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

Carsten Baum[1,2] *, Bernardo David[3] **, Rafael Dowsley[4] ***,
Ravi Kishore[3] †, Jesper Buus Nielsen[2] ‡, and Sabine Oechsner[5] §

[1] Technical University of Denmark, Denmark
cabau@dtu.dk
[2] Aarhus University, Denmark
{cbaum,jbn}@cs.au.dk
[3] IT University of Copenhagen, Denmark
{beda,rava}@itu.dk
[4] Monash University, Australia
rafael@dowsley.net
[5] University of Edinburgh, United Kingdom
s.oechsner@ed.ac.uk

Abstract. Recently, time-based primitives such as time-lock puzzles (TLPs) and verifiable delay functions (VDFs) 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.

* 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 the Independent Research Fund Denmark grants number 9040-00399B (TrA^{2}C), 9131-00075B (PUMA) and 0165-00079B (P2DP).

*** 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.

† Supported by the Independent Research Fund Denmark grant number 9131-00075B (PUMA).

‡ 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 Input Output (iohk.io) through their funding of the Edinburgh Blockchain Technology Lab. Partially done while Sabine Oechsner was with Aarhus University and supported by the Danish Independent Research Council under Grant-ID DFF-8021-00366B (BETHE) and Concordium Foundation.

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 (PV-TLPs). We use our new UC VDF to prove a folklore result on VDF-based randomness beacons used in industry and build an improved randomness beacon from our new UC PV-TLPs. We moreover construct the first multiparty computation protocol with punishable output-independent aborts (POIA-MPC), *i.e.* MPC with OIA and financial punishment for cheating. Our novel POIA-MPC both establishes the feasibility of (non-punishable) OIA-MPC and significantly improves on the efficiency of state-of-the-art OIA-2PC and (non-OIA) MPC with punishable aborts.

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) [39] and Verifiable Delay Functions (VDFs) [12]. 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), after which it generates both an output and 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 [11,38,41], obtaining new properties [25] and achieving stronger security guarantees [22,31,26]. These works are motivated by the many applications of TLPs and VDFs, such as randomness beacons [12,13], partially fair secure computation [20] and auctions [13]. 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 [39,13,11,31,26] and all known constructions of VDFs [12,38,41,22,25] do not offer general composability guarantees, meaning it is not possible to easily and securely use those in more complex protocols.

The current default tool for proving security of cryptographic constructions under general composability is the Universal Composability (UC) framework [14]. However, the UC framework is inherently asynchronous and does not capture time, meaning that a notion of passing time has to be added in order to analyze time-based constructions in UC. Recently, TARDIS [6] introduced a suitable time model and the first UC construction of TLPs, proven secure under the iterated squaring assumption of [39] using a programmable random oracle. [6] also shows 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, TARDIS [6] showed that UC TLPs 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 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 TLP applications (*e.g.* auctions [13]) are intrinsically multiparty and VDFs are used in practice for building randomness beacons [12,40]. The TARDIS TLP formalization and its applications also give adversaries exactly as much power in breaking the time-based assumption as the honest parties, which appears very restrictive and unrealistic.

1.1 Our Contributions

In this work, we present the first UC-secure constructions of VDFs and introduce the related notion of Publicly Verifiable TLPs, which we also construct. Using these primitives as building blocks, we construct a new more efficient randomness beacon and Multiparty Computation with Output-Independent Abort (OIA-MPC) and Punishable Abort. Our constructions are both practical and proven secure under general composition, and support adversaries who can break the timing assumptions faster than honest parties.

UC Verifiable Delay Functions. We introduce the *first* UC definition of VDFs [12], which is a delicate task and a contribution on its own. We also present a matching construction that consists in compiling a trapdoor VDF [41] into a UC-secure VDF in the random oracle model while only increasing the proof size by a small constant. Even though we manage to construct a very simple and efficient compiler, the security proof for this construction is highly detailed and complex. Based on our UC VDF, we give the *first* security proof of a folklore randomness beacon construction [12].

UC Publicly Verifiable Time-Lock Puzzles (PV-TLP). We introduce publicly verifiable TLPs (PV-TLP), presenting an ideal functionality and a UC-secure construction for this primitive. A party who solves a PV-TLP (or its creator) can prove to any third party that a certain message was contained in the PV-TLP (or that it was invalid) in way that verifying the proof takes constant time. We show that the TLP of [6] allows for proving that a message was contained in a valid TLP. Next, we introduce a new UC-secure PV-TLP scheme based on trapdoor VDFs that allows for a solver to prove that a puzzle is invalid, similarly to the construction of [26], which does not achieve UC security.

Efficient UC Randomness Beacon from PV-TLP. Building on our new notion (and construction) of PV-TLPs, we introduce a new provably secure randomness beacon protocol. Our construction achieves far better best case scenario

efficiency than the folklore VDF-based construction [12]. Our novel PV-TLP-based construction *requires only $O(n)$ broadcasts (as does [12]) to generate a uniformly random output*, where n is the number of parties. Differently from the VDF-based construction [12], whose execution time is *at least* the worst case communication channel delay, our protocol 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*. This construction and its proof require not only a simple application of UC PV-TLPs but also a careful analysis of the relative delays between PV-TLPs broadcast channels/public ledgers and PV-TLPs. We not only present this new protocol but also provide a full security proof in the partially synchronous model (where the communication delay is unknown), characterizing the protocol's worst case execution time in terms of the communication delay upper bound. In comparison, no security proof for the construction of [12] is presented in their work.

