single value is needed. UC-secure randomness beacons based on verifiable random functions [19,2] 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 2-party setting. Couteau et al. [18] presented a secure two-party fair exchange 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 protocol output. Baum et al. [5] showed how to construct a secure 2-party computation protocol with output-independent abort and composition guarantees. Neither of these works considers the important multiparty setting.

Another work which considers fairness is that of Garay *et al.* [25], which introduced the notion of resource-fairness for protocols in UC. Their work is able to construct fair MPC in a modified UC framework, while we obtain OIA-MPC which can be used to obtain partially fair secure computation (as defined in [26]). The key difference is that their resource-fairness framework needs to modify the UC framework in such a way that environments, adversaries and simulators must have an a priori bounded running time. Our work, by relying on the TARDIS model of [5], does not have to make such stringent (and arguably unrealistic) modifications/restrictions to the UC environment.

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. In this model, rational adversaries have a financial incentive to act fair. Works that achieve fair output delivery with penalties such as [1,7,31,4] 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. Our construction of POIA-MPC forces the adversary to make the decision before seeing the output and incentives can be based on the expected gain of cheating in the computation instead.

1.3 Our Techniques

Publicly Verifiable TLPs. We define the notion of (composable) publicly verifiable TLPs, which allow for a prover who performs all the computational steps needed to solve a TLP to convince any verifier that the TLP solution it obtained is valid, 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 [5], 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 show 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. Additionally, we show that publicly verifiable TLPs can be constructed from any sequential computation with trapdoors.

Verifiable Delay Functions. We depart from a generic stand alone continuous VDF [23] (or rather, a weaker notion of a verifiable sequential computation) to obtain a UC-secure continuous VDF in the global random oracle model (which is necessary for realizing UC-secure time-based primitives as proven in [5]). We capture the stand alone continuous VDF in UC following a similar approach to the one of [5] for capturing the iterated squaring assumption. Interestingly, our construction is simple and efficient: it basically consists of evaluating the input for a number Γ of steps with the stand alone VDF and then computing the UC VDF output and proof as $out = H(sid|\Gamma|\mathtt{st}_\Gamma|\pi')$ and $\pi = (\mathtt{st}_\Gamma, \pi')$ where $\mathtt{st}_\Gamma$ and π' are the output and proof obtained from the stand alone VDF and where H is a global random oracle. Verification is done by checking that out is computed according to the values in π and that π' is valid for the input and $\mathtt{st}_\Gamma$. Even though this construction is simple, analyzing its security requires a complex simulator keeping track of both honest and adversarial VDF evaluations.

Composable Randomness Beacons. We realize a guaranteed output delivery (G.O.D.) coin tossing functionality that works like a randomness beacon from publicly verifiable TLPs and semi-synchronous delayed multiparty communication (through either a delayed broadcast or a public ledger), where there is a finite but unknown communication delay. We consider an honest majority and build on the standard commit-then-reveal coin tossing approach but substitute 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 commitments from half of the parties have been received (meaning at least one commitment is from an honest party), 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. *Even without knowing the maximum communication delay,* our protocols dynamically adjust δ to guarantee that the adversary cannot solve honest parties' TLPs before it has sent its own, meaning it cannot bias the output. If all parties cooperate in sending and opening their TLPs as soon as possible, the output can be obtained as fast as the communication channel delay allows.

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. This property allows honest parties to agree on which parties they may exclude from future secure computations. Moreover, we want to achieve output-independent abort, ensuring that an adversary does not learn the output of a secure computation before deciding that it will cause an abort. While these properties are simple to achieve in the two-party setting with timed commitments [5], achieving them with multiple parties is much more complex. We observe that the use of synchronous communication seems crucial, and we use a broadcast channel to generalize the work of [5]. We then achieve punishable output-independent abort by using publicly verifiable primitives and a smart contract in a way similar to [4]. While we depart from [4,5], care must be taken in the proof since no ideal functionalities have previously been designed for such composable timed primitives. Moreover, we improve the output share consistency check of [4] by using weaker, non-homomorphic commitments, which dramatically reduces the protocol complexity.

2 Preliminaries

We 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 [12,13] 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 is also an iTM, can corrupt a subset $I \subset \mathcal{P}$ as defined by the security model and gains control over these parties. The parties can 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 $\mathcal{P}$. 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.*

Public Verifiability. We model the public verification of outputs, for simplicity, by having a static set of verifiers $\mathcal{V}$. These parties exist during the protocol execution (observing the public protocol transcript) but only act when they receive an input to be publicly verified. Converting our approach to dynamic sets of verifiers (as in e.g. [3]) is possible using standard techniques.

2.2 The TARDIS [5] Composable Time Model

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

Initialize a set of registered parties $P = \emptyset$, 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 $\mathcal{P}$ with pid pid, add pid to P and send (`registered`) to $\mathcal{P}$.

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 $P = L_P$, 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 $\mathcal{P}$ with pid pid $\in P$, add pid to L_P and send (`recorded`) to $\mathcal{P}$.

Fig. 1: Global ticker functionality $\mathcal{G}_{\mathsf{ticker}}$(from [5]).

The TARDIS [5] model expresses time within the GUC framework in such a way that protocols can be made oblivious to clock ticks. To achieve this, TARDIS provides a global ticker functionality $\mathcal{G}_{\mathsf{ticker}}$ as depicted in Fig. 1. This global ticker provides "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 by one unit at a time. Parties may observe events triggered by elapsed time, but not the time as it elapses in $\mathcal{G}_{\mathsf{ticker}}$. Ticked functionalities can freely interpret ticks and perform arbitrary internal state changes. 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). An honest party may contact an arbitrary number of functionalities before asking $\mathcal{G}_{\mathsf{ticker}}$ to proceed. We refer to [5] for more details.

How we use the TARDIS [5] model. To control the observable side effects of ticks, the protocols and ideal functionalities that we present in this work are restricted to interact in the "pull model", precluding functionalities from implicitly providing communication channels between parties. Parties therefore have to actively query functionalities in order to obtain new messages, and they obtain the activation token back upon completion. Ticks to ideal functionalities are modeled as follows: upon each activation, the functionality first checks with $\mathcal{G}_{\mathsf{ticker}}$ if a tick has happened and if so, may act accordingly. For this, it will execute code in a special **Tick** interface.

In comparison to [5], after every tick, each ticked functionality $\mathcal{F}$ that we define (unless mentioned otherwise) allows the adversary to 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 approach is that it simplifies the specification of a correct $\mathcal{F}$. This is because it makes it easier to avoid edge cases where an adversary could influence the output message buffer of $\mathcal{F}$ such that certain conditions supposedly guaranteed by $\mathcal{F}$ break. 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 when being activated after a tick event.

Modeling Start (De)synchronization. In the 2-party setting considered in TARDIS [5] there is no need to capture the fact that parties receive inputs and start executing protocols at different points in time, since parties can adopt the default behavior of waiting for a message from the other before progressing. However, in the multiparty setting (and specially in applications sensitive to time), start synchronization is an important issue that has been observed before in the literature (*e.g.* [32,29]) although it is often overlooked. In the spirit of the original TARDIS model, we flesh out this issue by ensuring that time progresses regardless of honest parties having received their inputs (meaning that protocols may be insecure if a fraction of the parties receive inputs "too late"). Formally, we require that every (honest) party sends (`activated`) to $\mathcal{G}_{\mathsf{ticker}}$ during every activation regardless of having received it's input. We explicitly address the start synchronization conditions required for our protocols to be secure.

Ticked Functionalities. We explicitly mention when a functionality $\mathcal{F}$ is "ticked". 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 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 $\mathcal{M}$. $\mathcal{M}$ contains all the "output-ready" messages that can be read by the parties directly. The content of $\mathcal{M}$ cannot be changed by $\mathcal{A}$ and he cannot prevent parties from reading it. "Messages" from $\mathcal{F}$ may e.g. be messages that have been sent between parties or delayed responses from $\mathcal{F}$ to a request from a party.

We assume that each ticked functionality $\mathcal{F}$ has two special interfaces. One, as mentioned above, is called **Tick** and is activated internally, as outlined before, upon activation of $\mathcal{F}$ if a tick event just happened on $\mathcal{G}_{\mathsf{ticker}}$. The second is called **Fetch Messages**. This latter interface allows parties to obtain entries of $\mathcal{M}$. The code for **Fetch Messages** 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 functionalities in [5] 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 $\mathcal{Q}$ and finally hands out i, mid to the ideal adversary $\mathcal{S}$ as well as potentially also m. This allows $\mathcal{S}$ to influence delivery of m by $\mathcal{F}$ at will by referring to each unique mid. We 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 $T = \{\mathcal{P}_{i_1}, \ldots, \mathcal{P}_{i_k}\}$. Sample unused message IDs $\mathsf{mid}_{i_1}, \ldots, \mathsf{mid}_{i_k}$.
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 $T = \{\mathcal{P}_{i_1}, \ldots, \mathcal{P}_{i_k}\}$. Sample unused message IDs $\mathsf{mid}_{i_1}, \ldots, \mathsf{mid}_{i_k}$.
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 $(\mathsf{Request}, \mathsf{sid}, i, \mathsf{mid}_{i_1}, \ldots, \mathsf{mid}_{i_k})$ to $\mathcal{S}$.

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

– Send $(\mathsf{Request}, \mathsf{sid}, i, m, \mathsf{mid}_{i_1}, \ldots, \mathsf{mid}_{i_k})$ 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* mid for all messages. As the adversary can influence delivery on mid-basis, this ensures simultaneous delivery. We indicate this by using the prefix "simultaneously" in the first two macros.

3 Multi-Party Message Delivery

In this section we will model two common multi-party messaging primitives in the TARDIS [5] framework, namely authenticated broadcast and public ledgers.

Ticked Authenticated Broadcast In Fig. 2 we describe a ticked functionality $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ for delayed authenticated simultaneous broadcast. $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ allows each party $\mathcal{P}_i \in \mathcal{P}$ to broadcast one message m_i in such a way that each m_i is delivered to all parties at the same tick (although different messages m_i and m_j may be delivered at different ticks). This functionality guarantees messages to be delivered at most Δ ticks after they were input. Moreover, it requires that all parties $\mathcal{P}_i \in \mathcal{P}$ must provide inputs m_i within a period of Γ ticks, modeling a start synchronization requirement. In case this loose start synchronization condition is not fulfilled, the functionality no longer provides any guarantees, allowing the adversary to freely manipulate message delivery, as specified by the **Total Breakdown** instructions.

In comparison to the two-party secure channel functionality $\mathcal{F}_{\mathsf{smt,delay}}^{\Delta}$ of [5], our broadcast functionality $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ uses a scheduling-based approach and explicitly captures start synchronization requirements. Using scheduling makes formalizing the multiparty case much easier while requiring start synchronization allows us to realize the functionality as discussed below. This also means that $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ is not a simple generalization of the ticked channels of [5].

We briefly discuss how to implement $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Gamma,\Delta}$. We could start from a synchronous broadcast protocol like [22] or the one in [21] with early stopping. These protocols require all parties to start in the same round and that they terminate within some known upper bound. For $t < n/3$ corruptions we could use [17] to first synchronize the parties before running such a broadcast. If $t \geq n/3$ we can get rid of the requirement that they start in the same round using the round stretching techniques of [33]. This will maintain that the parties terminate within some known upper bound. Then use n instances of such a broadcast channel to let each party broadcast a value. When starting the protocols at time t a party $\mathcal{P}_i$ knows that all protocol instances terminate before time $t + \Delta$ so it can wait until time $t + \Delta$ and collect the set of outputs. Notice that by doing so the original desynchronization Γ is maintained. When using protocols with early stopping [21], the parties might terminate down to one round apart in time. But this will be one of the stretched rounds, so it will increase the original desynchronization by a constant factor.

Ticked Public Ledger In order to define a ledger functionality $\mathcal{F}_{\mathsf{Ledger}}$, we adapt ideas from Badertscher et al. [3]. The ledger functionality $\mathcal{F}_{\mathsf{Ledger}}$ is, due to space limitations, presented in Fig. 20 in Supplementary Material B. There, we also describe it in more detail. The original ledger functionality of Badertscher et al. [3] keeps track of many relevant times and interacts with a global clock in order to take actions at the appropriate time. Our ledger functionality $\mathcal{F}_{\mathsf{Ledger}}$, 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 some of them reach zero.

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

The ticked functionality $\mathcal{F}^{\Gamma,\Delta}_{\mathsf{BC,delay}}$ is parameterized by maximal input desynchronization Γ, maximal delay $\Delta \geq \Gamma$, parties $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ and adversary $\mathcal{S}$. $\mathcal{S}$ may corrupt a strict subset $I \subset \mathcal{P}$. The functionality uses the identifier ssid to distinguish different instances per sid. $\mathcal{F}^{\Gamma,\Delta}_{\mathsf{BC,delay}}$ for each ssid has internal states $\mathtt{st}_{\mathsf{ssid}}, \mathsf{done}_{\mathsf{ssid}}$ that are initially $\perp$.

Send: Upon receiving an input $(\mathsf{Send}, \mathsf{sid}, \mathsf{ssid}, m_i)$ from an honest party $\mathcal{P}_i$: 1. If $\mathtt{st}_{\mathsf{ssid}} = \perp$ then set $\mathtt{st}_{\mathsf{ssid}} = \Gamma$. If either $\mathtt{st}_{\mathsf{ssid}} = \top$ or $\mathcal{P}_i$ sent $(\mathsf{Send}, \mathsf{sid}, \mathsf{ssid}, \cdot)$ before then go to **Total Breakdown**; 2. For all $\mathcal{P}_j \in \mathcal{P}$, add $(\Delta, \mathsf{sid}, \mathcal{P}_j, (\mathcal{P}_i, m_i, \mathsf{ssid}))$ to $\mathcal{Q}$; 3. If all honest parties sent $(\mathsf{Send}, \mathsf{sid}, \mathsf{ssid}, \cdot)$ then set $\mathsf{done}_{\mathsf{ssid}} = \top$; 4. Send $(\mathsf{Send}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_i, m_i)$ to $\mathcal{S}$.

Total Breakdown: Doing a total breakdown means the ideal functionality from now on relays all inputs to $\mathcal{S}$, otherwise ignores the input and lets $\mathcal{S}$ determine all outputs from then on. The ideal functionality becomes a proxy for $\mathcal{S}$.

Tick:

1. If $\mathtt{st}_{\mathsf{ssid}} = a$ for $a \geq 0$: (a) If $a > 0$ then set $\mathtt{st}_{\mathsf{ssid}} = a - 1$; (b) If $a = 0$ and if there is $\mathcal{P}_i \in \mathcal{P} \setminus I$ that did not send $(\mathsf{Send}, \mathsf{sid}, \mathsf{ssid}, \cdot)$ then go to **Total Breakdown**, otherwise set $\mathtt{st}_{\mathsf{ssid}} = \top$; (c) If $\mathsf{done}_{\mathsf{ssid}} = \top$ then wait for m_i from $\mathcal{S}$ for each $\mathcal{P}_i \in I$ and, if $\mathcal{S}$ sends it, then add $(a, \mathsf{sid}, \mathcal{P}_j, (\mathcal{P}_i, m_i, \mathsf{ssid}))$ to $\mathcal{Q}$ for all $\mathcal{P}_j \in \mathcal{P}$, and set $\mathtt{st}_{\mathsf{ssid}} = \top$.
2. Remove each $(0, \mathsf{sid}, \mathcal{P}_i, M)$ from $\mathcal{Q}$ and add $(\mathsf{sid}, \mathcal{P}_i, M)$ to $\mathcal{M}$.
3. Replace each $(\mathtt{cnt}, \mathsf{sid}, \mathcal{P}_i, M)$ in $\mathcal{Q}$ with $(\mathtt{cnt} - 1, \mathsf{sid}, \mathcal{P}_i, M)$.

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

- If $(\mathsf{Deliver}, \mathsf{sid}, \mathsf{ssid}) \in \mathcal{D}$ and $\mathsf{done}_{\mathsf{ssid}} = \top$ then, for all $\mathcal{P}_i \in \mathcal{P}$, remove $(\mathtt{cnt}, \mathsf{sid}, \mathcal{P}_j, (\mathcal{P}_i, m_i, \mathsf{ssid}))$ from $\mathcal{Q}$ and add $(\mathsf{sid}, \mathcal{P}_j, (\mathcal{P}_i, m_i, \mathsf{ssid}))$ to $\mathcal{M}$.

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

We also enforce liveness and chain quality properties, and our ledger functionality can be realized by the same protocols as [3].

4 Publicly Verifiable Time-Lock Puzzles

In this section, we describe an ideal functionality $\mathcal{F}_{\mathsf{tlp}}$ for publicly verifiable TLPs. Intuitively, a publicly verifiable TLP allows a prover who performs all computational steps needed for solving a TLP to later convince a verifier that the solution is correct while requiring the verifier to perform only a constant amount of computational steps. The ideal functionality $\mathcal{F}_{\mathsf{tlp}}$ as presented in Figure 3 models exactly that behavior: $\mathcal{F}_{\mathsf{tlp}}$ has an extra interface for any verifier to check whether a certain solution to a given TLP is correct.

In Supplementary Material C, we provide Protocol π_{tlp} that realizes $\mathcal{F}_{\mathsf{tlp}}$ and prove Theorem 1. Interestingly, our TLP protocol generalizes the construction from [5], building on a generic primitive called *trapdoor sequential computation* captured by functionality $\mathcal{F}_{\mathsf{tsc}}$, which is a generalization of functionality $\mathcal{F}_{\mathsf{rsw}}$ (that captures the RSW assumption as defined in Supplementary Material A.1).

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 $\mathcal{TAG}$. The functionality also interacts with an adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{tlp}}$ contains initially empty lists steps (honest puzzle transitions), omsg (output messages).

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

1. If $\mathcal{P}_o$ is honest sample $\mathtt{tag} \xleftarrow{\$} \mathcal{TAG}$ and $\Gamma + 1$ random distinct states $\mathtt{st}_j \xleftarrow{\$} \{0,1\}^\tau$ for $j \in \{0, \ldots, \Gamma\}$. If $\mathcal{P}_o$ is corrupted, let $\mathcal{S}$ provide values $\mathtt{tag} \in \mathcal{TAG}$ and $\Gamma + 1$ distinct values $\mathtt{st}_j \in \mathcal{ST}$.
2. Append $(\mathtt{st}_0, \mathtt{tag}, \mathtt{st}_\Gamma, m)$ to omsg, append $(\mathtt{st}_j, \mathtt{st}_{j+1})$ to steps for $j \in \{0, \ldots, \Gamma - 1\}$, output $(\mathsf{CreatedPuzzle}, \mathsf{sid}, \mathsf{puz} = (\mathtt{st}_0, \Gamma, \mathtt{tag}), \mathtt{st}_\Gamma)$ to $\mathcal{P}_o$ and $(\mathsf{CreatedPuzzle}, \mathsf{sid}, \mathsf{puz})$ 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 is a $\mathtt{st}'$ such that $(\mathtt{st}, \mathtt{st}') \in \mathsf{steps}$, append $(P_i, \mathtt{st}, \mathtt{st}')$ to $\mathcal{Q}$ and ignore the next steps. Otherwise, proceed as follows:

- If $\mathcal{P}_o$ is honest, sample $\mathtt{st}' \xleftarrow{\$} \mathcal{ST}$.
- If $\mathcal{P}_o$ is corrupted, send $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st})$ to $\mathcal{S}$ and wait for the 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 m such that $(\mathtt{st}_0, \mathtt{tag}, \mathtt{st}, m) \in \mathsf{omsg}$, append $(\mathtt{st}_0, \mathtt{tag}, \mathtt{st}, \perp)$ to omsg.
- If $\mathcal{P}_o$ is corrupted and there is no m such that $(\mathtt{st}_0, \mathtt{tag}, \mathtt{st}, m) \in \mathsf{omsg}$, 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. 3: Ticked Functionality $\mathcal{F}_{\mathsf{tlp}}$ for publicly verifiable time-lock puzzles.

Theorem 1. *Protocol π_{tlp} UC-realizes $\mathcal{F}_{\mathsf{tlp}}$ in the $\mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{tsc}}$-hybrid model with computational security against a static adversary. For every static adversary $\mathcal{A}$ and environment $\mathcal{Z}$, there exists a simulator $\mathcal{S}$ such that $\mathcal{Z}$ cannot distinguish π_{tlp} composed with $\mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{tsc}}$ and $\mathcal{A}$ from $\mathcal{S}$ composed with $\mathcal{F}_{\mathsf{tlp}}$.*

5 Universally Composable Verifiable Delay Functions

We present a generic UC construction of VDFs as modeled in functionality $\mathcal{F}_{\mathsf{VDF}}$ (described in Figure 6) from a generic verifiable sequential computation scheme modeled in functionality $\mathcal{F}_{\mathsf{psc}}$ (described in Figure 5) and a global random oracle $\mathcal{G}_{\mathsf{rpoRO}}$. Our construction is presented in Protocol π_{VDF} (described in Figure 4).

Verifiable Sequential Computation We model a stand alone VDF as a generic verifiable sequential computation captured in functionality $\mathcal{F}_{\mathsf{psc}}$. This functionality is an extension of $\mathcal{F}_{\mathsf{tsc}}$ (which is used for our UC-secure TLP) with two crucial differences: 1. There is no trapdoor allowing for immediate computation of many steps; 2. It is possible to obtain a proof that a given output state is obtained after a certain number of steps from a given input state. While computing the next step from a given input step takes one tick, generating and verifying such a proof of computation takes non-constant amounts of steps modeled as functions of the number of computational steps between the input and output states. This model captures the fact that the existing techniques (*e.g.* [34,40,23]) for generating/verifying such proofs are not constant time.

VDF Functionality We define an ideal functionality $\mathcal{F}_{\mathsf{VDF}}$ for VDFs in Figure 6. While this functionality bears similarities to $\mathcal{F}_{\mathsf{psc}}$, it takes special care of encoding inputs and outputs in order to avoid malleability and allow for equivocation and extraction. In order to evaluate a VDF on input in, the caller uses the **Solve** interface to perform each evaluation step. Once the desired number of steps have been evaluated through **Solve**, the caller can obtain the corresponding output out along with proof π by calling interface **Get Output**. Since generating and verifying the proof may take longer than one solution step, the outputs from the **Get Output** and **Verify** interfaces are delayed accordingly. Further discussion on $\mathcal{F}_{\mathsf{VDF}}$ is presented in Supplementary Material D.

Construction Our protocol π_{VDF} realizing $\mathcal{F}_{\mathsf{VDF}}$ in the $\mathcal{F}_{\mathsf{psc}}, \mathcal{G}_{\mathsf{rpoRO}}$-hybrid model is described in Figure 4. Departing from $\mathcal{F}_{\mathsf{psc}}, \mathcal{G}_{\mathsf{rpoRO}}$ this protocol works by letting the state $\mathtt{el}_1$ be the VDF input in. Once all the Γ solution steps are computed and the final state $\mathtt{el}_\Gamma$ is obtained, the output is defined as $out = H_2(\mathsf{sid}|\Gamma|\mathtt{el}_\Gamma|\pi')$ where H_2 is an instance of $\mathcal{G}_{\mathsf{rpoRO}}$ and the proof π' is obtained by sending $(\mathsf{Prove}, \mathsf{sid}, \mathtt{el}_1, \ldots, \mathtt{el}_\Gamma)$ to $\mathcal{F}_{\mathsf{psc}}$ (*i.e.* proving $\mathtt{el}_\Gamma$ was obtained after Γ computation steps starting from $\mathtt{el}_1$), with the VDF proof defined as $\pi = (\Gamma, \mathtt{el}_1, \mathtt{el}_\Gamma, \pi')$. Verification of an output out obtained from input in with proof π consists of again setting the initial state $\mathtt{el}'_1 = in$ and the output $out' = H_2(sid|\Gamma|\mathtt{el}_\Gamma|\pi')$, then checking that $out = out'$ and verifying with $\mathcal{F}_{\mathsf{psc}}$ that π' is valid with respect to $\Gamma, \mathtt{el}'_1, \mathtt{el}_\Gamma$. The security of Protocol π_{VDF} is formally stated in Theorem 2, which is proven in Supplementary Material D..

Theorem 2. *Protocol π_{VDF} UC-realizes $\mathcal{F}_{\mathsf{VDF}}$ in the $\mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{psc}}$-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_{VDF} composed with $\mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{psc}}$ and $\mathcal{A}$ from $\mathcal{S}$ composed with $\mathcal{F}_{\mathsf{VDF}}$.*

Protocol π_{VDF}

Protocol π_{VDF} is parameterized by a computational security parameter τ, a state space $\mathcal{ST} = \{0,1\}^{\tau}$ and a proof space $\mathcal{PROOF} = \{0,1\}^{5\tau}$. π_{tlp} is executed by a set of parties $\mathcal{P}$ interacting among themselves and with functionalities $\mathcal{F}_{\mathsf{psc}}$ and $\mathcal{G}_{\mathsf{rpoRO2}}$ (an instance of $\mathcal{G}_{\mathsf{rpoRO}}$ with domain $\{0,1\}^{2\cdot\tau}$ and output size $\{0,1\}^{\tau}$). Each party $\mathcal{P}_i \in \mathcal{P}$ maintains initially empty sets L_i^1, L_i^2, L_i^3.

Solve: Upon receiving input $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st})$, a party $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{Step}, \mathsf{sid}, \mathtt{st})$ to $\mathcal{F}_{\mathsf{psc}}$, receiving $(\mathsf{Step}, \mathsf{sid}, \mathtt{st})$ in response, then sends $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$.

Get Output: Upon receiving $(\mathsf{GetOutput}, \mathsf{sid}, in, \mathtt{st}_1, \ldots, \mathtt{st}_\Gamma)$ as input, a party $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{Prove}, \mathsf{sid}, \mathtt{st}_1, \ldots, \mathtt{st}_\Gamma)$ to $\mathcal{F}_{\mathsf{psc}}$, receiving $(\mathsf{Proof}, \mathsf{sid}, \mathtt{st}_1, \ldots, \mathtt{st}_\Gamma)$ in response, then sends $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$.

Verify: On input $(\mathsf{Verify}, \mathsf{sid}, in, out, \Gamma, \pi)$, a party $\mathcal{P}_i \in \mathcal{P}$ parses π as $\pi = (\mathtt{st}_\Gamma, \pi')$ and proceeds as follows:

1. Send $(\textsc{Hash-Query}, (sid|\Gamma|\mathtt{st}_\Gamma|\pi'))$ to $\mathcal{G}_{\mathsf{rpoRO2}}$, obtaining $(\textsc{Hash-Confirm}, h_2)$. Send $(\textsc{IsProgrammed}, (sid|\Gamma|\mathtt{st}_\Gamma|\pi'))$ to $\mathcal{G}_{\mathsf{rpoRO2}}$, obtaining $(\textsc{IsProgrammed}, b_2)$. If $b_2 = 1$ or $out \neq h_2$, output $(sid, in, out, \Gamma, \pi, 0)$ and ignore the next steps.
2. Send $(\mathsf{Verify}, \mathsf{sid}, in, \mathtt{st}_\Gamma, \Gamma, \pi')$ to $\mathcal{F}_{\mathsf{psc}}$, then send $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$.

Fetch State: Upon receiving $(\mathsf{Fetch}, \mathsf{sid})$, output $(\mathsf{Fetch}, \mathsf{sid}, L_i^1, L_i^2, L_i^3)$.

Tick: Every time $\mathcal{P}_i$ is activated, first send $(\mathsf{Output}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{psc}}$, receiving $(\mathsf{Complete}, \mathsf{sid}, \overline{L_i^1}, \overline{L_i^2}, \overline{L_i^3})$. If $(\overline{L_i^1}, \overline{L_i^2}, \overline{L_i^3}) \neq (L_i^1, L_i^2, L_i^3)$, set $(L_i^1, L_i^2, L_i^3) \leftarrow (\overline{L_i^1}, \overline{L_i^2}, \overline{L_i^3})$ and proceed as follows:

1. For all $(\mathcal{P}_i, (\mathtt{st}, \overline{\mathtt{st}})) \in L_i^1$, output $(sid, \mathtt{st}, \overline{\mathtt{st}})$.
2. For all $(\mathcal{P}_i, (\mathtt{st}_1, \mathtt{st}_\Gamma, \pi')) \in L_i^2$, proceed as follows:
   (a) Send $(\textsc{Hash-Query}, (sid|\Gamma|\mathtt{st}_\Gamma|\pi'))$ to $\mathcal{G}_{\mathsf{rpoRO2}}$, obtaining $(\textsc{Hash-Confirm}, h_2)$. Send $(\textsc{IsProgrammed}, (sid|\Gamma|\mathtt{st}_\Gamma|\pi'))$ to $\mathcal{G}_{\mathsf{rpoRO2}}$, obtaining $(\textsc{IsProgrammed}, b_2)$. If $b_2 = 1$, abort.
   (b) Set $out = h_2$, $\pi = (\mathtt{st}_\Gamma, \pi')$ and output $(sid, \mathtt{st}_1, out, \Gamma, \pi)$.
3. For all $(\mathcal{V}_i, (in, \mathtt{st}_\Gamma, \pi', b_\pi)) \in L_i^3$, output $(sid, in, out, \Gamma, \pi, b_\pi))$.
4. If $\mathcal{P}_i$ has no inputs, send $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$. Otherwise process the inputs as described above.

Otherwise, in case $(\overline{L_i^1}, \overline{L_i^2}, \overline{L_i^3}) = (L_i^1, L_i^2, L_i^3)$, if $\mathcal{P}_i$ has no inputs, send $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$, else process the inputs as described above.

Fig. 4: Protocol π_{VDF} realizing $\mathcal{F}_{\mathsf{VDF}}$ in the $\mathcal{F}_{\mathsf{psc}}, \mathcal{G}_{\mathsf{rpoRO}}$-hybrid model.

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

$\mathcal{F}_{\mathsf{psc}}$ interacts with a set of parties $\mathcal{P}$ and an adversary $\mathcal{S}$. It is parameterized by a computational security parameter τ, an initially empty set prf, two lists in and out (computational steps), two lists $\mathcal{Q}_{\mathsf{p}}, \mathcal{Q}_{\mathsf{v}}$ (proofs that currently get computed and proofs that currently get verified), and two lists $\mathcal{M}_{\mathsf{p}}, \mathcal{M}_{\mathsf{v}}$ (outputs of the proof computation and verification steps). $\mathcal{F}_{\mathsf{psc}}$ uses two public functions $f, g : \{0,1\}^\star \mapsto \mathbb{N}$ which describe how many ticks it takes to either create a proof from k consecutive elements (f) or to validate such a proof (g).

Step: Upon receiving $(\mathsf{Step}, \mathsf{sid}, \mathtt{el})$ from $\mathcal{P}_i \in \mathcal{P}$ or $\mathcal{S}$ where $\mathtt{el} \in \{0,1\}^\tau$:

1. If $(\mathtt{el}, \mathtt{nxt}) \in \mathsf{seq}$ for a $\mathtt{nxt} \in \{0,1\}^\tau$, then set $\overline{\mathtt{nxt}} = \mathtt{nxt}$.
2. If $(\mathtt{el}, \mathtt{nxt}) \notin \mathsf{seq}$, proceed as follows:
   - If $\mathcal{P}_i$ is honest, sample $\overline{\mathtt{nxt}} \in \{0,1\}^\tau$ and add $(\mathtt{el}, \overline{\mathtt{nxt}})$ to seq.
   - If $\mathcal{P}_i$ is corrupted or if the message is from $\mathcal{S}$, set $\overline{\mathtt{nxt}} = \perp$.

Finaly, add $(\mathcal{P}_i, (\mathtt{el}, \overline{\mathtt{nxt}}))$ to in and return $(\mathsf{Step}, \mathsf{sid}, \mathtt{el})$ to $\mathcal{P}_i$.

Prove: Upon receiving $(\mathsf{Prove}, \mathsf{sid}, \mathtt{el}_1, \ldots, \mathtt{el}_k)$ from $\mathcal{P}_i \in \mathcal{P}$, add $(\mathcal{P}_i, (f(\mathtt{el}_1, \ldots, \mathtt{el}_k), \mathtt{el}_1, \ldots, \mathtt{el}_k))$ to $\mathcal{Q}_{\mathsf{p}}$ and output $(\mathsf{Proof}, \mathsf{sid}, \mathtt{el}_1, \ldots, \mathtt{el}_k)$ to $\mathcal{P}_i$.

Verify: Upon receiving $(\mathsf{Verify}, \mathsf{sid}, \mathtt{el}_\mathtt{I}, \mathtt{el}_\mathtt{O}, k, \pi)$ from $\mathcal{P}_i \in \mathcal{P}$ where $\pi \in \{0,1\}^\tau$, add $(\mathcal{P}_i, (g(\mathtt{el}_\mathtt{I}, \mathtt{el}_\mathtt{O}, k, \pi), \mathtt{el}_\mathtt{I}, \mathtt{el}_\mathtt{O}, k, \pi))$ to $\mathcal{Q}_{\mathsf{p}}$ and output $(\mathsf{Verify}, \mathsf{sid}, \mathtt{el}_\mathtt{I}, \mathtt{el}_\mathtt{O}, k, \pi)$ to $\mathcal{P}_i$.

Output: Upon receiving $(\mathsf{Output}, \mathsf{sid})$ by $\mathcal{P}_i \in \mathcal{P}$, retrieve the set L_i^1 of all entries $(\mathcal{P}_i, \cdot)$ in out, the set L_i^2 of all entries $(\mathcal{P}_i, \cdot)$ in $\mathcal{M}_{\mathsf{p}}$ and the set L_i^3 of all entries $(\mathcal{P}_i, \cdot)$ in $\mathcal{M}_{\mathsf{v}}$, remove L_i^j from their lists and send $(\mathsf{Complete}, \mathsf{sid}, L_i^1, L_i^2, L_i^3)$ to $\mathcal{P}_i$.

Tick: Set $\mathcal{M}_{\mathsf{p}} = \emptyset, \mathcal{M}_{\mathsf{v}} = \emptyset$ and process queues $\mathsf{in}, \mathcal{Q}_{\mathsf{v}}, \mathcal{Q}_{\mathsf{p}}$:

1. For each $(\mathcal{P}_i, (\mathtt{el}, \perp)) \in \mathsf{in}$, send $(\mathsf{Step}, \mathsf{sid}, \mathtt{el})$ to $\mathcal{S}$ and wait for answer $(\mathsf{Step}, \mathsf{sid}, \mathtt{el}, \overline{\mathtt{nxt}})$. If $\overline{\mathtt{nxt}} \notin \{0,1\}^\tau$ or there exists $(\overline{\mathtt{nxt}}, \cdot) \in \mathsf{seq}$ or there exists $(\cdot, \overline{\mathtt{nxt}}) \in \mathsf{seq}$, output $\perp$ and halt. Otherwise replace $(\mathcal{P}_i, (\mathtt{el}, \perp))$ with $(\mathcal{P}_i, (\mathtt{el}, \overline{\mathtt{nxt}}))$ in in. Set $\mathsf{out} \leftarrow \mathsf{in}$ and $\mathsf{in} = \emptyset$.
2. For each $(\mathcal{P}_i, (c, \mathtt{el}_\mathtt{I}, \mathtt{el}_\mathtt{O}, k, \pi)) \in \mathcal{Q}_{\mathsf{v}}$: if $c = 0$ then remove $(\mathcal{P}_i, (0, \mathtt{el}_\mathtt{I}, \mathtt{el}_\mathtt{O}, k, \pi))$ from $\mathcal{Q}_{\mathsf{v}}$ and add $(\mathcal{P}_i, (\mathtt{el}_\mathtt{I}, \mathtt{el}_\mathtt{O}, k, \pi, 1))$ to $\mathcal{M}_{\mathsf{v}}$ if $(\mathtt{el}_\mathtt{I}, \mathtt{el}_\mathtt{O}, k, \pi) \in \mathsf{prf}$, otherwise add $(\mathcal{P}_i, (\mathtt{el}_\mathtt{I}, \mathtt{el}_\mathtt{O}, k, \pi, 0))$. If $c > 0$ then replace $(\mathcal{P}_i, (c, \mathtt{el}_\mathtt{I}, \mathtt{el}_\mathtt{O}, k, \pi))$ with $(\mathcal{P}_i, (c-1, \mathtt{el}_\mathtt{I}, \mathtt{el}_\mathtt{O}, k, \pi))$ in $\mathcal{Q}_{\mathsf{v}}$.
3. For each $(\mathcal{P}_i, (c, \mathtt{el}_\mathtt{I}, \ldots, \mathtt{el}_\mathtt{O})) \in \mathcal{Q}_{\mathsf{p}}$, if $c = 0$, remove $(\mathcal{P}_i, (0, \mathtt{el}_\mathtt{I}, \ldots, \mathtt{el}_\mathtt{O}))$ from $\mathcal{Q}_{\mathsf{p}}$ and proceed as follows:
   (a) If $(\mathtt{el}_\mathtt{I}, \mathtt{el}_\mathtt{O}, k, \pi) \in \mathsf{prf}$, add $(\mathcal{P}_i, (\mathtt{el}_\mathtt{I}, \mathtt{el}_\mathtt{O}, k, \pi))$ to $\mathcal{M}_{\mathsf{p}}$ and skip next steps.
   (b) If $(\mathtt{el}_\mathtt{I}, \mathtt{el}_\mathtt{O}, k, \pi) \notin \mathsf{prf}$, check that $(\mathtt{el}_j, \mathtt{el}_{j+1}) \in \mathsf{seq}$ for all $j \in [k-1]$. If this check fails, add $(\mathcal{P}_i, (\mathsf{Invalid}, \mathtt{el}_\mathtt{I}, \ldots, \mathtt{el}_\mathtt{O}))$ to $\mathcal{M}_{\mathsf{p}}$ and skip Step (c).
   (c) Send $(\mathsf{ProofStr}, \mathsf{sid}, \mathcal{P}_i, \mathtt{el}_\mathtt{I}, \ldots, \mathtt{el}_\mathtt{O})$ to $\mathcal{S}$ and wait for response $(\mathsf{ProofStr}, \mathsf{sid}, \mathcal{P}_i, \mathtt{el}_\mathtt{I}, \mathtt{el}_\mathtt{O}, \pi)$. If $\pi \notin \{0,1\}^\tau$ or $(\mathtt{el}'_\mathtt{I}, \mathtt{el}'_\mathtt{O}, k', \pi) \in \mathsf{prf}$ or if $(\cdot, \cdot, \cdot, \pi) \in \mathcal{Q}_{\mathsf{v}}$ then $\mathcal{F}_{\mathsf{psc}}$ halts. Otherwise add $(\mathtt{el}_\mathtt{I}, \mathtt{el}_\mathtt{O}, k, \pi)$ to prf and add $(\mathcal{P}_i, (\mathtt{el}_\mathtt{I}, \mathtt{el}_\mathtt{O}, k, \pi))$ to $\mathcal{M}_{\mathsf{p}}$.

   If $c > 0$ then replace $(\mathcal{P}_i, (c, \mathtt{el}_\mathtt{I}, \mathtt{el}_\mathtt{O}))$ with $(\mathcal{P}_i, (c-1, \mathtt{el}_\mathtt{I}, \mathtt{el}_\mathtt{O}))$ in $\mathcal{Q}_{\mathsf{p}}$.

Fig. 5: Ticked Functionality $\mathcal{F}_{\mathsf{psc}}$ for provable sequential computations.

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

$\mathcal{F}_{\mathsf{VDF}}$ is parameterized by a set of parties $\mathcal{P}$, a computational security parameter τ, a state space $\mathcal{ST}$ and a proof space $\mathcal{PROOF}$. In addition to $\mathcal{P}$ the functionality interacts with an ideal adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{VDF}}$ contains initially empty lists steps (honest VDF transitions), OUT (outputs), $\mathcal{Q}_{\mathsf{p}}$ (proof computation queue) and $\mathcal{Q}_{\mathsf{v}}$ (verification computation queue), $\mathcal{M}_{\mathsf{p}}$ (proof output queue) and $\mathcal{M}_{\mathsf{v}}$ (verification output queue). $\mathcal{F}_{\mathsf{VDF}}$ is parameterized by two functions $f, g : \{0,1\}^\star \mapsto \mathbb{N}$.

Solve: Upon receiving $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st})$ from $\mathcal{P}_i \in \mathcal{P}$ where $\mathtt{st} \in \mathcal{ST}$:

- If $(\mathtt{st}, \cdot) \notin \mathsf{steps}$, if $\mathcal{P}_i$ is honest, sample $\mathtt{st}' \xleftarrow{\$} \mathcal{ST}$, append $(\mathtt{st}, \mathtt{st}')$ to steps and $(P_i, \mathtt{st}, \mathtt{st}')$ to $\mathcal{Q}$. If $\mathcal{P}_i$ is instead corrupted, append $(P_i, \mathtt{st}, \perp)$ to $\mathcal{Q}$.
- If $(\mathtt{st}, \mathtt{st}') \in \mathsf{steps}$, append $(P_i, \mathtt{st}, \mathtt{st}')$ to $\mathcal{Q}$.

Get Output: Upon receiving $(\mathsf{GetOutput}, \mathsf{sid}, \mathtt{st}_1, \dots, \mathtt{st}_\Gamma)$ from party $\mathcal{P}_i \in \mathcal{P}$ where $\mathtt{st}_1, \dots, \mathtt{st}_\Gamma \in \mathcal{ST}$, add $(\mathcal{P}_i, (f(\mathtt{st}_1, \dots, \mathtt{st}_\Gamma), \mathtt{st}_1, \dots, \mathtt{st}_\Gamma))$ to $\mathcal{Q}_{\mathsf{p}}$ and output $(\mathsf{Proof}, \mathsf{sid}, \mathtt{st}_1, \dots, \mathtt{st}_\Gamma)$ to $\mathcal{P}_i$.

Verify: Upon receiving $(\mathsf{Verify}, \mathsf{sid}, in, out, \Gamma, \pi)$ from a party $\mathcal{V}_i \in \mathcal{V}$ where $\pi \in \mathcal{PROOF}$, add $(\mathcal{V}_i, (g(in, out, \Gamma, \pi), in, out, \Gamma, \pi))$ to $\mathcal{Q}_{\mathsf{p}}$ and output $(\mathsf{Verify}, \mathsf{sid}, in, out, \Gamma, \pi)$ to $\mathcal{V}_i$.

Fetch State: Upon receiving $(\mathsf{Fetch}, \mathsf{sid})$ from $\mathcal{P}_i \in \mathcal{P}$ retrieve the set L_i^1 of all entries $(\mathcal{P}_i, \cdot)$ in $\mathcal{M}$, the set L_i^2 of all entries $(\mathcal{P}_i, \cdot)$ in $\mathcal{M}_{\mathsf{p}}$ and the set L_i^3 of all entries $(\mathcal{P}_i, \cdot)$ in $\mathcal{M}_{\mathsf{v}}$, remove L_i^j from their respective lists and output $(\mathsf{Fetch}, \mathsf{sid}, L_i^1, L_i^2, L_i^3)$ to $\mathcal{P}_i$. Upon receiving $(\mathsf{Fetch}, \mathsf{sid})$ from $\mathcal{S}$, output $(\mathsf{Fetch}, \mathsf{sid}, \{L_i^1, L_i^2, L_i^3\}_{\mathcal{P}_i \in \mathcal{P}})$.

Tick: Set $\mathcal{M}_{\mathsf{p}} = \emptyset, \mathcal{M}_{\mathsf{v}} = \emptyset$ and process the message queues $\mathcal{Q}, \mathcal{Q}_{\mathsf{p}}, \mathcal{Q}_{\mathsf{v}}$:

1. For each $(\mathcal{P}_i, \mathtt{st}, \perp) \in \mathcal{Q}$, send $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st})$ to $\mathcal{S}$ and wait for answer $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st}, \mathtt{st}')$. If $\mathtt{st}' \notin \mathcal{ST}$ or $(\mathtt{st}', \cdot) \in \mathsf{steps}$ or $(\cdot, \mathtt{st}') \in \mathsf{steps}$, $\mathcal{F}_{\mathsf{VDF}}$ halts. Otherwise, replace $(\mathcal{P}_i, \mathtt{st}, \perp)$ with $(\mathcal{P}_i, \mathtt{st}, \mathtt{st}')$ in $\mathcal{Q}$. Set $\mathcal{M} \leftarrow \mathcal{Q}$ and $\mathcal{Q} = \emptyset$.
2. For each $(\mathcal{V}_i, (c, in, out, \Gamma, \pi)) \in \mathcal{Q}_{\mathsf{v}}$: if $c = 0$ then remove $(\mathcal{V}_i, (0, in, out, \Gamma, \pi))$ from $\mathcal{Q}_{\mathsf{v}}$ and add $(\mathcal{V}_i, (in, out, \Gamma, \pi, 1))$ to $\mathcal{M}_{\mathsf{v}}$ if $(in, out, \Gamma, \pi) \in \mathsf{OUT}$, otherwise add $(\mathcal{V}_i, (in, out, \Gamma, \pi, 0))$. If $c > 0$ then replace $(\mathcal{V}_i, (c, in, out, \Gamma, \pi))$ with $(\mathcal{V}_i, (c-1, in, out, \Gamma, \pi))$ in $\mathcal{Q}_{\mathsf{v}}$.
3. For each $(\mathcal{P}_i, (c, \mathtt{st}_1, \dots, \mathtt{st}_\Gamma)) \in \mathcal{Q}_{\mathsf{p}}$, if $c = 0$, remove $(\mathcal{P}_i, (0, \mathtt{st}_1, \dots, \mathtt{st}_\Gamma))$ from $\mathcal{Q}_{\mathsf{p}}$ and proceed as follows:
   (a) If $(\mathtt{st}_1, out, \Gamma, \pi) \in \mathsf{OUT}$, add $(\mathcal{P}_i, (\mathtt{st}_1, out, \pi))$ to $\mathcal{M}_{\mathsf{p}}$ and skip next steps.
   (b) If $(\mathtt{st}_1, out, \Gamma, \pi) \notin \mathsf{OUT}$, if $(\mathtt{st}_1, \mathtt{st}_2), (\mathtt{st}_2, \mathtt{st}_3), \dots, (\mathtt{st}_{\Gamma-1}, \mathtt{st}_\Gamma) \notin \mathsf{steps}$, add $(\mathcal{P}_i, (\mathsf{Invalid}, \mathtt{st}_1, \dots, \mathtt{st}_\Gamma))$ to $\mathcal{M}_{\mathsf{p}}$ and skip the next step.
   (c) Send $(\mathsf{ProofStr}, \mathsf{sid}, \mathcal{P}_i, \mathtt{st}_1, \dots, \mathtt{st}_\Gamma)$ to $\mathcal{S}$ and wait for response $(\mathsf{ProofStr}, \mathsf{sid}, \mathcal{P}_i, \mathtt{st}_1, \mathtt{st}_\Gamma, \pi)$. If $\pi \notin \mathcal{PROOF}$ or $(\mathtt{st}'_1, out', \Gamma', \pi) \in \mathsf{OUT}$ or if $(\cdot, \pi) \in \mathcal{Q}_{\mathsf{v}}$, then $\mathcal{F}_{\mathsf{VDF}}$ halts. Otherwise sample $out \xleftarrow{\$} \{0,1\}^\tau$, add $(\mathtt{st}_1, out, \Gamma, \pi)$ to OUT and add $(\mathcal{P}_i, (\mathtt{st}_1, out, \Gamma, \pi))$ to $\mathcal{M}_{\mathsf{p}}$.

   If $c > 0$ replace $(\mathcal{P}_i, (c, \mathtt{st}_1, \dots, \mathtt{st}_\Gamma))$ with $(\mathcal{P}_i, (c-1, \mathtt{st}_1, \dots, \mathtt{st}_\Gamma))$ in $\mathcal{Q}_{\mathsf{p}}$.

Fig. 6: Ticked Functionality $\mathcal{F}_{\mathsf{VDF}}$ for Verifiable Delay Functions.

6 UC-secure Semi-Synchronous Randomness Beacons

We model a randomness beacon as a publicly verifiable coin tossing functionality $\mathcal{F}_{\mathsf{RB}}{}^{\Delta_{\mathsf{TLP-RB}}}$ presented in Figure 7. Even though this functionality does not periodically produce new random values as in some notions of randomness beacons, it can be periodically queried by the parties when they need new randomness.

Functionality $\mathcal{F}_{\mathsf{RB}}{}^{\Delta_{\mathsf{TLP-RB}}}$

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

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

Verify: Upon receiving $(\text{Verify}, \mathsf{sid}, x)$ from $\mathcal{V}_j \in \mathcal{V}$, if $(\text{Tossed}, \mathsf{sid}, x)$ has been sent to all parties in $\mathcal{P}$ set $f = 1$, else set $f = 0$. Send $(\text{Verify}, \mathsf{sid}, x, f)$ 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. 7: Ticked Functionality $\mathcal{F}_{\mathsf{RB}}{}^{\Delta_{\mathsf{TLP-RB}}}$ for Randomness Beacons.

6.1 Randomness Beacons from VDFs

It has been suggested that VDFs can be used to obtain a randomness beacon [9] via a simple protocol where parties post plaintext values $r_1, \dots, r_n$ on a public ledger and then evaluate a VDF on input $H(r_1|\dots|r_n)$, where $H()$ is a cryptographic hash function, in order to obtain a random output r. However, despite being used in industry [39], the security of this protocol was never formally proven due to the lack of composability guarantees for VDFs. Our work settles this question by formalizing Protocol $\pi_{\mathsf{VDF-RB}}$ and proving Theorem 3 in Supplementary Material E.

Theorem 3. *If $\Delta = \mathsf{maxTXDelay} + \mathsf{emptyBlocks} \cdot \mathsf{slackWindow}$ (computed from $\mathcal{F}_{\mathsf{Ledger}}$'s parameters) is finite (though unkown), Protocol $\pi_{\mathsf{VDF-RB}}$ UC-realizes $\mathcal{F}_{\mathsf{RB}}{}^{\Delta_{\mathsf{TLP-RB}}}$ in the $\mathcal{F}_{\mathsf{tlp}}, \mathcal{F}_{\mathsf{Ledger}}$-hybrid model with computational security against static adversaries corrupting $t < n/2$ parties in $\mathcal{P}$ for $\Delta_{\mathsf{TLP-RB}} = 2(\Delta + 1) + \sum_{i=1}^{\Delta} i$. 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 $\pi_{\mathsf{VDF-RB}}$ by $\mathcal{A}$ composed with $\mathcal{F}_{\mathsf{tlp}}, \mathcal{F}_{\mathsf{Ledger}}$ from an ideal execution with $\mathcal{S}$ and $\mathcal{F}_{\mathsf{RB}}{}^{\Delta_{\mathsf{TLP-RB}}}$.*

6.2 Randomness Beacons from TLPs

In order to construct a UC-secure randomness beacon from TLPs and a semi-synchronous broadcast channel $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ (with finite but unknown delay Δ), we depart from a simple commit-then-open protocol for n parties with honest majority where commitments are substituted by publicly verifiable TLPs as captured in $\mathcal{F}_{\mathsf{tlp}}$. Such a protocol involves each party $\mathcal{P}_i$ posting a TLP containing a random value r_i, waiting for a set of at least $1 + n/2$ TLPs to be received and then opening their TLPs, which can be publicly verified. The output is defined as $r = r_{j_1} \oplus \cdots \oplus r_{j_{1+n/2}}$, where values r_j are valid TLP openings. If an adversary tries to bias the output by refusing to reveal the opening of its TLP, the honest parties can recover by solving the TLP themselves.

To ensure the adversary cannot bias/abort this protocol, we must ensure two conditions: 1. At least $1 + n/2$ TLPs are broadcast and at least 1 is generated by an honest party (*i.e.* it contains an uniformly random r_i); 2. The adversary must broadcast its TLPs before the honest TLPs open, so it does not learn any of the honest parties' r_i and cannot choose its own r_is in any way that biases the output. While condition 1 is trivially guaranteed by honest majority, we ensure condition 2 by dynamically adjusting the number of steps δ needed to solve the TLPs *without prior knowledge of the maximum broadcast delay Δ*. Every honest party checks that at least $1 + n/2$ TLPs have been received from distinct parties *before* a timeout of δ ticks counted from the moment they broadcast their own TLPs. If this is not the case, the honest parties increase δ and repeat the protocol from the beginning until they receive at least $1 + n/2$ TLPs from distinct parties before the timeout. In the optimistic scenario where all parties follow the protocol (*i.e.* revealing TLP openings) and where the protocol is not repeated, this protocol terminates as fast as all publicly verifiable openings to the TLPs are revealed with computational and communication complexities of $O(n)$. Otherwise, the honest parties only have to solve the TLPs provided by corrupted parties (who do not post a valid opening 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 [35]. 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 RANDAO'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. An adversary could slow down the protocol execution by forcing honest parties to solve the unopened TLPs, but it would not be able to bias the output.

We design and prove security of our protocol with an honest majority in the semi-synchronous model where $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ has a finite but unknown maximum delay Δ. However, *if we were in a synchronous setting with a known broadcast delay Δ, we could achieve security with a dishonest majority* by proceeding to the **Opening Phase** after a delay of $\delta > \Delta$, since there would be a guarantee that all honest party TLPs have been received.

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

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

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

1. **Commitment Phase:** For $i \in \{1, \dots, n\}$, 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 $(\mathsf{Send}, \mathsf{sid}, \mathsf{ssid}, \mathtt{puz}_i)$ to $\mathcal{F}^{\Gamma,\Delta}_{\mathsf{BC,delay}}$ and send $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$.
   (c) Wait for all $\mathcal{P}_j \in \mathcal{P}$ to broadcast their TLPs by setting $\mathtt{cst}_i = 0$ and performing the following steps every time it is activated: i. Send $(\mathsf{Fetch}, \mathsf{sid})$ to $\mathcal{F}^{\Gamma,\Delta}_{\mathsf{BC,delay}}$, receiving $(\mathsf{Fetch}, \mathsf{sid}, L)$; ii. Check that $1 + n/2$ messages of the form $(\mathcal{P}_i, \mathsf{sid}, (\mathcal{P}_j, \mathtt{puz}_j, \mathsf{ssid}))$ from different parties are in L (we call the set of such parties $\mathcal{C}$) and proceed to the **Opening Phase** if yes; iii. 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 $\mathtt{cst}_i$; iv. If $\mathtt{cst}_i = \delta$, increment δ and go back to Step 1(a). v. Send $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st}^i_{\mathtt{cst}_i})$ to $\mathcal{F}^i_{\mathsf{tlp}}$; vi. Send $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$.
2. **Opening Phase:** All parties $\mathcal{P}_i \in \mathcal{C}$ proceed as follows:
   (a) Send $(\mathsf{Send}, \mathsf{sid}, \mathsf{ssid}', \mathtt{st}^i_\delta, r_i)$ to $\mathcal{F}^{\Gamma,\Delta}_{\mathsf{BC,delay}}$.
   (b) Wait for all $\mathcal{P}_j \in \mathcal{C}$ to broadcast a solution to their TLPs by setting $\mathtt{cst}_i = 0$ and performing following steps every time it is activated: i. 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 $\mathtt{cst}_i$; ii. If $\mathtt{cst}_i = \delta$, exit loop and go to next step. iii. Send $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st}^i_{\mathtt{cst}_i})$ to $\mathcal{F}^i_{\mathsf{tlp}}$; iv. Send $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$;
   (c) Send $(\mathsf{Fetch}, \mathsf{sid})$ to $\mathcal{F}^{\Gamma,\Delta}_{\mathsf{BC,delay}}$, receiving $(\mathsf{Fetch}, \mathsf{sid}, L)$. Check that all messages of the form $(\mathcal{P}_i, \mathsf{sid}, (\mathcal{P}_j, \mathtt{st}^j_\delta, r_j, \mathsf{ssid}'))$ from $\mathcal{P}_j \in \mathcal{C}$ is a valid solution to $\mathtt{puz}_j$ by sending $(\mathsf{Verify}, \mathsf{sid}, \mathtt{puz}_j, \mathtt{st}^j_\delta, r_j)$ to $\mathcal{F}^j_{\mathsf{tlp}}$ and checking that the answer is $(\mathsf{Verified}, \mathsf{sid}, \mathtt{puz}_j, \mathtt{st}^j_\delta, r_j, 1)$. Send $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$. If this check passes for all $\mathtt{puz}_j$ from $\mathcal{P}_j \in \mathcal{C}$, compute $r = \bigoplus_{r_i \in \mathcal{V}} r_i$, output $(\textsc{Tossed}, \mathsf{sid}, r)$ and skip **Recovery Phase**. Otherwise, proceed.
3. **Recovery Phase:** For $i \in \{1, \dots, n\}$, 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 $(\mathsf{Fetch}, \mathsf{sid})$ 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 $\mathtt{cst}_j$; ii. Send $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st}^j_{\mathtt{cst}_j})$ to $\mathcal{F}^j_{\mathsf{tlp}}$; 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 and sending $(\mathsf{Send}, \mathsf{sid}, \mathsf{ssid}'', (\mathtt{st}^j_\delta, r_j))$ to $\mathcal{F}^{\Gamma,\Delta}_{\mathsf{BC,delay}}$. If all $r_1, \dots, r_{|\mathcal{C}|}$ have been obtained, $\mathcal{P}_i$ exits the loop and proceeds to the next step. Send $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$.
   (b) Let $\mathcal{G}$ be the set of all r_j such that $\mathcal{P}_j \in \mathcal{C}$, r_j is a valid solution of $\mathtt{puz}_j$ and $r_j \neq \perp$. (*i.e.* $\mathcal{G}$ is the set of values r_i from valid TLPs posted in the commitment phase). Compute $r = \bigoplus_{r_j \in \mathcal{G}} r_j$, output $(\textsc{Tossed}, \mathsf{sid}, r)$. Send $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$.

Fig. 8: Protocol $\pi_{\mathsf{TLP-RB}}$: Commitment, Opening and Recovery Phases

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

Verify: On input $(\textsc{Verify}, \mathsf{sid}, x)$, send $(\mathsf{Fetch}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$, receiving $(\mathsf{Fetch}, \mathsf{sid}, L)$ and determining $\mathcal{C}$ for sid from L (by looking for the first $1 + n/2$ messages of the form $(\mathcal{P}_i, \mathsf{sid}, (\mathcal{P}_j, \mathtt{puz}_j, \mathsf{ssid})))$. Check that each message of the form $(\mathcal{P}_i, \mathsf{sid}, (\mathcal{P}_h, \mathtt{st}_\delta^j, r_j, \mathsf{ssid}'))$ or $(\mathcal{P}_i, \mathsf{sid}, (\mathcal{P}_h, \mathtt{st}_\delta^j, r_j, \mathsf{ssid}''))$ in L for $\mathcal{P}_j \in \mathcal{C}$ and $\mathcal{P}_h \in \mathcal{P}$ (*i.e.* messages containing solutions to a puzzle $\mathtt{puz}_j$ from a party $\mathcal{P}_j \in \mathcal{C}$, also including messages potentially sent in the recovery phase by any party $\mathcal{P}_h \in \mathcal{P}$ who solved an unopened $\mathtt{puz}_j$) contains a valid solution to $\mathtt{puz}_j$ by sending $(\mathsf{Verify}, \mathsf{sid}, \mathtt{puz}_j, \mathtt{st}_\delta^j, r_j)$ to $\mathcal{F}_{\mathsf{tlp}}$ and checking that the answer is $(\mathsf{Verified}, \mathsf{sid}, \mathtt{puz}_j, \mathtt{st}_\delta^j, r_j, 1)$. Let $\mathcal{G}$ be the set of all r_j such that $\mathcal{P}_j \in \mathcal{C}$, r_j is a valid solution of $\mathtt{puz}_j$ and $r_j \neq \perp$. Check if $x = \bigoplus_{r_j \in \mathcal{G}} r_j$. If all checks pass set $f = 1$, else 0, output $(\textsc{Verify}, \mathsf{sid}, x, f)$.

Fig. 9: Protocol $\pi_{\mathsf{TLP-RB}}$: Verifiy.

We describe protocol $\pi_{\mathsf{TLP-RB}}$ in Figures 8 and 9 and state its security in Theorem 4, which is proven in Supplementary Material F.

Theorem 4. *If Δ is finite (though unknown) and all $\mathcal{P}_i \in \mathcal{P}$ receive inputs within a delay of Γ ticks of each other, Protocol $\pi_{\mathsf{TLP-RB}}$ UC-realizes ${\mathcal{F}_{\mathsf{RB}}}^{\Delta_{\mathsf{TLP-RB}}}$ in the $\mathcal{F}_{\mathsf{tlp}}, \mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$-hybrid model with computational security against static adversaries corrupting $t < \frac{n}{2}$ parties in $\mathcal{P}$ for $\Delta_{\mathsf{TLP-RB}} = 3(\Delta+1)+\sum_{i=1}^{\Delta} i$. 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 $\pi_{\mathsf{TLP-RB}}$ by $\mathcal{A}$ composed with $\mathcal{F}_{\mathsf{tlp}}, \mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ from an ideal execution with $\mathcal{S}$ and ${\mathcal{F}_{\mathsf{RB}}}^{\Delta_{\mathsf{TLP-RB}}}$.*

6.3 Using a Public Ledger $\mathcal{F}_{\mathsf{Ledger}}$ with $\pi_{\mathsf{TLP-RB}}$

Instead of using a delayed broadcast $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$, 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 $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ in the following way:

– At every point of $\pi_{\mathsf{TLP-RB}}$ where parties send $(\mathsf{Send}, \mathsf{sid}, \mathsf{ssid}, m)$ to $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$, 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 $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ 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 5. *If $\Delta = \mathsf{maxTXDelay} + \mathsf{emptyBlocks} \cdot \mathsf{slackWindow}$ (computed from $\mathcal{F}_{\mathsf{Ledger}}$'s parameters) is finite (though unknown), Protocol $\pi_{\mathsf{TLP-RB-LEDGER}}$ UC-realizes ${\mathcal{F}_{\mathsf{RB}}}^{\Delta_{\mathsf{TLP-RB}}}$ 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 $\mathcal{P}$ for $\Delta_{\mathsf{TLP-RB}} =$*

$3(\Delta + 1) + \sum_{i=1}^{\Delta} i$. *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 $\mathcal{F}_{\mathsf{RB}}{}^{\Delta_{\mathsf{TLP-RB}}}$.*

Proof. This theorem is proven in Supplementary Material F. □

7 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 the broadcast as modeled in $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ and TLPs.

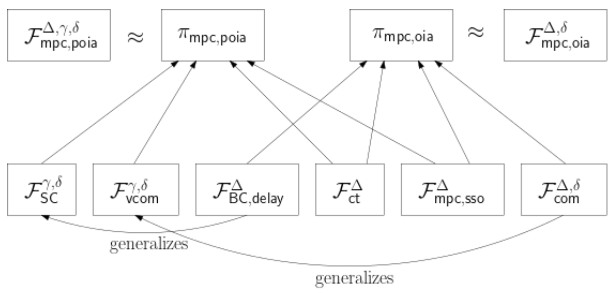
7.1 Functionalities for Output-Independent Abort

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. $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$ (Fig. 11 and Fig. 12) for secure MPC with secret-shared output.
2. $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta}$ (Fig. 13 and Fig. 14) for MPC with output-independent abort.

In Supplementary Material G, we also introduce the following functionalities:

1. $\mathcal{F}_{\mathsf{ct}}^{\Delta}$ (Fig. 18) for coin-flipping with abort.
2. $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta}$ (Fig. 28) for commitments with delayed non-interactive openings.
3. $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta}$ (Fig. 29 and Fig. 30) for commitments with verifiable delayed non-interactive openings.
4. $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ (Fig. 32 and Fig. 33) which is an abstraction of a smart contract.
5. $\mathcal{F}_{\mathsf{mpc,poia}}^{\Delta,\gamma,\delta}$ (Fig. 34 and Fig. 35) for MPC with punishable output-independent abort.



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

Fig. 10: How MPC with (Punishable) Output-Independent Abort is constructed.

Before formally introducing all functionalities and explaining them in more detail, we show how they are related in our construction in Figure 10. As can be seen there our approach is twofold. First, we will realize $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta}$ via the protocol $\pi_{\mathsf{mpc,oia}}$ relying on $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}, \mathcal{F}_{\mathsf{ct}}^{\Delta}, \mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$ and $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta}$. Then, we will show how to implement $\mathcal{F}_{\mathsf{mpc,poia}}^{\Delta,\gamma,\delta}$ via the protocol $\pi_{\mathsf{mpc,poia}}$ (a generalization of $\pi_{\mathsf{mpc,oia}}$) which uses $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}, \mathcal{F}_{\mathsf{ct}}^{\Delta}, \mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$ as well as $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta}$. As mentioned in Fig. 10, $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta}$ and $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ can be thought of as generalizations of $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta}$ and $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$. We now describe the functionalities for $\pi_{\mathsf{mpc,oia}}$ in more detail.

Functionality $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$ (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. 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. Then accept x_i as input for $\mathcal{P}_i$.
2. 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. If all parties sent $(\mathsf{Compute}, \mathsf{sid})$ compute and store $(y_1, \ldots, y_m) \leftarrow C(x_1, \ldots, x_n)$.
2. Notify $\mathcal{S}$ about a message with prefix 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. 11: Ticked Functionality $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$ for MPC with Secret-Shared Output and Linear Secret Share Operations.

MPC with Secret-Shared Output. The functionality $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$ is formally introduced in Fig. 11 and Fig. 12. It directly translates an MPC protocol with secret-shared output into the TARDIS 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. $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$ can be instantiated from many different MPC protocols, such as those based on secret-sharing [6] or multiparty BMR [27].

Functionality $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$ (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. 12: Ticked Functionality $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$ for MPC with Secret-Shared Output and Linear Secret Share Operations, Part 2.

Commitments with Delayed Openings. In Fig. 28 in Supplementary Material G we describe the functionality $\mathcal{F}^{\Delta,\delta}_{\mathsf{com}}$ for commitments with 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 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.

While both the **Commit** and **Open** directly resemble their counterparts in a normal commitment functionality, the **Delayed Open** logic is not as straightforward. 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 - for which they obtain a message DOpen. 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. If the openings, once triggered by **Tick**, are written to the output queue $\mathcal{M}$ then they can directly be read by the respective parties. $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta}$ ensures that *all honest parties will learn the delayed opening simultaneously.*

In Supplementary Material G 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 [5].

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, each party will 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. 18 in Supplementary Material A. Note that $\mathcal{F}_{\mathsf{ct}}^{\Delta}$ can easily be realized in the $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}, \mathcal{F}_{\mathsf{com}}^{\Delta,\delta}$-hybrid model.

[4] See Theorem 7 for more details. To adapt the construction to $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta}$ it is sufficient to replace the bulletin board with a broadcast functionality such as $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$.

7.2 Building MPC with Output-Independent Abort

We will now describe how to construct an MPC protocol that guarantees output-independent abort. This generalizes [5] to the 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 same set of cheaters. Our protocol, on a high level, works as follows:

1. The parties begin by sending a message *beat* (i.e. a heartbeat) to the functionality $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$. Throughout the protocol, they do the following in parallel to running the MPC protocol, unless mentioned otherwise:
   All parties wait for a broadcast message *beat* from all parties on $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$. If some parties did not send their message to $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ in one iteration then all parties abort. Otherwise, they send *beat* in another iteration to $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$.
2. 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 $\mathcal{P}_i$.
3. 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.
4. 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.
5. For each $i \in [n]$ the parties compute and open $\boldsymbol{t}_i = \boldsymbol{r}_i + \mathbf{A}\boldsymbol{y}_i$ using $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$. 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.
6. Each party that obtained $\boldsymbol{t}_i$ changes the next *beat* message to $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ to *ready*. Once parties receive *ready* from all other parties and are therefore synchronized, they simultaneously perform a delayed open of both $\boldsymbol{y}_i, \boldsymbol{r}_i$ using their commitments (and ignore $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ from now on). Parties which don't open commitments in time or whose opened values do not yield $\boldsymbol{t}_i$ as outlined before are considered as cheaters.

Intuitively, our construction has output-independent abort because of the timing of the opening: Until Step 6, 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.

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

The ticked functionality runs with n parties $\mathcal{P} = \{\mathcal{P}_1, \dots, \mathcal{P}_n\}$ and an adversary $\mathcal{S}$ who may corrupt a strict subset $I \subset \mathcal{P}$. $\mathcal{F}^{\Delta,\delta}_{\mathsf{mpc,oia}}$ 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. 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. Then accept x_i as input for $\mathcal{P}_i$.
2. 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 $x_1, \dots, x_n$ 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 $y = C(x_1, \dots, x_n)$ 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 $\mathcal{S}$ 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 IDs.
   (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 each 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 $\mathtt{st} = \mathtt{sync}$ and notify $\mathcal{S}$ about a message with prefix Reveal.

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

Concerning agreement on the output of the honest parties, we see that if all honest parties initially start almost synchronized (i.e. at most Γ ticks apart) then if they do not abort during the protocol they will simultaneously open their commitments. Therefore, using $\mathcal{F}^{\Gamma,\Delta}_{\mathsf{BC,delay}}$ guarantees that they all have the same view of all adversarial messages during the **Reveal** phase.

Interestingly, our construction does not need homomorphic commitments as was necessary in [5,4] to achieve their verifiable or output-independent abort in UC. Clearly, our solution can also be used to improve these protocols and

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

```
Tick:
  1. Set $\mathtt{f} \leftarrow \bot$, 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 $\mathtt{f} \leftarrow \top$.
  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 $x = 0$:
         (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 Output to each party $\mathcal{P} \setminus I$
             via $\mathcal{Q}$ with delay $\delta$. If $J \neq \emptyset$ then send message $J$ with prefix Abort to
             each party $\mathcal{P} \setminus I$ via $\mathcal{Q}$ with delay $\delta$.
         (c) Send $(\mathsf{Output}, \mathsf{sid}, y)$ and the IDs to $\mathcal{S}$.
  4. If $\mathtt{st} = \mathtt{sync}$ and $f = \top$ then set $\mathtt{st} = \mathtt{wait}(\delta)$ and 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 sid except to Fetch Message.
```

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

to simplify their constructions. The full protocol can be found in Fig. 15 and Fig. 16. In Supplementary Material G we prove the following Theorem:

Theorem 6. *Let λ be the statistical security parameter and $\delta > \Delta$. Furthermore, assume that all honest parties obtain their inputs at most Γ ticks apart. Then the protocol $\pi_{\mathsf{mpc,oia}}$ GUC-securely implements the ticked functionality $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta}$ in the $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}, \mathcal{F}_{\mathsf{com}}^{\Delta,\delta}, \mathcal{F}_{\mathsf{ct}}^{\Delta}, \mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$-hybrid model against any static adversary corrupting up to $n-1$ parties in $\mathcal{P}$. The transcripts are statistically indistinguishable.*

7.3 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.

In order to manage monetary contributions of parties, we will use a smart contract functionality $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ 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 $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ for synchronization. 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 during the opening phase, which $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ can then punish.

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

All parties $\mathcal{P}$ have access to one instance of the functionalities $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$, $\mathcal{F}^{\Delta}_{\mathsf{ct}}$ and $\mathcal{F}^{\Gamma,\Delta}_{\mathsf{BC,delay}}$. Furthermore, each $\mathcal{P}_i \in \mathcal{P}$ has it's own $\mathcal{F}^{\Delta,\delta,i}_{\mathsf{com}}$ where it acts as the dedicated sender and all other parties of $\mathcal{P}$ are receivers.
Throughout the protocol, we say "$\mathcal{P}_i$ ticks" when we mean that it sends (activated) to $\mathcal{G}_{\mathsf{ticker}}$. We say that "$\mathcal{P}_i$ waits" when we mean that it, upon each activation, first checks if the event happened and if not, sends (activated) to $\mathcal{G}_{\mathsf{ticker}}$.

Upon every activation: Let c be a counter that is initially 0. $\mathcal{P}_i$ sends $(\mathsf{Send}, \mathsf{sid}, c, \mathit{beat})$ to the functionality $\mathcal{F}^{\Gamma,\Delta}_{\mathsf{BC,delay}}$ (with c as ssid). Throughout $\pi_{\mathsf{mpc,oia}}$, each $\mathcal{P}_i$ waits for $\mathcal{F}^{\Gamma,\Delta}_{\mathsf{BC,delay}}$ to return $(\mathcal{P}_j, \mathit{beat}, c)$ for all other $\mathcal{P}_j \in \mathcal{P}$. If it does, then each $\mathcal{P}_i$ increases c by 1 and sends $(\mathsf{Send}, \mathsf{sid}, c, \mathit{beat})$ to $\mathcal{F}^{\Gamma,\Delta}_{\mathsf{BC,delay}}$. Otherwise the parties abort.

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

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

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 ticks. Then it waits until it obtains a message $\{y_{i,\mathsf{cid}}\}_{\mathsf{cid}\in\mathcal{T}_y}$ with prefix 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 ticks. It then waits until it obtains a message $\{r_{i,\mathsf{cid}}\}_{\mathsf{cid}\in\mathcal{T}_r}$ with prefix 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 ticks. It then waits for messages $(\mathsf{Commit}, \mathsf{sid}, \mathsf{cid}_j)$ from the $\mathcal{F}^{\Delta,\delta,j}_{\mathsf{com}}$-instances of all other $\mathcal{P}_j \in \mathcal{P} \setminus \{\mathcal{P}_i\}$.
5. Each $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{Toss}, \mathsf{sid}, m \cdot \lambda)$ to $\mathcal{F}^{\Delta}_{\mathsf{ct}}$ and ticks. 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 ticks. It then waits for the message $\{(\mathsf{cid}, t_{1,\mathsf{cid}}, \ldots, t_{n,\mathsf{cid}})\}_{\mathsf{cid}\in\mathcal{T}_t}$ with prefix 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.

In order to identify the set J_2 of parties that open in time but with incorrect output shares, $\mathcal{F}^{\gamma,\delta}_{\mathsf{SC}}$ must be able to check openings of $\mathcal{F}^{\Delta,\delta}_{\mathsf{com}}$. For this reason we will enhance this functionality to $\mathcal{F}^{\gamma,\delta}_{\mathsf{vcom}}$ which has verifiability for opened values. Moreover, we ask that the opened values of $\mathcal{F}^{\gamma,\delta}_{\mathsf{vcom}}$ have to be transferable: if any party $\mathcal{P}_i$ has found the opening, then by sending it to $\mathcal{F}^{\gamma,\delta}_{\mathsf{SC}}$ the smart contract

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

Reveal: If **Share** completed successfully:

1. Each party changes the messages to $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ to $(\mathsf{Send}, \mathsf{sid}, c, ready)$. Upon receiving the first $(\mathcal{P}_j, ready, c)$ for all $\mathcal{P}_j \in \mathcal{P}$ from $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$, each $\mathcal{P}_i$ sends $(\mathsf{DOpen}, \mathsf{sid}, \mathsf{cid}_i)$ to $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta,i}$ and ticks. It also stops sending *beat* messages to $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$.
2. Each $\mathcal{P}_i \in \mathcal{P}$ waits until $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta,i}$ returns $(\mathsf{DOpened}, \mathsf{sid}, (\mathsf{cid}_i, (\boldsymbol{y}_i, \boldsymbol{r}_i)))$. If it opens then $\mathcal{P}_i$ checks if it obtained a message with prefix DOpen from all other $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta,j}$. Let $J_1 \subset \mathcal{P}$ be the set of parties such that $\mathcal{P}_i$ did not obtain DOpen before it received $(\mathsf{DOpened}, \mathsf{sid}, (\mathsf{cid}_i, (\boldsymbol{y}_i, \boldsymbol{r}_i)))$.
3. 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 $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta,j}$. It then defines J_2 as the set of all parties $\mathcal{P}_j$ such that $\boldsymbol{t}_j \neq \boldsymbol{r}_j + \mathbf{A}\boldsymbol{y}_j$.
4. 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.

should can verify the opening without running the delayed opening step which involves solving a TLP.

In the full protocol, parties will then first compute on their inputs and generate shares of the outputs as in $\pi_{\mathsf{mpc,oia}}$, although using the aforementioned different functionalities. Then, before starting the opening phase, each party will send a 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 G for the concrete construction.

Achieving Partial Fairness. Gordon and Katz [26] introduced the concept of partially fair two-party computation, where correctness and privacy of the secure protocol always holds but fairness may not hold with probability $1/|poly(\lambda)|$. This notion, generalized to the multiparty-setting, can also be achieved by extending $\pi_{\mathsf{mpc,oia}}$. There, the parties would sample a secret bit b fairly in MPC. If $b = 1$ then the reconstructed output would be the actual output of the computation together with the bit b, while if $b = 0$ then the output phase would only reveal a "dummy output" and b. Over multiple output rounds, this process will be repeated until $b = 1$. Partial fairness of the approach follows as the adversary has to decide if or not it will reveal the output *before learning the value of* b *for the respective round.* This idea can also be combined with financial incentives to strengthen the guarantees of $\pi_{\mathsf{mpc,poia}}$.

References

1. M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Fair two-party computations via bitcoin deposits. In *FC 2014 Workshops*, LNCS. Springer, Heidelberg, Mar. 2014.
2. C. Badertscher, P. Gazi, A. Kiayias, A. Russell, and V. Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In *ACM CCS 2018*. ACM Press, Oct. 2018.
3. C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas. Bitcoin as a transaction ledger: A composable treatment. In *CRYPTO 2017, Part I*, LNCS. Springer, Heidelberg, Aug. 2017.
4. C. Baum, B. David, and R. Dowsley. Insured MPC: Efficient secure computation with financial penalties. In *FC 2020*, LNCS. Springer, Heidelberg, Feb. 2020.
5. C. Baum, B. David, R. Dowsley, J. B. Nielsen, and S. Oechsner. Tardis: A foundation of time-lock puzzles in uc. to appear at EUROCRYPT 2021, 2020. https://eprint.iacr.org/2020/537.
6. R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT 2011*, LNCS. Springer, Heidelberg, May 2011.
7. I. Bentov and R. Kumaresan. How to use bitcoin to design fair protocols. In *CRYPTO 2014, Part II*, LNCS. Springer, Heidelberg, Aug. 2014.
8. N. Bitansky, S. Goldwasser, A. Jain, O. Paneth, V. Vaikuntanathan, and B. Waters. Time-lock puzzles from randomized encodings. In *ITCS 2016*. ACM, Jan. 2016.
9. D. Boneh, J. Bonneau, B. Bünz, and B. Fisch. Verifiable delay functions. In *CRYPTO 2018, Part I*, LNCS. Springer, Heidelberg, Aug. 2018.
10. D. Boneh and M. Naor. Timed commitments. In *CRYPTO 2000*, LNCS. Springer, Heidelberg, Aug. 2000.
11. J. Camenisch, M. Drijvers, T. Gagliardoni, A. Lehmann, and G. Neven. The wonderful world of global random oracles. In *EUROCRYPT 2018, Part I*, LNCS. Springer, Heidelberg, Apr. / May 2018.
12. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*. IEEE Computer Society Press, Oct. 2001.
13. R. Canetti, Y. Dodis, R. Pass, and S. Walfish. Universally composable security with global setup. In *TCC 2007*, LNCS. Springer, Heidelberg, Feb. 2007.
14. I. Cascudo and B. David. SCRAPE: Scalable randomness attested by public entities. In *ACNS 17*, LNCS. Springer, Heidelberg, July 2017.
15. I. Cascudo and B. David. ALBATROSS: Publicly AttestabLe BATched Randomness based On Secret Sharing. In *ASIACRYPT 2020, Part III*, LNCS. Springer, Heidelberg, Dec. 2020.
16. R. Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *18th ACM STOC*. ACM Press, May 1986.
17. B. A. Coan, D. Dolev, C. Dwork, and L. J. Stockmeyer. The distributed firing squad problem. *SIAM J. Comput.*, 18(5):990–1012, 1989.
18. 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.
19. B. David, P. Gazi, A. Kiayias, and A. Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *EUROCRYPT 2018, Part II*, LNCS. Springer, Heidelberg, Apr. / May 2018.

20. L. De Feo, S. Masson, C. Petit, and A. Sanso. Verifiable delay functions from supersingular isogenies and pairings. In *ASIACRYPT 2019, Part I*, LNCS. Springer, Heidelberg, Dec. 2019.
21. D. Dolev, R. Reischuk, and H. R. Strong. Early stopping in byzantine agreement. *J. ACM*, 37(4):720–741, 1990.
22. D. Dolev and H. R. Strong. Polynomial algorithms for multiple processor agreement. In H. R. Lewis, B. B. Simons, W. A. Burkhard, and L. H. Landweber, editors, *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*, pages 401–407. ACM, 1982.
23. N. Ephraim, C. Freitag, I. Komargodski, and R. Pass. Continuous verifiable delay functions. In *EUROCRYPT 2020, Part III*, LNCS. Springer, Heidelberg, May 2020.
24. N. Ephraim, C. Freitag, I. Komargodski, and R. Pass. Non-malleable time-lock puzzles and applications. Cryptology ePrint Archive, Report 2020/779, 2020. https://eprint.iacr.org/2020/779.
25. J. A. Garay, P. D. MacKenzie, M. Prabhakaran, and K. Yang. Resource fairness and composability of cryptographic protocols. In *TCC 2006*, LNCS. Springer, Heidelberg, Mar. 2006.
26. S. D. Gordon and J. Katz. Partial fairness in secure two-party computation. *Journal of Cryptology*, (1), Jan. 2012.
27. C. Hazay, P. Scholl, and E. Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. In *ASIACRYPT 2017, Part I*, LNCS. Springer, Heidelberg, Dec. 2017.
28. J. Katz, J. Loss, and J. Xu. On the security of time-lock puzzles and timed commitments. In *TCC 2020, Part III*, LNCS. Springer, Heidelberg, Nov. 2020.
29. J. Katz, U. Maurer, B. Tackmann, and V. Zikas. Universally composable synchronous computation. In *TCC 2013*, LNCS. Springer, Heidelberg, Mar. 2013.
30. A. Kiayias, A. Russell, B. David, and R. Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *CRYPTO 2017, Part I*, LNCS. Springer, Heidelberg, Aug. 2017.
31. R. Kumaresan and I. Bentov. How to use bitcoin to incentivize correct computations. In *ACM CCS 2014*. ACM Press, Nov. 2014.
32. E. Kushilevitz, Y. Lindell, and T. Rabin. Information-theoretically secure protocols and security under composition. In *38th ACM STOC*. ACM Press, May 2006.
33. Y. Lindell, A. Lysyanskaya, and T. Rabin. Sequential composition of protocols without simultaneous termination. In A. Ricciardi, editor, *PODC 2002*, 2002.
34. K. Pietrzak. Simple verifiable delay functions. In *ITCS 2019*. LIPIcs, Jan. 2019.
35. randao.org. RANDAO: Verifiable random number generation, 2017. https://www.randao.org/whitepaper/Randao_v0.85_en.pdf accessed on 20/02/2020.
36. R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto, 1996.
37. L. Rotem, G. Segev, and I. Shahaf. Generic-group delay functions require hidden-order groups. In *EUROCRYPT 2020, Part III*, LNCS. Springer, Heidelberg, May 2020.
38. V. Shoup. Lower bounds for discrete logarithms and related problems. In *EUROCRYPT'97*, LNCS. Springer, Heidelberg, May 1997.
39. VDF Alliance Team. Vdf alliance, 2020. https://www.vdfalliance.org/what-we-do.
40. B. Wesolowski. Efficient verifiable delay functions. In *EUROCRYPT 2019, Part III*, LNCS. Springer, Heidelberg, May 2019.

Supplementary Material

A Additional Functionalities

We use 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. 18 and Fig. 17.

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 $h \xleftarrow{\$} \{0,1\}^{\ell(\tau)}$ and set $\mathsf{List}_{\mathcal{H}} = \mathsf{List}_{\mathcal{H}} \cup \{(m, h)\}$.
2. If this query is made by $\mathcal{S}$, or if $s \neq \mathsf{sid}$, then add (s, m', h) to the (initially empty) list of illegitimate queries $\mathcal{Q}_s$.
3. Send $(\textsc{Hash-Confirm}, h)$ 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. 17: Restricted observable and programmable global random oracle functionality $\mathcal{G}_{\mathsf{rpoRO}}$ from [11].

A.1 Modeling Rivest *et al.*'s Time-Lock Assumption [36]

We describe in Fig. 19 the ideal functionality $\mathcal{F}_{\mathsf{rsw}}$ from [5] that captures the hardness assumption used by Rivest et al. [36] 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 [38], 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 [5], this definition of $\mathcal{F}_{\mathsf{rsw}}$ is corroborated by a recent result [37] 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

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 $\mathcal{S}$. It is parameterized by the output domain $\mathbb{F}$.

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 $(\mathsf{Abort}, \mathsf{sid}) \in \mathcal{D}$ 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. 18: Functionality $\mathcal{F}_{\mathsf{ct}}^{\Delta}$ for Multiparty Coin Tossing.

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

$\mathcal{F}_{\mathsf{rsw}}$ 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 $(\mathsf{GetElement}, \mathsf{sid}, \mathtt{td}', g, \mathsf{Invalid})$ to $\mathcal{P}_i$. Otherwise, if there is a $\mathtt{el}$ such that $(\mathtt{el}, g) \in \mathsf{group}$ 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 there is no a such that $(\mathtt{el}, a) \in \mathsf{group}$, output $(\mathsf{Pow}, \mathsf{sid}, \mathtt{td}', \mathtt{el}, x, \mathsf{Invalid})$ to $\mathcal{P}_i$. Otherwise, proceed:

1. Convert $x \in \mathbb{Z}$ 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 there is no $\mathtt{el}'$ such that $(\mathtt{el}', y) \in \mathsf{group}$, pick $\mathtt{el}' \xleftarrow{\$} \{0,1\}^\tau$ 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 there is no a, b such that $(\mathtt{el}_1, a), (\mathtt{el}_2, b) \in \mathsf{group}$, then output $(\mathsf{Invalid}, \mathsf{sid})$ to $\mathcal{P}_i$.
2. Compute $c \leftarrow ab \bmod N$. If there is no $\mathtt{el}_3$ such that $(\mathtt{el}_3, c) \in \mathsf{group}$, pick $\mathtt{el}_3 \xleftarrow{\$} \{0,1\}^\tau$ 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 there is no a such that $(\mathtt{el}, a) \in \mathsf{group}$, output $(\mathsf{Invalid}, \mathsf{sid})$ to $\mathcal{P}_i$.
2. Compute $y \leftarrow a^{-1} \bmod N$. If there is no $\mathtt{el}'$ such that $(\mathtt{el}', y) \in \mathsf{group}$, sample $\mathtt{el}' \xleftarrow{\$} \{0,1\}^\tau$ 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 [5] capturing the time lock assumption of [36].

B The Ticked Public Ledger

In order to define a ledger functionality $\mathcal{F}_{\mathsf{Ledger}}$, we adapt ideas from Badertscher et al. [3]. The ledger functionality $\mathcal{F}_{\mathsf{Ledger}}$ is presented in Fig. 20. 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 that can 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).

The ledger functionality of Badertscher et al. [3] 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 some of them 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 [3].

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 $\mathsf{emptyBlocks} \leftarrow \mathsf{maxEmpty}$; else 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. 20: Ledger Functionality $\mathcal{F}_{\mathsf{Ledger}}$.

C Publicly Verifiable Time-Lock Puzzles, continued

In this appendix we will first present the generalized hardness assumption $\mathcal{F}_{\mathsf{tsc}}$ for generic TLPs. Then we will present a protocol that implements $\mathcal{F}_{\mathsf{tlp}}$ using $\mathcal{F}_{\mathsf{tsc}}$.

C.1 Modeling Generic TLPs

We provide an abstract generic sequential computation functionality with a trapdoor in Fig. 21.

In comparison to $\mathcal{F}_{\mathsf{rsw}}$ from [5] (presented in Supplementary Material A) $\mathcal{F}_{\mathsf{tsc}}$ comes without an explicit RSA group embedded into it. Instead, it only operates on a sequence of random values which it extends as is necessary. Any party having a trapdoor will be able to use **RandomAccess** which permits to perform arbitrarily long sequences without any delay, while **Step** can be used by "normal" parties to compute the sequence one step at a tick. Observe that by construction, the results to different parties are consistent.

In comparison to $\mathcal{F}_{\mathsf{rsw}}$ there are some crucial differences: first, we only allow "one" type of step while $\mathcal{F}_{\mathsf{rsw}}$ allows to combine arbitrary RSA-group elements. We think that this is a reasonable restriction, as such an extra functionality does not improve attacks on $\mathcal{F}_{\mathsf{rsw}}$ in generic models [28]. Moreover, as π_{tlp} does not require the use of **GetElement** or **Invert** we can omit these.

Furthermore, the implied structure between different elements as well as the elements themselves can be decided upon by the simulator if the owner is dishonest. This means that the structure is not fixed by the RSA group, but can be arbitrary to the extent that it is consistent between different requests and for different parties. The elements are, on the other hand, random if the functionality was set up by an honest party.

C.2 Constructing the Publicly Verifiable TLP

We now show how to construct a TLP functionality with a public verification interface that is constructed in the $\mathcal{F}_{\mathsf{tsc}}, \mathcal{G}_{\mathsf{rpoRO}}$-hybrid model. We describe Protocol π_{tlp} in Figure 22 and note that it is a generalization of the construction from [5]. 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. [36] are used as setup (we refer the reader to Supplementary Material A for their full descriptions). Our main insights are i) that the specific properties of the RSW puzzle are not necessary in our construction and can be abstracted away into a generic trapdoored sequential computation assumption; and ii) that a puzzle solution $\mathtt{el}, m$ for a puzzle $\mathtt{puz}$ can be publicly 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 $\mathtt{puz} = (\mathtt{el}_0, \varGamma, \mathtt{tag})$ encodes in its $\mathtt{tag}$ both the final state $\mathtt{el}_\varGamma$ obtained after $\varGamma$ computational steps as well as the trapdoor information $\mathtt{td}$ for the functionality $\mathcal{F}_{\mathsf{tsc}}$ that can be used

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

$\mathcal{F}_{\mathsf{tsc}}$ 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. $\mathcal{F}_{\mathsf{tsc}}$ contains a trapdoor string $\mathtt{td}$, a map seq mapping from $\{0,1\}^\tau$ to $\{0,1\}^\tau$ and maps in and out associating parties in $\mathcal{P}$ to a list of entries from $(\{0,1\}^\tau)^2$.

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

1. If $\mathcal{P}_o$ is corrupted then wait for message $(\mathsf{Trapdoor}, \mathsf{sid}, \mathsf{trp})$ from $\mathcal{S}$ with $\mathsf{trp} \in \{0,1\}^\tau$. If $\mathcal{P}_o$ is honest then sample $\mathsf{trp} \leftarrow \{0,1\}^\tau$.
2. Set $\mathtt{td} = \mathsf{trp}$ and output $(\mathsf{Created}, \mathsf{sid}, \mathtt{td})$ to $\mathcal{P}_o$.

RandomElement: Upon receiving $(\mathsf{Rand}, \mathsf{sid}, \mathsf{trp})$ from $\mathcal{P}_i \in \mathcal{P}$, if $\mathsf{trp} \neq \mathtt{td}$, send $(\mathsf{Rand}, \mathsf{sid}, \mathsf{Invalid})$ to $\mathcal{P}_i$. Otherwise do as follows:

1. If $\mathcal{P}_i$ is corrupted then wait for message $(\mathsf{Rand}, \mathsf{sid}, \mathtt{el})$ from $\mathcal{S}$ with $\mathtt{el} \in \{0,1\}^\tau$. If $\mathcal{P}_i$ is honest then sample $\mathtt{el} \leftarrow \{0,1\}^\tau$.
2. If $(\mathtt{el}, \cdot) \notin \mathsf{seq}$ then add $(\mathtt{el}, \perp)$ to seq. Output $(\mathsf{Rand}, \mathsf{sid}, \mathtt{el})$ to $\mathcal{P}_i$.

RandomAccess: Upon receiving $(\mathsf{RandAcc}, \mathsf{sid}, \mathsf{trp}, \mathtt{el}_\mathtt{I}, x)$ from $\mathcal{P}_i \in \mathcal{P}$ with $x \in \mathbb{N}$, if $\mathsf{trp} \neq \mathtt{td}$ or $(\mathtt{el}_\mathtt{I}, \mathtt{nxt}) \notin \mathsf{seq}$ (where $\mathtt{nxt}$ may be $\perp$), output $(\mathsf{RandAcc}, \mathsf{sid}, \mathsf{trp}, \mathtt{el}_\mathtt{I}, x, \mathsf{Invalid})$ to $\mathcal{P}_i$. Otherwise, proceed:

1. Define $\mathtt{el}_\mathtt{O} = \mathtt{el}_\mathtt{I}$ and $y = x$.
2. If $y = 0$ then output $(\mathsf{RandAcc}, \mathsf{sid}, \mathsf{trp}, \mathtt{el}_\mathtt{I}, x, \mathtt{el}_\mathtt{O})$ to $\mathcal{P}_i$.
3. If $(\mathtt{el}_\mathtt{O}, \mathtt{nxt}') \in \mathsf{seq}$ with $\mathtt{nxt}' \neq \perp$ then set $\mathtt{el}_\mathtt{O} = \mathtt{nxt}', y = y - 1$ and go to Step 2. Otherwise do as follows:
   (a) If $\mathcal{P}_o$ is corrupted then send $(\mathsf{Next?}, \mathsf{sid}, \mathtt{el}_\mathtt{O})$ and wait for message $(\mathsf{Next}, \mathsf{sid}, \mathtt{el}_\mathtt{O}, \overline{\mathtt{nxt}})$ from $\mathcal{S}$ with $\overline{\mathtt{nxt}} \in \{0,1\}^\tau$.
   (b) If $\mathcal{P}_o$ is honest then pick a random $\overline{\mathtt{nxt}} \in \{0,1\}^\tau$.
   (c) Replace $(\mathtt{el}_\mathtt{O}, \perp)$ in seq with $(\mathtt{el}_\mathtt{O}, \overline{\mathtt{nxt}})$ and add $(\overline{\mathtt{nxt}}, \perp)$ to seq.
   (d) Set $\mathtt{el}_\mathtt{O} = \overline{\mathtt{nxt}}, y = y - 1$ and go to Step 2.

Step: Upon receiving $(\mathsf{Step}, \mathsf{sid}, \mathtt{el})$ from $\mathcal{P}_i \in \mathcal{P}$:

1. If $(\mathtt{el}, \mathtt{nxt}) \notin \mathsf{seq}$, then output $(\mathsf{Invalid}, \mathsf{sid})$ to $\mathcal{P}_i$.
2. If $(\mathtt{el}, \mathtt{nxt}) \in \mathsf{seq}$ with $\mathtt{nxt} \neq \perp$ then set $\overline{\mathtt{nxt}} = \mathtt{nxt}$. Otherwise do as follows:
   (a) If $\mathcal{P}_o$ is corrupted then send $(\mathsf{Next?}, \mathsf{sid}, \mathtt{el})$ and wait for message $(\mathsf{Next}, \mathsf{sid}, \mathtt{el}, \overline{\mathtt{nxt}})$ from $\mathcal{S}$ with $\overline{\mathtt{nxt}} \in \{0,1\}^\tau$.
   (b) If $\mathcal{P}_o$ is honest then pick a random $\overline{\mathtt{nxt}} \in \{0,1\}^\tau$.
   (c) Replace $(\mathtt{el}, \perp)$ in seq with $(\mathtt{el}, \overline{\mathtt{nxt}})$ and add $(\overline{\mathtt{nxt}}, \perp)$ to seq.
3. Add $(\mathcal{P}_i, (\mathtt{el}, \overline{\mathtt{nxt}}))$ to in and return $(\mathsf{Step}, \mathsf{sid}, \mathtt{el})$ 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. 21: Functionality $\mathcal{F}_{\mathsf{tsc}}$ capturing trapdoor sequential computations.

to compute $\mathtt{el}_\Gamma$ from $\mathtt{el}_0$ in constant time via the **RandomAccess** interface. Given a candidate solution $\mathtt{el}, m$ for $\mathtt{puz}$, the verifier can confirm that $\mathtt{tag}$ does encode $\texttt{el}$ as $\texttt{el}_{\Gamma}$ and recompute m in constant time, since it also uses $\texttt{el}$ to recover $\texttt{td}$. The use of a global $\mathcal{F}_{\mathsf{tsc}}$ for the computation and of a global random oracle $\mathcal{G}_{\mathsf{rpoRO}}$ for generating $\texttt{tag}$ 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 $\texttt{puz}, \texttt{st}$ 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 $\texttt{puz}$ in constant time. We formally state the security of Protocol π_{tlp} in Theorem 1, which we recall below for the sake of clarity.

Theorem 1 *Protocol π_{tlp} UC-realizes $\mathcal{F}_{\mathsf{tlp}}$ in the $\mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{tsc}}$-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 $\mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{tsc}}$ 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 $\mathcal{G}_{\mathsf{rpoRO1}}, \mathcal{G}_{\mathsf{rpoRO2}}, \mathcal{F}_{\mathsf{tsc}}$ unless otherwise stated. $\mathcal{S}$ is presented in Figure 23 (Corrupted $\mathcal{P}_o$) and in Figure 24 (Honest $\mathcal{P}_o$). It forwards messages from $\mathcal{A}$ and simulated hybrid functionalities to $\mathcal{G}_{\mathsf{ticker}}$. For any environment $\mathcal{Z}$, we argue that an execution with $\mathcal{S}$ and $\mathcal{F}_{\mathsf{tlp}}$ is indistinguishable from an execution with $\mathcal{A}$ in the $\mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{tsc}}$-hybrid model.

First, we observe that, apart from the **Public Verification** interface, $\mathcal{S}$ is constructed exactly as the simulator in Theorem 2 of [5]. We therefore only have to consider the changes due to **Public Verification**.

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 $\mathcal{G}_{\mathsf{rpoRO1}}, \mathcal{G}_{\mathsf{rpoRO2}}, \mathcal{F}_{\mathsf{tsc}}$. Notice that the **Public Verification** procedure is executed locally by a verifier who has received $\texttt{puz}, \texttt{el}, m$ and that it corresponds exactly to executing the steps of the **Get Message** procedure with input $\texttt{puz}, \texttt{el}$ 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 $\mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{tsc}}$-hybrid model as proven in [5], the output of the simulated **Public Verification** procedure with input $\texttt{puz} = (\texttt{el}_0, \Gamma, \texttt{tag} = (\texttt{tag}_1, \texttt{tag}_2)), \texttt{el}', m'$ where $\texttt{puz}$ was generated with message m and trapdoor $\texttt{td}$ will only differ from that of $\mathcal{F}_{\mathsf{tlp}}$ if $\mathcal{A}$ finds $\texttt{el}', m' \neq m$ such that querying $\mathcal{G}_{\mathsf{rpoRO1}}$ with $(\textsc{Hash-Query}, (\texttt{el}_0|\texttt{el}'))$ yields $h_1 = \texttt{tag}_1$ where $(m'|\texttt{td}) = \texttt{tag}_1 \oplus h_1$ and that querying $\mathcal{G}_{\mathsf{rpoRO2}}$ with $(\textsc{Hash-Query}, (h_1|m'|\texttt{td}))$ yields $h_2 = \texttt{tag}_2$, which only happens with negligible probability in τ since $\mathcal{A}$ can only make a polynomial number of queries to $\mathcal{G}_{\mathsf{rpoRO1}}, \mathcal{G}_{\mathsf{rpoRO2}}$. □

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 $\mathcal{TAG} = \{0,1\}^{2\cdot\tau} \times \{0,1\}^\tau$. 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 $\mathcal{F}_{\mathsf{tsc}}$, $\mathcal{G}_{\mathsf{rpoRO1}}$ (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 $\{0,1\}^{4\cdot\tau}$ 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 $\mathcal{F}_{\mathsf{tsc}}$ obtaining $(\mathsf{Created}, \mathsf{sid}, \mathtt{td})$.
2. Send $(\mathsf{Rand}, \mathsf{sid}, \mathtt{td})$ to $\mathcal{F}_{\mathsf{tsc}}$, obtaining $(\mathsf{Rand}, \mathsf{sid}, \mathtt{el}_0)$.
3. Send $(\mathsf{RandAcc}, \mathsf{sid}, \mathtt{td}, \mathtt{el}_0, \Gamma)$ to $\mathcal{F}_{\mathsf{tsc}}$, obtaining $(\mathsf{RandAcc}, \mathsf{sid}, \mathtt{td}, \mathtt{el}_0, \Gamma, \mathtt{el}_\Gamma)$.
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. Set $\mathtt{tag}_1 = h_1 \oplus (m|\mathtt{td})$, $\mathtt{tag}_2 = h_2$ and $\mathtt{tag} = (\mathtt{tag}_1, \mathtt{tag}_2)$, and output $(\mathsf{CreatedPuzzle}, \mathsf{sid}, \mathsf{puz} = (\mathtt{el}_0, \Gamma, \mathtt{tag}), \mathtt{el}_\Gamma)$.

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

Get Message: Upon receiving $(\mathsf{GetMsg}, \mathsf{sid}, \mathsf{puz}, \mathtt{el})$ as input, a party $\mathcal{P}_i \in \mathcal{P}$ parses $\mathsf{puz} = (\mathtt{el}_0, \Gamma, \mathtt{tag})$, 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 $(\mathsf{RandAcc}, \mathsf{sid}, \mathtt{td}, \mathtt{el}_0, \Gamma)$ to $\mathcal{F}_{\mathsf{tsc}}$, obtaining $(\mathsf{RandAcc}, \mathsf{sid}, \mathtt{td}, \mathtt{el}_0, \Gamma, \mathtt{el}_\Gamma)$.
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 $b_1 = 1$ or $b_2 = 1$.
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 $(\mathsf{Verify}, \mathsf{sid}, \mathsf{puz}, \mathtt{st}, m)$, a verifier $\mathcal{V}_i$ executes the steps of **Get Message** with input $(\mathsf{GetMsg}, \mathsf{puz}, \mathtt{el})$ 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 $(\mathsf{Verified}, \mathsf{sid}, \mathsf{puz}, \mathtt{st}, m, b)$.

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

Fig. 22: Protocol π_{tlp} realizing publicly verifiable time-lock puzzle functionality $\mathcal{F}_{\mathsf{tlp}}$ in the $\mathcal{F}_{\mathsf{tsc}}, \mathcal{G}_{\mathsf{rpoRO}}$-hybrid model.

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

Simulator $\mathcal{S}$ interacts with environment $\mathcal{Z}$, functionalities $\mathcal{F}_{\mathsf{tlp}}, \mathcal{G}_{\mathsf{rpoRO1}}, \mathcal{G}_{\mathsf{rpoRO2}}, \mathcal{F}_{\mathsf{tsc}}$ 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 $\mathcal{F}_{\mathsf{tsc}}$ 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 requests 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 initially empty lists $\mathsf{tag}\text{-}\overline{\mathsf{tag}}, \mathsf{el}\text{-}\mathsf{st}$ which contain translations between the tags and element labels of $\mathcal{F}_{\mathsf{tsc}}$ and $\mathcal{F}_{\mathsf{tlp}}$ respectively.

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 $(\mathsf{RandAcc}, \mathsf{sid}, \mathtt{td}, \mathtt{el}_0, \Gamma)$ to $\mathcal{F}_{\mathsf{tsc}}$, obtaining $(\mathsf{RandAcc}, \mathsf{sid}, \mathtt{td}, \mathtt{el}_0, \Gamma, \mathtt{el}_\Gamma)$. Check that there exists a request $(\textsc{Hash-Query}, (\mathtt{el}_0|\mathtt{el}_\Gamma))$ 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 above checks fails, 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 $(\mathsf{RandAcc}, \mathsf{sid}, \mathtt{td}, \mathtt{el}_j, 1)$ to $\mathcal{F}_{\mathsf{tsc}}$, obtaining $(\mathsf{RandAcc}, \mathsf{sid}, \mathtt{td}, \mathtt{el}_j, 1, \mathtt{el}_{j+1})$.
2. Sample $\overline{\mathtt{tag}} \xleftarrow{\$} \mathcal{TAG}$, append $(\mathtt{tag}, \overline{\mathtt{tag}})$ to $\mathsf{tag}\text{-}\overline{\mathsf{tag}}$.
3. Send $(\mathsf{CreatePuzzle}, \mathsf{sid}, \Gamma, m)$ to $\mathcal{F}_{\mathsf{tlp}}$ and provide $\mathtt{st}_0, \dots, \mathtt{st}_\Gamma, \overline{\mathtt{tag}}$.

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

– If there is $\mathtt{el}$ such that $(\mathtt{el}, \mathtt{st}) \in \mathsf{el}\text{-}\mathsf{st}$, send $(\mathsf{RandAcc}, \mathsf{sid}, \mathtt{td}, \mathtt{el}, 1)$ to $\mathcal{F}_{\mathsf{tsc}}$, obtaining $(\mathsf{RandAcc}, \mathsf{sid}, \mathtt{td}, \mathtt{el}, 1, \mathtt{el}')$. Otherwise, send $(\mathsf{Rand}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{tsc}}$, obtaining $(\mathsf{Rand}, \mathsf{sid}, \mathtt{el}')$.

– If there is no $\mathtt{st}'$ such that $(\mathtt{el}', \mathtt{st}') \in \mathsf{el}\text{-}\mathsf{st}$, then 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 $\mathtt{puz} = (\mathtt{st}_0, \Gamma, \overline{\mathtt{tag}})$ 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 $(\mathsf{RandAcc}, \mathsf{sid}, \mathtt{td}, \mathtt{el}_0, \Gamma)$ to $\mathcal{F}_{\mathsf{tsc}}$, obtaining $(\mathsf{RandAcc}, \mathsf{sid}, \mathtt{td}, \mathtt{st}_0, \Gamma, \mathtt{el}_\Gamma)$. Check that $\mathtt{tag}_2 = h_2$ and $\mathtt{el} = \mathtt{el}_\Gamma$.

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

Fig. 23: Simulator $\mathcal{S}$ for the case of a corrupted $\mathcal{P}_o$ in π_{tlp}.

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

Simulator $\mathcal{S}$ interacts with environment $\mathcal{Z}$, functionalities $\mathcal{F}_{\mathsf{tlp}}, \mathcal{G}_{\mathsf{rpoRO1}}, \mathcal{G}_{\mathsf{rpoRO2}}, \mathcal{F}_{\mathsf{tsc}}$ 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 $\mathcal{F}_{\mathsf{tsc}}$ unless explicitly stated, keeping lists of all such requests. However, for every query $(\textsc{IsProgrammed}, m)$ to $\mathcal{G}_{\mathsf{rpoRO1}}$ or $\mathcal{G}_{\mathsf{rpoRO2}}$, $\mathcal{S}$ answers with $(\textsc{IsProgrammed}, 0)$ if m has been programmed by $\mathcal{S}$ itself. $\mathcal{S}$ keeps initially empty lists $\mathsf{el\text{-}st}, \mathsf{next}$.

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 $\mathcal{F}_{\mathsf{tsc}}$ obtaining $(\mathsf{Created}, \mathsf{sid}, \mathtt{td})$. Send $(\mathsf{Rand}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{tsc}}$, obtaining $(\mathsf{Rand}, \mathsf{sid}, \mathtt{el}_0)$. Send $(\mathsf{RandAcc}, \mathsf{sid}, \mathtt{td}, \mathtt{el}_0, \Gamma)$ to $\mathcal{F}_{\mathsf{tsc}}$, obtaining $(\mathsf{RandAcc}, \mathsf{sid}, \mathtt{td}, \mathtt{el}, \Gamma, \mathtt{el}_\Gamma)$.
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 $(\mathsf{Step}, \mathsf{sid}, \mathtt{el})$ to $\mathcal{F}_{\mathsf{tsc}}$ on behalf of $\mathcal{P}_i \in \mathcal{P} \setminus \mathcal{P}_o$ such that there is an $\mathtt{el}$ such that $(\mathtt{el}, \mathtt{st}) \in \mathsf{el\text{-}st}$, $\mathcal{S}$ proceeds as follows:

1. Send $(\mathsf{RandAcc}, \mathsf{sid}, \mathtt{td}, \mathtt{el}, 1)$ to $\mathcal{F}_{\mathsf{tsc}}$, obtaining $(\mathsf{RandAcc}, \mathsf{sid}, \mathtt{td}, \mathtt{el}, 1, \mathtt{el}')$.
2. If there is no entry $(\mathtt{el}', \mathtt{st}') \in \mathsf{el\text{-}st}$ for any $\mathtt{st}'$, 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 $\mathcal{F}_{\mathsf{tsc}}$ 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 $\mathcal{P}_i \in \mathcal{P} \setminus \mathcal{P}_o$, 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 $\mathcal{F}_{\mathsf{tsc}}$ 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. 24: Simulator $\mathcal{S}$ for the case of an honest $\mathcal{P}_o$ in π_{tlp}.

D Security Analysis of Protocol π_{VDF}

Before proceeding to the security analysis of our UC VDF construction π_{VDF}, we will discuss the main details and particularities of our formulations of a stand alone verifiable sequential computation scheme $\mathcal{F}_{\mathsf{psc}}$ and of a UC continuous VDF $\mathcal{F}_{\mathsf{VDF}}$. We then recall Theorem 2 and present a simulator for π_{VDF} along with the security proof.

D.1 Functionalities $\mathcal{F}_{\mathsf{psc}}$ and $\mathcal{F}_{\mathsf{VDF}}$

Functionality $\mathcal{F}_{\mathsf{psc}}$ (described in Figure 5) captures the notion of a generic stand alone verifiable sequential computation scheme (weaker than a continuous VDF as defined in [23]) in a similar way as the iterated squaring assumption from [36] is captured in [5]. Basically, this functionality allows the evaluation of computational steps taking as input a initial state $\mathtt{el}$ and outputting a state $\mathtt{nxt}$ after a computational delay, which is modeled as a tick. After a number k of such evaluation steps are computed departing from an initial step $\mathtt{el}_{\mathtt{I}}$ to a final step $\mathtt{el}_{\mathtt{O}}$, $\mathcal{F}_{\mathsf{psc}}$ allows for the generation of a proof π' that $\mathtt{el}_{\mathtt{O}}$ was obtained from $\mathtt{el}_{\mathtt{I}}$ through k computational steps upon being queried with all intermediate steps $\mathtt{el}_{\mathtt{I}}, \mathtt{el}_2, \ldots, \mathtt{el}_{k-1}, \mathtt{el}_{\mathtt{O}}$. Later on, this proof π' can be verified with respect to $\mathtt{el}_{\mathtt{I}}, \mathtt{el}_{\mathtt{O}}, k$ in time essentially independent from the number of steps k. Since current techniques [23] for generating and verifying such a proof do require non-constant computational time, we model the number of ticks necessary for generating/verifying the proof by functions $f(\mathtt{el}_{\mathtt{I}}, \mathtt{el}_2, \ldots, \mathtt{el}_{k-1}, \mathtt{el}_{\mathtt{O}})$ and $g(\mathtt{el}_{\mathtt{I}}, \mathtt{el}_{\mathtt{O}}, k, \pi')$.

Notice that we need this functionality to capture a stand alone verifiable sequential computation because, as observed in [5], exposing the actual states from a concrete computational problem would allow the environment to perform several computational steps without activating other parties. For example, if this scheme is instantiated from the iterated squaring assumption of [36] where the evaluation consists in computing a sequence of squarings starting from an element of the group of residues modulo an RSA modulus N, revealing the group structure (*i.e.* N) and the representation of the group elements would allow the environment to compute any number of squarings before activating the other parties. Hence, we must embed the computational problem in a functionality. However, notice that this functionality does not guarantee that the states it outputs are uniformly random or non-malleable, as it allows the adversary to choose the representation of each state, which will be crucial in our proof. What $\mathcal{F}_{\mathsf{psc}}$ does guarantee is that proofs are only generated and successfully verified if the claimed number of computational steps is indeed correct, also guaranteeing that the transitions between states $\mathtt{el}$ and $\mathtt{nxt}$ is injective.

We model a UC VDF in Functionality $\mathcal{F}_{\mathsf{VDF}}$ (described in Figure 6), This functionality guarantees the properties expected from a continuous VDF. It ensures that each computational step taken in evaluating the VDF takes at least a fixed amount of time (one tick) and guarantees that the output obtained after a number of steps is (close to) uniformly random and unpredictable even to the adversary. Since it models a continuous VDF, $\mathcal{F}_{\mathsf{VDF}}$ allows for callers to obtain the output of evaluating an input for after each step that is executed, without requiring the number of steps that will be evaluated to be known *a priori*. Naturally, $\mathcal{F}_{\mathsf{VDF}}$ also provides a proof that each output has been correctly obtained by computing a certain number of steps on a given input. As it is the case with $\mathcal{F}_{\mathsf{psc}}$, the time required to generate and verify such proofs is variable and modeled as functions $f(\mathtt{st}_1, \ldots, \mathtt{st}_\Gamma)$ and $g(in, out, \Gamma, \pi)$, respectively. Moreover, similarly to $\mathcal{F}_{\mathsf{psc}}$, $\mathcal{F}_{\mathsf{VDF}}$ allows the ideal adversary to choose the representation of intermediate computational steps involved in evaluating the VDF, even though the output is guaranteed to be random. Again this is necessary in order to construct a simulator. Another particularity of $\mathcal{F}_{\mathsf{VDF}}$ used in the proof is a leakage of each evaluation performed by an honest party at the tick when the result is returned to the original caller. This leakage does not affect the soundness of the VDF nor the randomness of its output, but is necessary for consistently simulating an execution.

Instantiating $\mathcal{F}_{\mathsf{psc}}$ and implications to π_{VDF}. Functionality $\mathcal{F}_{\mathsf{psc}}$ can be instantiated from the UC formulation of the iterated squarings problem from [5] and the soundness of Fiat-Shamir using the techniques from [23] in the global random oracle model, which is anyway necessary for obtaining such UC-secure time-based primitives as shown in [5]. In this case, the times for evaluating computational steps and generating/verifying proofs is equivalent to those of [23], which allows for efficient generation of proofs of computation for any newly computed states departing from the initial state, without the need to know the number of steps to be computed beforehand. However, if $\mathcal{F}_{\mathsf{psc}}$ was to be instantiated with a standard VDF (*e.g.* [40,34]), the effect it would have on π_{VDF} is that the proof generation/verification times would grow if the caller tried to generate a proof for arbitrary states. Hence, we would be implementing a $\mathcal{F}_{\mathsf{VDF}}$ with proof generation/verification times that crucially depend on the number of steps evaluated. However, if we restrict the caller to only generating proofs for a number computational steps known *a priori*, our constructions can be instantiated from schemes like [40,34].

D.2 Proof

We recall Theorem 2 and present a proof.

Theorem 2 *Protocol π_{VDF} UC-realizes $\mathcal{F}_{\mathsf{VDF}}$ in the $\mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{psc}}$-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_{VDF} composed with $\mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{psc}}$ and $\mathcal{A}$ from $\mathcal{S}$ composed with $\mathcal{F}_{\mathsf{VDF}}$.*

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 $\mathcal{G}_{\mathsf{rpoRO2}}, \mathcal{F}_{\mathsf{psc}}$ 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 $\mathcal{F}_{\mathsf{VDF}}$ is indistinguishable from an execution of π_{VDF} with $\mathcal{A}$ in the $\mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{psc}}$-hybrid model.

We describe simulator $\mathcal{S}$ in Figures 25 and 26. The main idea of this simulator is that $\mathcal{S}$ observes the adversary $\mathcal{A}$'s queries to $\mathcal{F}_{\mathsf{psc}}$ in order simulate matching queries to $\mathcal{F}_{\mathsf{VDF}}$. $\mathcal{S}$ answers the queries from both $\mathcal{F}_{\mathsf{psc}}$ and $\mathcal{F}_{\mathsf{VDF}}$ when asked for the state values for each of the new intermediate states by selecting random state values and providing the same value to both functionalities in such a way that $\mathcal{F}_{\mathsf{psc}}$ outputs to $\mathcal{A}$ the same states outputted by $\mathcal{F}_{\mathsf{VDF}}$ when queried on the same input state. When $\mathcal{A}$ queries $\mathcal{F}_{\mathsf{psc}}$ on $(\mathsf{Prove}, \mathsf{sid}, \mathtt{el}_1, \dots, \mathtt{el}_\Gamma)$, $\mathcal{S}$ simulates the **Get Output** procedure by querying $\mathcal{F}_{\mathsf{VDF}}$ with $(\mathsf{GetOutput}, \mathsf{sid}, \mathtt{st}_1, \dots, \mathtt{st}_\Gamma)$. Later on, when $\mathcal{F}_{\mathsf{VDF}}$ answers with the output *out*, $\mathcal{S}$ programs $\mathcal{G}_{\mathsf{rpoRO2}}$ on $(sid|\Gamma|\mathtt{st}_\Gamma|\pi')$ so that it answers with *out* when $\mathcal{A}$ computes the VDF output. The verification procedure is simulated by simply allowing $\mathcal{A}$ to make the necessary queries to $\mathcal{F}_{\mathsf{psc}}$ and $\mathcal{G}_{\mathsf{rpoRO2}}$, since by that point the answers to verification queries are already set up in such a way that verification succeeds if and only if it would have succeeded with $\mathcal{F}_{\mathsf{VDF}}$.

The crux of this strategy is that $\mathcal{S}$ essentially only deviates from the protocol in the equivocation step using $\mathcal{G}_{\mathsf{rpoRO2}}$ and only risks making $\mathcal{F}_{\mathsf{VDF}}$ and $\mathcal{F}_{\mathsf{psc}}$ inconsistent when choosing values for states. Other than that, queries to $\mathcal{F}_{\mathsf{psc}}$ are simulated towards the adversary exactly as they should, keeping the correspondence between states of $\mathcal{F}_{\mathsf{psc}}$ and states of $\mathcal{F}_{\mathsf{VDF}}$. The simulation fails if a query on the last $\mathcal{F}_{\mathsf{psc}}$ state $\mathtt{st}_\Gamma$ is issued to $\mathcal{G}_{\mathsf{rpoRO2}}$ before $\mathcal{S}$ obtains the final output from $\mathcal{F}_{\mathsf{VDF}}$. However, this only happens if $\mathcal{A}$ manages to compute Γ computational steps from $\mathtt{st}_1$ to $\mathtt{st}_\Gamma$ in less ticks than it takes to evaluate the input in corresponding to $\mathtt{st}_1$ with $\mathcal{F}_{\mathsf{VDF}}$. $\mathcal{A}$ can only do so by guessing $\mathtt{st}_\Gamma$ (or an intermediate step), which only happens with probability negligible in the computational security parameter τ. Other than that, the simulation can only fail if $\mathcal{S}$ provides a value that is already in use for a new state when queried by $\mathcal{F}_{\mathsf{psc}}$ and $\mathcal{F}_{\mathsf{VDF}}$. However, $\mathcal{S}$ samples these values uniformly at random, meaning such a collision only happens with negligible probability. Hence, we conclude that the execution with $\mathcal{S}$ and $\mathcal{F}_{\mathsf{VDF}}$ is indistinguishable from the execution of π_{VDF} with $\mathcal{A}$ in the $\mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{psc}}$-hybrid model. $\square$

Part 1 of Simulator $\mathcal{S}$ for π_{VDF}

Simulator $\mathcal{S}$ interacts with environment $\mathcal{Z}$, functionalities $\mathcal{F}_{\mathsf{VDF}}, \mathcal{G}_{\mathsf{rpoRO2}}, \mathcal{F}_{\mathsf{psc}}$ and an internal copy of an $\mathcal{A}$ corrupting one or more parties $\mathcal{P}_i \in \mathcal{P}$. $\mathcal{S}$ forwards all messages between $\mathcal{A}$ and $\mathcal{Z}$. Moreover, $\mathcal{S}$ forwards all queries to $\mathcal{G}_{\mathsf{rpoRO2}}$ and $\mathcal{F}_{\mathsf{psc}}$ unless explicitly stated, keeping lists of all such requests. However, for every query $(\textsc{IsProgrammed}, m)$ to $\mathcal{G}_{\mathsf{rpoRO2}}$, $\mathcal{S}$ answers with $(\textsc{IsProgrammed}, 0)$ if m has been programmed by $\mathcal{S}$ itself. $\mathcal{S}$ keeps initially empty lists $\mathsf{pnext}, \mathsf{steps}$. For every corrupted party $\mathcal{P}_i \in \mathcal{P}$, $\mathcal{S}$ maintains initially empty lists VL_i^1, VL_i^2, VL_i^3 (for messages from $\mathcal{F}_{\mathsf{VDF}}$) and PL_i^1, PL_i^2, PL_i^3 (for messages from $\mathcal{F}_{\mathsf{psc}}$).

Solve: Upon receiving a query $(\mathsf{Step}, \mathsf{sid}, \mathtt{el})$ from $\mathcal{A}$ on behalf of $\mathcal{P}_i \in \mathcal{P}$ to $\mathcal{F}_{\mathsf{psc}}$, $\mathcal{S}$ sends $(\mathsf{Step}, \mathsf{sid}, \mathtt{el})$ to $\mathcal{F}_{\mathsf{psc}}$ and $(\mathsf{Solve}, \mathsf{sid}, \mathtt{el})$ to $\mathcal{F}_{\mathsf{VDF}}$ on behalf of $\mathcal{P}_i$. $\mathcal{S}$ simulates a VDF evaluation step with $\mathcal{F}_{\mathsf{VDF}}$ and later ensures that the state obtained by $\mathcal{A}$ from $\mathcal{F}_{\mathsf{psc}}$ is consistent with that provided by $\mathcal{F}_{\mathsf{VDF}}$ (following the steps in the **Tick** interface).

Get Output: If $\mathcal{A}$ sends $(\mathsf{Prove}, \mathsf{sid}, \mathtt{el}_1, \ldots, \mathtt{el}_\Gamma)$ to $\mathcal{F}_{\mathsf{psc}}$ on behalf of $\mathcal{P}_i \in \mathcal{P}$, $\mathcal{S}$ proceeds as follows:

1. Send $(\mathsf{Prove}, \mathsf{sid}, \mathtt{el}_1, \ldots, \mathtt{el}_\Gamma)$ to $\mathcal{F}_{\mathsf{psc}}$ on behalf of $\mathcal{P}_i$.
2. Send $(\mathsf{GetOutput}, \mathsf{sid}, \mathtt{st}_1, \ldots, \mathtt{st}_\Gamma)$ to $\mathcal{F}_{\mathsf{VDF}}$ on behalf of $\mathcal{P}_i$.
3. Sample $\pi' \xleftarrow{\$} \{0,1\}^\tau$ and add $(\Gamma, \mathtt{el}_1, \mathtt{el}_\Gamma, \pi')$ to pnext.

As described in the **Tick** interface, in the appropriate tick, $\mathcal{S}$ sends the value π' when queried with $(\mathsf{ProofStr}, \mathsf{sid}, \mathcal{P}_i, \mathtt{el}_1, \mathtt{el}_\Gamma)$ by $\mathcal{F}_{\mathsf{psc}}$ and $\pi = (\mathtt{el}_\Gamma, \pi')$ when queried with $(\mathsf{ProofStr}, \mathsf{sid}, \mathcal{P}_i, \mathtt{st}_1, \ldots, \mathtt{st}_\Gamma)$ by $\mathcal{F}_{\mathsf{VDF}}$. $\mathcal{S}$ also programs $\mathcal{G}_{\mathsf{rpoRO2}}$ on $(sid|\Gamma|\mathtt{el}_\Gamma|\pi')$ so that it answers with the output out provided by $\mathcal{F}_{\mathsf{VDF}}$. After that, $\mathcal{S}$ forwards queries to $\mathcal{G}_{\mathsf{rpoRO2}}$ and $\mathcal{F}_{\mathsf{psc}}$ from $\mathcal{A}$ on behalf of corrupted parties $\mathcal{P}_i \in \mathcal{P}$, allowing $\mathcal{A}$ to perform the necessary steps for **Get Output**. However, for every query $(\textsc{IsProgrammed}, m)$ to $\mathcal{G}_{\mathsf{rpoRO2}}$, $\mathcal{S}$ always answers with $(\textsc{IsProgrammed}, 0)$ if $\mathcal{S}$ has programmed $\mathcal{G}_{\mathsf{rpoRO2}}$ on message m.

Verify: $\mathcal{S}$ simulates this step exactly as in π_{VDF}, forwarding all messages between $\mathcal{A}$ and $\mathcal{G}_{\mathsf{rpoRO2}}$ and $\mathcal{F}_{\mathsf{psc}}$ from $\mathcal{A}$ on behalf of corrupted parties $\mathcal{P}_i \in \mathcal{P}$, with the exception of answering every query $(\textsc{IsProgrammed}, m)$ to $\mathcal{G}_{\mathsf{rpoRO2}}$ with $(\textsc{IsProgrammed}, 0)$ if $\mathcal{S}$ has programmed $\mathcal{G}_{\mathsf{rpoRO2}}$ on message m. Notice that the queries to $\mathcal{G}_{\mathsf{rpoRO2}}$ and $\mathcal{F}_{\mathsf{psc}}$ are already adjusted such that verification succeeds if and only if (in, out, Γ, π) has been computed correctly.

Fetch State: Upon receiving message $(\mathsf{Output}, \mathsf{sid})$ from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{psc}}$ on behalf of $\mathcal{P}_i \in \mathcal{P}$, $\mathcal{S}$ sends $(\mathsf{Complete}, \mathsf{sid}, PL_i^1, PL_i^2, PL_i^3)$ to $\mathcal{A}$.

Fig. 25: Part 1 of Simulator $\mathcal{S}$ for π_{VDF}.

Part 2 of Simulator $\mathcal{S}$ for π_{VDF}

Tick: Immediately after each tick, $\mathcal{S}$ keeps track of new messages, answers $\mathcal{F}_{\mathsf{psc}}, \mathcal{F}_{\mathsf{VDF}}$ and programs $\mathcal{G}_{\mathsf{rpoRO}}$ as needed.

– $\mathcal{S}$ sends $(\mathsf{Output}, \mathsf{sid})$ (resp. $(\mathsf{Fetch}, \mathsf{sid})$) to $\mathcal{F}_{\mathsf{psc}}$ (resp. $\mathcal{F}_{\mathsf{VDF}}$), receiving $(\mathsf{Complete}, \mathsf{sid}, \overline{PL_i^1}, \overline{PL_i^2}, \overline{PL_i^3})$ (resp. $(\mathsf{Fetch}, \mathsf{sid}, \overline{VL_i^1}, \overline{VL_i^2}, \overline{VL_i^3})$). If $(\overline{PL_i^1}, \overline{PL_i^2}, \overline{PL_i^3}) = (PL_i^1, PL_i^2, PL_i^3)$ and $(\overline{VL_i^1}, \overline{VL_i^2}, \overline{VL_i^3}) = (VL_i^1, VL_i^2, VL_i^3)$, $\mathcal{S}$ does nothing. Otherwise, $\mathcal{S}$ sets $(PL_i^1, PL_i^2, PL_i^3) \leftarrow (\overline{PL_i^1}, \overline{PL_i^2}, \overline{PL_i^3})$, sets $(VL_i^1, VL_i^2, VL_i^3) \leftarrow (\overline{VL_i^1}, \overline{VL_i^2}, \overline{VL_i^3})$. For every corrupted $\mathcal{P}_i$ with updated lists (PL_i^1, PL_i^2, PL_i^3) and (VL_i^1, VL_i^2, VL_i^3):
  1. For all messages $(\mathcal{P}_i, (\mathtt{el}, \mathtt{nxt}))$ from PL_i^1 or $(\mathcal{P}_i, \mathtt{el}, \mathtt{nxt})$ from VL_i^1, if there is no $(\mathtt{el}, \mathtt{nxt}) \in \mathsf{steps}$, add $(\mathtt{el}, \mathtt{nxt})$ to steps.
  2. For all messages $(\mathcal{P}_i, (\mathtt{el}_1, \mathtt{el}_\Gamma, \Gamma, \pi'))$ from PL_i^2 and $(\mathcal{P}_i, (\mathtt{el}_1, out, \Gamma, \pi))$ from VL_i^2 where $\pi = (\mathtt{el}_\Gamma, \pi')$ such that $(\Gamma, \mathtt{el}_1, \mathtt{el}_\Gamma, \pi') \in \mathsf{pnext}$, $\mathcal{S}$ sends $(\textsc{Program-RO}, (sid|\Gamma|\mathtt{el}_\Gamma|\pi'), out)$ to $\mathcal{G}_{\mathsf{rpoRO2}}$ and removes $(\Gamma, \mathtt{el}_1, \mathtt{el}_\Gamma, \pi')$ from pnext. Since π' 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 is negligible. This step ensures that $\mathcal{A}$ obtains the output generated by $\mathcal{F}_{\mathsf{VDF}}$.

  For every honest $\mathcal{P}_i$ with an updated list (VL_i^1, VL_i^2, VL_i^3):
  1. For all messages $(\mathcal{P}_i, \mathtt{st}, \mathtt{st}')$ from VL_i^1, $\mathcal{S}$ sends $(\mathsf{Step}, \mathsf{sid}, \mathtt{st})$ to $\mathcal{F}_{\mathsf{psc}}$ and adds $(\mathtt{st}, \mathtt{st}')$ to steps. This step simulates honest parties evaluating the VDF with $\mathcal{F}_{\mathsf{VDF}}$, performing the same steps with $\mathcal{F}_{\mathsf{psc}}$ and ensuring the same intermediate states are registered by $\mathcal{F}_{\mathsf{psc}}$.
  2. For all messages $(\mathcal{P}_i, (\mathtt{st}_1, out, \Gamma, \pi))$ from VL_i^2 where $\pi = (\mathtt{el}_\Gamma, \pi')$, $\mathcal{S}$ sends $(\textsc{Program-RO}, (\Gamma|\mathtt{el}_\Gamma|\pi'), out)$ to $\mathcal{G}_{\mathsf{rpoRO2}}$. Since π' is randomly sampled by $\mathcal{F}_{\mathsf{VDF}}$ and still unknown to $\mathcal{A}$, $\mathcal{Z}$ or any other party at this point, the probability that this programming fails is negligible. This step simulates honest parties obtaining an output and proof after evaluating the VDF with $\mathcal{F}_{\mathsf{VDF}}$, performing the necessary steps with $\mathcal{F}_{\mathsf{psc}}$ to obtain π' and programming $\mathcal{G}_{\mathsf{rpoRO2}}$ so that the same output is computed.

– The following answers to queries from $\mathcal{F}_{\mathsf{psc}}$ and $\mathcal{F}_{\mathsf{VDF}}$ ensure that the states and proofs obtained by $\mathcal{A}$ when querying $\mathcal{F}_{\mathsf{psc}}$ are consistent with those provided by $\mathcal{F}_{\mathsf{VDF}}$:
  - Upon receiving $(\mathsf{Step}, \mathsf{sid}, \mathtt{el})$ from $\mathcal{F}_{\mathsf{psc}}$, if there is no $(\mathtt{el}, \overline{\mathtt{nxt}}) \in \mathsf{steps}$, sample $\overline{\mathtt{nxt}} \xleftarrow{\$} \{0,1\}^\tau$ and add $(\mathtt{el}, \overline{\mathtt{nxt}})$ to steps. Send $(\mathsf{Step}, \mathsf{sid}, \mathtt{el}, \overline{\mathtt{nxt}})$ to $\mathcal{F}_{\mathsf{psc}}$.
  - Upon receiving $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st})$ from $\mathcal{F}_{\mathsf{VDF}}$, if there is no $(\mathtt{st}, \mathtt{st}') \in \mathsf{steps}$, sample $\mathtt{st}' \xleftarrow{\$} \{0,1\}^\tau$ and add $(\mathtt{st}, \mathtt{st}')$ to steps. Send $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st}, \mathtt{st}')$ to $\mathcal{F}_{\mathsf{VDF}}$.
  - Upon receiving $(\mathsf{ProofStr}, \mathsf{sid}, \mathcal{P}_i, \mathtt{el}_1, \ldots, \mathtt{el}_\Gamma)$ from $\mathcal{F}_{\mathsf{psc}}$ such that $(\Gamma, \mathtt{el}_1, \mathtt{el}_\Gamma, \pi') \in \mathsf{pnext}$, send $(\mathsf{ProofStr}, \mathsf{sid}, \mathcal{P}_i, \mathtt{el}_1, \mathtt{el}_\Gamma, \pi')$ to $\mathcal{F}_{\mathsf{psc}}$.
  - Upon receiving $(\mathsf{ProofStr}, \mathsf{sid}, \mathcal{P}_i, \mathtt{el}_1, \ldots, \mathtt{el}_\Gamma)$ from $\mathcal{F}_{\mathsf{VDF}}$ to $\mathcal{S}$ such that $(\Gamma, \mathtt{el}_1, \mathtt{el}_\Gamma, \pi') \in \mathsf{pnext}$, send response $(\mathsf{ProofStr}, \mathsf{sid}, \mathcal{P}_i, \mathtt{el}_1, \mathtt{el}_\Gamma, \pi)$ where $\pi = (\mathtt{el}_\Gamma, \pi')$ to $\mathcal{F}_{\mathsf{VDF}}$.

Fig. 26: Part 2 of Simulator $\mathcal{S}$ for π_{VDF}.

Protocol $\pi_{\mathsf{VDF-RB}}$ is parameterized by an initial delay δ and is 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 $\mathcal{F}_{\mathsf{Ledger}}$ and $\mathcal{F}_{\mathsf{VDF}}$:

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

1. **Input Phase:** $\mathcal{P}_i$ proceeds as follows:
   (a) Sample $r_i \xleftarrow{\$} \{0,1\}^\tau$, send $(\mathsf{Submit}, \mathsf{sid}, r_i)$ to $\mathcal{F}_{\mathsf{Ledger}}$ and send $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$.
   (b) Wait for all $\mathcal{P}_j \in \mathcal{P}$ to broadcast their r_j by setting $\mathtt{cst} = 0, \mathtt{st}_0 = 1$ and performing the following steps every time it is activated ($\mathtt{st}_1$ is set to a constant because this VDF evaluation is only used to count the number of ticks until $1 + n/2$ r_j values are received): i. Send $(\mathsf{Read}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{Ledger}}$, receiving $(\mathsf{Read}, \mathsf{sid}, \mathsf{state}_i)$; ii. Check that $1 + n/2$ messages of the form $(\mathcal{P}_j, \mathsf{sid}, r_j)$ from different parties are in state_i (we call the set of such parties $\mathcal{C}$) and, if yes, proceed to **Output Phase**; iii. Send $(\mathsf{Fetch}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{VDF}}$, receiving $(\mathsf{Fetch}, \mathsf{sid}, L_i^1, L_i^2, L_i^3)$ in response. If $(\mathcal{P}_i, \mathtt{st}_{\mathtt{cst}}, \mathtt{st}') \in L_i^1$, increment $\mathtt{cst}$; iv. Send $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st}_{\mathtt{cst}})$ to $\mathcal{F}_{\mathsf{VDF}}$; v. If $\mathtt{cst} = \delta$, increment δ and go back to Step 1(a). vi. Send $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$;
2. **Output Phase:** $\mathcal{P}_i$ proceeds as follows:
   (a) Retrieve set $\{r_j\}_{\mathcal{P}_j \in \mathcal{C}}$ from$(\mathsf{Read}, \mathsf{sid}, \mathsf{state}_i)$ obtained in the last step from $\mathcal{F}_{\mathsf{Ledger}}$ and send $(\mathsf{Hash-Query}, \{r_j\}_{\mathcal{P}_j \in \mathcal{C}})$ to $\mathcal{G}_{\mathsf{rpoRO}}$, obtaining $(\textsc{Hash-Confirm}, in)$. Send $(\textsc{IsProgrammed}, \{r_j\}_{\mathcal{P}_j \in \mathcal{C}})$ to $\mathcal{G}_{\mathsf{rpoRO}}$, obtaining $(\textsc{IsProgrammed}, b)$. If $b = 1$, output $\perp$ and ignore the next steps.
   (b) Evaluate the $\mathcal{F}_{\mathsf{VDF}}$ on in by setting $\mathtt{cst} = 0, \mathtt{st}_0 = in$ and performing the following steps:
      i. Send $(\mathsf{Fetch}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{VDF}}$, receiving $(\mathsf{Fetch}, \mathsf{sid}, L_i^1, L_i^2, L_i^3)$ in response. If $(\mathcal{P}_i, \mathtt{st}_{\mathtt{cst}}, \mathtt{st}') \in L_i^1$, increment $\mathtt{cst}$.
      ii. If $\mathtt{cst} = \delta$, exit the loop and proceed to the next step.
      iii. Send $(\mathsf{Solve}, \mathsf{sid}, \mathtt{st}_{\mathtt{cst}})$ to $\mathcal{F}_{\mathsf{VDF}}$.
      iv. Send $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$.
   (c) Obtain the output of $\mathcal{F}_{\mathsf{VDF}}$ on $\mathtt{st}_1, \ldots, \mathtt{st}_\delta$ computed in the previous step by sending $(\mathsf{GetOutput}, \mathsf{sid}, \mathtt{st}_1, \ldots, \mathtt{st}_\delta)$ to $\mathcal{F}_{\mathsf{VDF}}$ and doing the following loop: i. Send $(\mathsf{Fetch}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{VDF}}$, receiving $(\mathsf{Fetch}, \mathsf{sid}, L_i^1, L_i^2, L_i^3)$ in response. If $(\mathcal{P}_i, (\mathtt{st}_1, out, \delta, \pi)) \in L_i^2$ then exit the loop; ii. Send $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$.
   (d) Save $(\mathtt{st}_1, out, \delta, \pi)$ for future public verification, output $(\textsc{Tossed}, \mathsf{sid}, out)$ and send $(\mathsf{Submit}, \mathsf{sid}, (\mathtt{st}_1, out, \delta, \pi))$ to $\mathcal{F}_{\mathsf{Ledger}}$ in order to allow verifiers to publicly verify the output at any point. Send $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$.

Verification: On input $(\textsc{Verify}, \mathsf{sid}, x)$, $\mathcal{V}_i$ proceeds as follows:

1. Send $(\mathsf{Read}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{Ledger}}$, receiving $(\mathsf{Read}, \mathsf{sid}, \mathsf{state}_i)$ and determining $\mathcal{C}$ for sid.
2. Send $(\mathsf{Hash-Query}, \{r_j\}_{\mathcal{P}_j \in \mathcal{C}})$ to $\mathcal{G}_{\mathsf{rpoRO}}$, obtaining $(\textsc{Hash-Confirm}, in)$. Send $(\textsc{IsProgrammed}, \{r_j\}_{\mathcal{P}_j \in \mathcal{C}})$ to $\mathcal{G}_{\mathsf{rpoRO}}$, obtaining $(\textsc{IsProgrammed}, b)$. If $b = 1$, output $(\textsc{Verify}, \mathsf{sid}, x, 0)$ and ignore the next steps.
3. Obtain $(\mathtt{st}_1, out, \delta, \pi)$ for sid from L and check that $\mathtt{st}_1 = in$. If not, output $(\textsc{Verify}, \mathsf{sid}, x, 0)$ and ignore the next steps.
4. Send $(\mathsf{Verify}, \mathsf{sid}, \mathtt{st}_1, out, \delta, \pi)$ to $\mathcal{F}_{\mathsf{VDF}}$ and do the following loop to obtain the verification result: (a) Send $(\mathsf{Fetch}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{VDF}}$, receiving $(\mathsf{Fetch}, \mathsf{sid}, L_i^1, L_i^2, L_i^3)$ in response. If $(\mathcal{V}_i, (\mathtt{st}_1, out, \delta, \pi, b)) \in L_i^3$ then exit the loop. (b) Send $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$.
5. If $b = 1$ then output $(\textsc{Verify}, \mathsf{sid}, x, 1)$. Otherwise output $(\textsc{Verify}, \mathsf{sid}, x, 0)$.

Fig. 27: Protocol $\pi_{\mathsf{VDF-RB}}$

E Randomness Beacon from VDFs

In Figure 27, we present $\pi_{\mathsf{VDF-RB}}$ which realizes $\mathcal{F}_{\mathsf{RB}}{}^{\Delta_{\mathsf{TLP-RB}}}$ from $\mathcal{F}_{\mathsf{VDF}}$, $\mathcal{F}_{\mathsf{Ledger}}$ and $\mathcal{G}_{\mathsf{rpoRO}}$. Protocol $\pi_{\mathsf{VDF-RB}}$ formalizes the folklore randomness beacon based on VDFs proposed in [9]. Even though the original protocol is not fully described (nor proven), we formalize the following informal construction using a semi-synchronous broadcast channel (where there is a finite but unknown delay):

1. All parties $\mathcal{P}_i \in_{\mathcal{P}}$ sample a random $r_i \xleftarrow{\$} \{0,1\}^\lambda$ and broadcast it.
2. Once $1 + n/2$ values $r_{j_1}, ..., r_{j_{1+n/2}}$ are received, every $\mathcal{P}_i$ computes $in = H(r_{j_1}, ..., r_{j_{1+n/2}})$ and computes a VDF with δ steps on input in.
3. Output whatever the VDF outputs.

As in the TLP based beacon, the main idea to prevent an adversary from biasing/aborting this protocol is to guarantee two conditions: 1. At least $1+n/2$ values r_j are received and at least 1 r_j is sampled uniformly at random by an honest party; 2. The adversary cannot compute an output of the VDF with δ steps before $1 + n/2$ values r_j are received, so it cannot choose its own value in a way that biases the output. While the first condition follows from honest majority, the second condition is guaranteed by dynamically adjusting the number of steps δ needed to compute the VDF *without prior knowledge of the maximum broadcast delay* Δ (as in our TLP based beacon). In order to do so, every party $\mathcal{P}_i$ checks that at least $1+n/2$ values r_j are received before δ ticks. If this is not the case, they increment δ and repeat the protocol from the beginning.

We design and analyse this protocol in the semi-synchronous model with an honest majority. However, as in the case of our TLP based beacon, in a synchronous scenario where the broadcast delay Δ is known, we could achieve security with a dishonest majority by proceeding to the **Opening Phase** after a delay of $\delta > \Delta$, since there would be a guarantee that all honest party values r_i have been received.

The security of Protocol $\pi_{\mathsf{VDF-RB}}$ is formally stated in Theorem 3, which we recall below for the sake of clarity.

Theorem 3 *If $\Delta = \mathsf{maxTXDelay} + \mathsf{emptyBlocks} \cdot \mathsf{slackWindow}$ (computed from $\mathcal{F}_{\mathsf{Ledger}}$'s parameters) is finite, Protocol $\pi_{\mathsf{VDF-RB}}$ UC-realizes $\mathcal{F}_{\mathsf{RB}}{}^{\Delta_{\mathsf{TLP-RB}}}$ in the $\mathcal{F}_{\mathsf{tlp}}, \mathcal{F}_{\mathsf{Ledger}}, \mathcal{G}_{\mathsf{rpoRO}}$-hybrid model with computational security against static adversaries corrupting $t < n/2$ parties in $\mathcal{P}$ for $\Delta_{\mathsf{TLP-RB}} = 2(\Delta+1)+\sum_{i=1}^{\Delta} i$. 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 $\pi_{\mathsf{VDF-RB}}$ by $\mathcal{A}$ composed with $\mathcal{F}_{\mathsf{tlp}}, \mathcal{F}_{\mathsf{Ledger}}, \mathcal{G}_{\mathsf{rpoRO}}$ from an ideal execution with $\mathcal{S}$ and $\mathcal{F}_{\mathsf{RB}}{}^{\Delta_{\mathsf{TLP-RB}}}$.*

Proof. We construct a simulator $\mathcal{S}$ that operates with an internal copy of the adversary $\mathcal{A}$, towards which it simulates an execution of $\pi_{\mathsf{VDF-RB}}$ as well as $\mathcal{F}_{\mathsf{VDF}}$, $\mathcal{G}_{\mathsf{rpoRO}}$ and $\mathcal{F}_{\mathsf{Ledger}}$. Essentially $\mathcal{S}$ executes the protocol $\pi_{\mathsf{VDF-RB}}$ exactly as an honest party would, forwarding all messages between $\mathcal{A}$ and $\mathcal{Z}$ up to the point where the protocol proceeds to the output phase. At this point $\mathcal{S}$ sends $(\textsc{Toss}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{RB}}{}^{\Delta_{\mathsf{TLP-RB}}}$ on behalf of the corrupted parties in $\mathcal{C}$ who sent inputs

r_j within the current delay δ, obtaining $(\textsc{Tossed}, \mathsf{sid}, x)$ from $\mathcal{F}_{\mathsf{RB}}{}^{\Delta_{\mathsf{TLP-RB}}}$. Next, $\mathcal{S}$ simulates the output of $\mathcal{F}_{\mathsf{VDF}}$ towards its internal copy of $\mathcal{A}$ as $(\mathtt{st}_1, x, \delta, \pi)$. There is no need to observe or program $\mathcal{G}_{\mathsf{rpoRO}}$, which serves only as an ideal hash function. Since neither the environment nor any other parties obtain the output x from $\mathcal{F}_{\mathsf{RB}}{}^{\Delta_{\mathsf{TLP-RB}}}$ before delay δ, $\mathcal{S}$ can obtain this output and program the simulated $\mathcal{F}_{\mathsf{VDF}}$ accordingly. Moreover, by following the instructions of an honest party in the input phase, $\mathcal{S}$ guarantees that the adversary cannot compute the VDF before $1 + n/2$ inputs are given (containing at least one honestly generated uniformly random r_i). Additionally, it is guaranteed that at a maximum delay $\delta = \Delta_{\mathsf{TLP-RB}} = 2(\Delta + 1) + \sum_{i=1}^{\Delta} i$ for $\Delta = \mathsf{maxTXDelay} + \mathsf{emptyBlocks} \cdot \mathsf{slackWindow}$ the simulator will proceed to the Output phase, since this accounts for iterating from initial delay $\delta = 1$ until $\delta = \mathsf{maxTXDelay} + \mathsf{emptyBlocks} \cdot \mathsf{slackWindow} + 1$, at which point all honest parties are guaranteed to agree on their (at least) $1 + n/2$ inputs. Since the protocol terminates and $\mathcal{A}$ cannot predict (nor bias) x, the simulation with $\mathcal{S}$ and $\mathcal{F}_{\mathsf{RB}}{}^{\Delta_{\mathsf{TLP-RB}}}$ as parameterized in the Theorem is indistinguishable from a real world execution of $\pi_{\mathsf{VDF-RB}}$ with $\mathcal{A}$. □

F Proofs for Theorems 4 and 5 for Composable Randomness Beacons

We present the proofs for Theorems 4 and 5 from Section 6 below. For the sake of clarity we repeat both theorems.

F.1 Proof for Theorem 4

Theorem 4 *If Δ is finite (though unknown) and all $\mathcal{P}_i \in \mathcal{P}$ receive inputs within a delay of Γ ticks of each other, Protocol $\pi_{\mathsf{TLP-RB}}$ UC-realizes $\mathcal{F}_{\mathsf{RB}}{}^{\Delta_{\mathsf{TLP-RB}}}$ in the $\mathcal{F}_{\mathsf{tlp}}, \mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$-hybrid model with computational security against static adversaries corrupting $t < \frac{n}{2}$ parties in $\mathcal{P}$ for $\Delta_{\mathsf{TLP-RB}} = 3(\Delta + 1) + \sum_{i=1}^{\Delta} i$. 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 $\pi_{\mathsf{TLP-RB}}$ by $\mathcal{A}$ composed with $\mathcal{F}_{\mathsf{tlp}}, \mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ from an ideal execution with $\mathcal{S}$ and $\mathcal{F}_{\mathsf{RB}}{}^{\Delta_{\mathsf{TLP-RB}}}$.*

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 $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ towards $\mathcal{A}$. For any environment $\mathcal{Z}$, we show that an execution with $\mathcal{S}$ and $\mathcal{F}_{\mathsf{RB}}{}^{\Delta_{\mathsf{TLP-RB}}}$ is indistinguishable from an execution with $\mathcal{A}$ in the $\mathcal{F}_{\mathsf{tlp}}, \mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$-hybrid model.

First of all, we observe that, since all honest parties receive inputs within a delay of Γ ticks and consequently start executing $\pi_{\mathsf{TLP-RB}}$, we are guaranteed that $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ never goes into a **TotalBreakdown**. Notice that, as soon as honest parties receive their inputs, they generate a TLP and broadcast it through $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$, which means that they all give an input to $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ within Γ ticks (one of the conditions for avoiding a **TotalBreakdown**). Moreover, by following the instructions of $\pi_{\mathsf{TLP-RB}}$, honest parties never provide inputs to the same instance of $\mathcal{F}^{\Gamma,\Delta}_{\mathsf{BC,delay}}$ (or rather an input with the same ssid) twice, which is the second condition for avoiding a $\mathcal{F}^{\Gamma,\Delta}_{\mathsf{BC,delay}}$. Hence, by following the instructions of an honest party in the simulated execution with $\mathcal{A}$, we ensure that $\mathcal{F}^{\Gamma,\Delta}_{\mathsf{BC,delay}}$ provides all of its guarantees (as is the case in a real world execution of $\pi_{\mathsf{TLP-RB}}$ with $\mathcal{A}$ where honest parties receive inputs within a delay of Γ ticks).

Next, we observe that, since Δ is finite and we have an honest majority, there exists a delay $\delta > \Delta$ such that the **Commitment Phase** will succeed and the parties will advance to the **Opening Phase**. Moreover, in the case of the smallest initial delay $\delta = 1$ we are guaranteed that the **Commitment Phase** will succeed after $\Delta_{\mathsf{TLP-RB}} = 3(\Delta + 1) + \sum_{i=1}^{\Delta} i$ ticks, which accounts for the time of iterating through all possible delays δ until arriving at $\delta = \Delta + 1$, at which point it is guaranteed that at all honest parties' TLPs will be received by all other honest parties (*i.e.* it is guaranteed that all honest parties proceed to the **Opening Phase**).

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 $(\mathcal{P}_j, \mathsf{sid}, \mathsf{puz}_j)$ 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 $t < n/2$ 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 $1+n/2$ TLPs must be received before proceeding and we are guaranteed that this happens because at least $1 + n/2$ parties are not corrupted and because we will eventually reach a δ that guarantees that all honest party messages are received (as argued above). 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 ${\mathcal{F}_{\mathsf{RB}}}^{\Delta_{\mathsf{TLP-RB}}}$. After the **Commitment Phase** is complete, $\mathcal{S}$ waits for $(\mathcal{P}_i, \mathsf{sid}, x)$ from ${\mathcal{F}_{\mathsf{RB}}}^{\Delta_{\mathsf{TLP-RB}}}$. $\mathcal{S}$ 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 puz_j in $(\mathcal{P}_j, \mathsf{sid}, \mathsf{puz}_j)$ broadcast by $\mathcal{P}_j$ is valid according to $\mathcal{F}_{\mathsf{tlp}}$ and extracts all r_j values from the valid puz_j, obtaining a set $\mathcal{G}$ 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 ${\mathcal{F}_{\mathsf{RB}}}^{\Delta_{\mathsf{TLP-RB}}}$ on behalf of each corrupted party $\mathcal{P}_j \in \mathcal{G}$ that broadcast a valid TLP.
- $\mathcal{S}$ equivocates the opening of puz_h from $\mathcal{P}_h$ in the **Opening Phase** so that it opens to a value r' such that $r' \oplus r_{\{j|\mathcal{P}_j \in \mathcal{G} \setminus \mathcal{P}_h\}} = x$.

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 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 $\mathcal{F}_{\mathsf{RB}}{}^{\Delta_{\mathsf{TLP-RB}}}$. This holds since only the valid TLPs puz_j sent before the first honest TLPs open are considered in computing the final output, the adversary does not learn the honest parties' values r_i before $\mathcal{S}$ does, and 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 $\mathcal{F}_{\mathsf{RB}}{}^{\Delta_{\mathsf{TLP-RB}}}$ for each of the corrupted parties that participated in the simulated execution correctly.

Hence, an execution with $\mathcal{S}$ and $\mathcal{F}_{\mathsf{RB}}{}^{\Delta_{\mathsf{TLP-RB}}}$ is indistinguishable from an execution with $\mathcal{A}$ in the $\mathcal{F}_{\mathsf{tlp}}, \mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$-hybrid model.

F.2 Proof for Theorem 5

Theorem 5 *If $\Delta = \mathsf{maxTXDelay} + \mathsf{emptyBlocks} \cdot \mathsf{slackWindow}$ (computed from $\mathcal{F}_{\mathsf{Ledger}}$'s parameters) is finite (though unknown), Protocol $\pi_{\mathsf{TLP-RB-LEDGER}}$ UC-realizes $\mathcal{F}_{\mathsf{RB}}{}^{\Delta_{\mathsf{TLP-RB}}}$ 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 $\mathcal{P}$ for $\Delta_{\mathsf{TLP-RB}} = 3(\Delta + 1) + \sum_{i=1}^{\Delta} i$. 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 $\mathcal{F}_{\mathsf{RB}}{}^{\Delta_{\mathsf{TLP-RB}}}$.*

The proof of this theorem follows from the proof of Theorem 4 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. As in the case of our TLP based beacon, in the case of the smallest initial delay $\delta = 1$ we are guaranteed that the **Commitment Phase** will succeed after $\Delta_{\mathsf{TLP-RB}} = 3(\Delta + 1) + \sum_{i=1}^{\Delta} i$ ticks for $\Delta = \mathsf{maxTXDelay} + \mathsf{emptyBlocks} \cdot \mathsf{slackWindow}$, which accounts for the time of iterating through all possible delays δ until arriving at $\delta = \Delta + 1$, at which point it is guaranteed that at all honest parties' TLPs will be agreed upon by all other honest parties (*i.e.* it is guaranteed that all honest parties proceed to the **Opening Phase**). 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 $\mathcal{S}$ as before 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 ${\mathcal{F}_{\mathsf{RB}}}^{\Delta_{\mathsf{TLP-RB}}}$ is indistinguishable from an execution of $\pi_{\mathsf{TLP-RB-LEDGER}}$ by $\mathcal{A}$ composed with $\mathcal{F}_{\mathsf{tlp}}, \mathcal{F}_{\mathsf{Ledger}}$.

G MPC with (Punishable) Output-Independent Abort, continued

In this Supplementary Material Section, we will first provide the full description of the functionality $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta}$ as well as a full proof of Theorem 6. This completes the description of $\pi_{\mathsf{mpc,oia}}$. Then, we show how to extend $\pi_{\mathsf{mpc,oia}}$ from Section 7 to financially punish cheaters. This will be done using a smart contract functionality $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ as well as a multi-party publicly verifiable delayed commitment $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta}$, both of which we will introduce here. The protocol then implements a modification of the previous functionality, which we call $\mathcal{F}_{\mathsf{mpc,poia}}^{\Delta,\gamma,\delta}$.

G.1 The Functionality $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta}$ for Commitments with Delayed Openings

In Figure Fig. 28 the functionality $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta}$ for commitments with verifiable delayed non-interactive openings is fully presented as it is used in $\pi_{\mathsf{mpc,oia}}$.

G.2 Proof of Theorem 6

We recall Theorem 6 below. 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 $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta} \circ \mathcal{S}$.

Theorem 6 *Let λ be the statistical security parameter and $\delta > \Delta$. Furthermore, assume that all honest parties obtain their inputs at most Γ ticks apart. Then the protocol $\pi_{\mathsf{mpc,oia}}$ GUC-securely implements the ticked functionality $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta}$ in the $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}, \mathcal{F}_{\mathsf{com}}^{\Delta,\delta}, \mathcal{F}_{\mathsf{ct}}^{\Delta}, \mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$-hybrid model against any static adversary corrupting up to $n-1$ parties in $\mathcal{P}$. The transcripts are statistically indistinguishable.*

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

Heartbeat: $\mathcal{S}$ will simulate the behavior of honest parties as in $\pi_{\mathsf{mpc,oia}}$ towards $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$, but $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ will never make a **Total Breakdown** due to behavior of the simulated honest parties (only when induced by $\mathcal{A}$). In case honest

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

parties would abort because dishonest parties did not send *beat* to $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$, then $\mathcal{S}$ makes $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta}$ abort as in the protocol. As in $\pi_{\mathsf{mpc,oia}}$, $\mathcal{S}$ will change to sending *ready* using $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ during **Reveal** and stop with this once it opened $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta}$

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

Input: The simulator behaves as during **Init**. For any honest party that provides an input into $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta}$ it inputs a dummy value into $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$. For every dishonest party $\mathcal{P}_i$ that provides an input to $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$ 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 $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta}$.

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

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

1. If an honest party $\mathcal{P}_i$ in $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta}$ sends Reveal then start broadcasting the *ready* message instead of the *beat* message for the simulated honest party on $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$. Conversely, if a dishonest party $\mathcal{P}_j$ starts broadcasting *ready* on $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ then send a Reveal-message in the name of $\mathcal{P}_j$ to $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta}$ once this broadcast round is finished.
2. Once $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ makes the first broadcast of only *ready* messages, then each simulated honest party $\mathcal{P}_i$ sends DOpen to its instance of $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta,i}$.
3. $\mathcal{S}$ now waits for δ ticks. 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 $J = J_1 \cup J_2$ 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 let us consider the **Heartbeat** mechanism. Here the difference between the real and the ideal world lies in the total breakdown of $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ due to behavior of honest parties. This will only occur in $\pi_{\mathsf{mpc,oia}}$ but not in $\mathcal{S}$, and it will happen if any honest party needs more than Γ ticks longer than the first honest party to submit a message for a respective c. If $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ does not break down, then all honest parties obtain the message *beat* in the same round and a total breakdown due to honest parties cannot happen for any future call to $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$, as all honest parties from now on act synchronized when sending messages to $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$. Therefore, this difference in behavior can only occur when $c = 0$. But by assumption, all honest parties obtain input within $\leq \Gamma$ ticks and they then immediately send *beat* to $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$. Therefore, honest parties never trigger a total breakdown in $\pi_{\mathsf{mpc,oia}}$ either and this is indistinguishable between protocol and simulation.

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 obtain 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_2 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. We need to argue that parties in the protocol agree on J and identify the same cheaters as in the simulation: First, all honest parties in the protocol start in the same "tick" round when opening their commitments. This is because, as argued before, they are synchronized in sending *beat* and are therefore also synchronized in sending *ready*. Therefore and because $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta}$ opens in a broadcast, they will always agree on the parties they identify as cheaters. 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. J_2 can by definition never contain an honest party. $\square$

G.3 Commitments with Publicly Verifiable Delayed Openings

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 $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ 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 $\mathsf{commits}[\mathsf{cid}] = (x, \mathsf{vt})$ 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. 29: Ticked Functionality $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta}$ For Multiparty Commitments with Verifiable Delayed Opening.

Functionality $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta}$ (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. 30: Ticked Functionality $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta}$ For Multiparty Commitments with Verifiable Delayed Opening.

In Fig. 29 and Fig. 30 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 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 which is part of the verification token makes it computationally infeasible for any $\mathcal{V}_i \in \mathcal{V}$ to simply brute-force the committed value in advance.

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, we use a bulletin board incorporated into the smart contract functionality $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ 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 7. *Protocol π_{vcom} GUC-realizes $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta}$ in the $\mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}, \mathcal{F}_{\mathsf{tlp}}$ 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 [11]. In our case $\mathcal{F}_{\mathsf{Auth}}$ is substituted by the authenticated bulletin board embedded in $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ through which messages are sent among parties. We can further extend the simulator $\mathcal{S}$ from [11] 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 $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta}$ 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 $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta}$ 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 [5]. 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 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. $\square$

G.4 The Smart Contract Functionality

The smart contract functionality $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ is depicted in Fig. 32 and Fig. 33. 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 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.

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 $\{0,1\}^\tau$), $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$, $\mathcal{F}_{\mathsf{tlp}}$ and $\mathcal{G}_{\mathsf{ticker}}$ as follows:

Commit: On input $(\mathsf{Commit}, \mathsf{sid}, \mathsf{cid}, x)$, $\mathcal{P}_{\mathsf{Send}}$ uniformly samples $r \xleftarrow{\$} \{0,1\}^\tau$ 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.
4. Send $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$.

$\mathcal{P}_{\mathsf{Send}}$ 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. 31: Protocol π_{vcom} for Multiparty Commitments with Verifiable Delayed Opening.

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 $\texttt{st}$ 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 $\texttt{st} = \mathsf{ready}$, 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 $(\mathsf{Fetch-BB}, \mathsf{sid})$ from a party in $\mathcal{P}$ or $\mathcal{V}$, output $(\mathsf{Return-BB}, \mathsf{sid}, \mathcal{B})$ to that party.

Deposit: On input $(\mathsf{Deposit}, \mathsf{sid}, \texttt{coins}(d))$ 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 $\texttt{st} \in \{\mathsf{ready}, \mathsf{dep}(x)\}$:

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

Fig. 32: Ticked Functionality $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ for Smart Contracts.

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.

G.5 MPC 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. 34 and Fig. 35.

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

Functionality $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ (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 $e_1, \dots, e_n \leftarrow d$.
        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. 33: Ticked Functionality $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ for Smart Contracts.

sharing is finished, deposit coins to $\mathcal{F}_{\mathsf{mpc,poia}}^{\Delta,\gamma,\delta}$ 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 $\mathcal{F}_{\mathsf{mpc,poia}}^{\Delta,\gamma,\delta}$, otherwise it switches to a *waiting state* $\mathtt{wait}$.

Similar to $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta}$, the functionality $\mathcal{F}_{\mathsf{mpc,poia}}^{\Delta,\gamma,\delta}$ 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 $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta}$. In addition, after these $\delta+\gamma$ ticks the functionality will either reimburse all parties with $\mathtt{coins}(d)$ 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 $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$.

The Protocol. The full protocol $\pi_{\mathsf{mpc,poia}}$ is depicted in Fig. 36 and Fig. 37. 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 $\pi_{\mathsf{mpc,oia}}$ we do not have to externally synchronize the honest parties using a

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 $\mathtt{st}$ 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 $\mathtt{st} = \perp$:

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 $\mathcal{P}_j \in (\mathcal{P} \cup \mathcal{V}) \setminus I$ 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 $(\mathsf{Deposit}, \mathsf{sid}, \mathtt{coins}(d))$ by $\mathcal{P}_i \in \mathcal{P}$, if **Share** finished, if no $\mathsf{DeliverShare}$ message is in $\mathcal{Q}$ and if $\mathtt{st} \in \{\mathtt{dep}, \perp\}$:

1. Simultaneously send a message (i, "$\mathtt{coins}(d)$") to the parties $\mathcal{P} \setminus \{\mathcal{P}_i\}$ via $\mathcal{Q}$ with delay 0.
2. If $\mathtt{st} = \perp$ then set $\mathtt{st} = \mathtt{dep}(\gamma)$.
3. Notify $\mathcal{S}$ about the message.

Fig. 34: Ticked Functionality $\mathcal{F}_{\mathsf{mpc,poia}}^{\Delta,\gamma,\delta}$ for Secure Multiparty Computation with Punishable Output-Independent Abort.

heartbeat, as the $\mathsf{AllDeposited}$ message from $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ now serves as synchronization point within the protocol.

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

```
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 $\mathtt{st} = \mathtt{dep}(x)$:
     (a) Set $\mathtt{st} = \mathtt{dep}(x - 1)$.
     (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 $\mathsf{sid}$ except to Fetch Message.
```

Fig. 35: Ticked Functionality $\mathcal{F}^{\Delta,\gamma,\delta}_{\mathsf{mpc,poia}}$ for Secure Multiparty Computation with Punishable Output-Independent Abort.

solve the solutions within the time-frame given by $\mathcal{F}^{\gamma,\delta}_{\mathsf{SC}}$, so the set J_2 identified by $\mathcal{F}^{\gamma,\delta}_{\mathsf{SC}}$ 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 8. *Let λ be the statistical security parameter and $\delta > \gamma$. Then the protocol $\pi_{\mathsf{mpc,poia}}$ GUC-securely implements the ticked functionality $\mathcal{F}^{\Delta,\gamma,\delta}_{\mathsf{mpc,poia}}$ in the $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}, \mathcal{F}^{\gamma,\delta}_{\mathsf{vcom}}, \mathcal{F}^{\Delta}_{\mathsf{ct}}, \mathcal{F}^{\gamma,\delta}_{\mathsf{SC}}$-hybrid model against any static adversary corrupting up to $n-1$ of the n parties in $\mathcal{P}$. The transcripts are statistically indistinguishable.*

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

All parties $\mathcal{P}$ have access to one instance of the functionalities $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}, \mathcal{F}^{\Delta}_{\mathsf{ct}}$ and $\mathcal{F}^{\gamma,\delta}_{\mathsf{SC}}$. Furthermore, each $\mathcal{P}_i \in \mathcal{P}$ has it's own $\mathcal{F}^{\gamma,\delta,i}_{\mathsf{vcom}}$ where it acts as the dedicated sender and all other parties of $\mathcal{P}$ are receivers.
Throughout the protocol, we say "$\mathcal{P}_i$ ticks" when we mean that it sends $(\mathsf{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$. We say that "$\mathcal{P}_i$ waits" when we mean that $\mathcal{P}_i$, upon each activation, first checks if the event happened and if not, sends $(\mathsf{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$.

Init:

1. Each $\mathcal{P}_i$ sends $(\mathsf{Register}, \mathsf{sid}, \{\mathcal{F}^{\gamma,\delta,j}_{\mathsf{vcom}}\}_{j\in[n]})$ to $\mathcal{F}^{\gamma,\delta}_{\mathsf{SC}}$ and ticks. Then it waits until it receives $\mathsf{Register}$ from $\mathcal{F}^{\gamma,\delta}_{\mathsf{SC}}$ for each $\mathcal{P} \setminus \{\mathcal{P}_i\}$.
2. Each $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{Init}, \mathsf{sid}, C)$ to $\mathcal{F}^{\Delta}_{\mathsf{mpc,sso}}$ and ticks. Then it 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 ticks. Then it 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 ticks. Then it 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 ticks. Then it waits 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 ticks. Then it waits 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 $\mathcal{F}^{\gamma,\delta,i}_{\mathsf{vcom}}$ and ticks. Then it waits for messages $(\mathsf{Commit}, \mathsf{sid}, \mathsf{cid}_j)$ from the $\mathcal{F}^{\gamma,\delta,j}_{\mathsf{vcom}}$-instances of all other parties $\mathcal{P}_j \in \mathcal{P} \setminus \{\mathcal{P}_i\}$.
5. Each $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{Toss}, \mathsf{sid}, m \cdot \lambda)$ to $\mathcal{F}^{\Delta}_{\mathsf{ct}}$ and ticks. 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 ticks. It then 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 $j \in [n]$.
8. Each $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{Broadcast}, \mathsf{sid}, (\mathbf{A}, \boldsymbol{t}_1, \ldots, \boldsymbol{t}_n))$ to $\mathcal{F}^{\gamma,\delta}_{\mathsf{SC}}$ and ticks.
9. Each $\mathcal{P}_i \in \mathcal{P}$ waits until it received n identical broadcasts $(\mathsf{Broadcast}, \mathsf{sid}, (\mathbf{A}, \boldsymbol{t}_1, \ldots, \boldsymbol{t}_n))$, one from each $\mathcal{P}_j \in \mathcal{P}$.

Fig. 36: Protocol $\pi_{\mathsf{mpc,poia}}$ for MPC with Punishable Output-Independent Abort.

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 $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$ and ticks.
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 $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta,i}$ and ticks. 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 $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta}$, ticks 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. 37: Protocol $\pi_{\mathsf{mpc,poia}}$ for MPC with Punishable Output-Independent Abort.