UC Multiparty Computation (MPC) with Output Independent Abort (OIA-MPC). We construct the first UC-secure protocol for Multiparty Computation with Output Independent Abort (OIA-MPC), which is a stronger notion of MPC where aborts by cheaters must be made before they know the output. This notion is a generalization of the limited OIA-2PC result from [6]. As our central challenge, we identify the necessity of synchronizing honest parties so that their views allow them to agree on the same set of cheaters. We design a protocol that only requires that honest parties are not too much out of sync when the protocol starts and carefully analyze its security.

UC MPC with Punishable Output Independent Abort (POIA-MPC) from PV-TLP. We construct the first protocol for Multiparty Computation with Punishable Output Independent Abort (POIA-MPC), generalizing OIA-MPC to a setting where i) outputs can be publicly verified; and ii) cheaters in the output stage can be identified and financially punished. Our construction employs our new publicly verifiable TLPs to construct a commitment scheme with delayed opening. To use this simple commitment scheme, we improve the currently best [4] techniques for publicly verifiable MPC with cheater identification in the output stage. We achieve this by eliminating the need for homomorphic commitments, which makes our construction highly efficient. We do not punish cheating that occurs before the output phase (i.e. before the output can be known), as this requires expensive MPC with publicly verifiable identifiable abort [30,34,8]. Our approach is also taken in other previous works [1,10,35,4].

1.2 Related Work

The recent work of Baum et al. [6] introduced the first construction of a composable TLP, while previous constructions such as [39,13,11] were only proven to be stand-alone secure. As an intermediate step towards composable TLPs, non-malleable TLPs were constructed in [31,26]. The related notion of VDFs has been investigated in [12,38,41,22,25]. Also for these constructions, composability

guarantees have so far not been shown. Hence, issues arise when using these primitives as building blocks in more complex protocols, since 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) [33,16] and on VDFs [12], although neither of these constructions is composable. The best UC-secure randomness beacons based on PVSS [17] still require $O(n^2)$ communication where n is the number of parties even if only one single value is needed. UC-secure randomness beacons based on verifiable random functions [21,2] can 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 [18], which includes the 2-party setting. Couteau et al. [20] 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. [6] 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.* [27], 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 [28]). 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. Being based on the TARDIS model of [6], our work uses the standard UC framework without such stringent (and arguably unrealistic) modifications/restrictions.

An alternative, recently popularized idea is to circumvent the impossibility result of [18] by imposing financial penalties. In this model, cheating behavior is punished using cryptocurrencies and smart contracts, which incentivizes rational adversaries to act honestly. Works that achieve fair output delivery with penalties such as [1,10,35,4] allow the adversary to make the abort decision *after* he sees the output. Therefore financial incentives must be chosen according to the adversary's worst-case gain. Our POIA-MPC construction forces the adversary to decide before seeing the output and incentives can be based on the expected gain of cheating in the computation instead. All these mentioned works as well as ours focus on penalizing cheating during the output phase only, as current MPC protocols with publicly verifiable cheater identification are costly [7,34,8].

2 Preliminaries

We use the (Global) Universal Composability or (G)UC model [14,15] 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*. We denote with λ the statistical and τ the computational security parameter.

Public Verifiability in UC. We model the public verification of protocol 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.

2.1 The TARDIS [6] Composable Time Model

The TARDIS [6] 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 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 can observe 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 [6] for more details.

How we use the TARDIS [6] model. To control the observable side effects of ticks, the protocols and ideal functionalities presented in this work are restricted to interact in the[6] "pull model". This precludes functionalities from

[6] The pull model, a standard approach in networking, has been used in previous works before such as [32].

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

Initialize a set of registered parties $\mathtt{Pa} = \emptyset$, a set of registered functionalities $\mathtt{Fu} = \emptyset$, a set of activated parties $L_{\mathtt{Pa}} = \emptyset$, and a set of functionalities $L_{\mathtt{Fu}} = \emptyset$ that have been informed about the current tick.

Party registration: Upon receiving $(\mathtt{register}, \mathsf{pid})$ from honest party $\mathcal{P}$ with pid pid, add pid to $\mathtt{Pa}$ and send $(\mathtt{registered})$ to $\mathcal{P}$.

Functionality registration: Upon receiving $(\mathtt{register})$ from functionality $\mathcal{F}$, add $\mathcal{F}$ to $\mathtt{Fu}$ and send $(\mathtt{registered})$ to $\mathcal{F}$.

Tick: Upon receiving $(\mathtt{tick})$ from the environment, do the following:

1. If $\mathtt{Pa} = L_{\mathtt{Pa}}$, reset $L_{\mathtt{Pa}} = \emptyset$ and $L_{\mathtt{Fu}} = \emptyset$, and send $(\mathtt{ticked})$ to the adversary $\mathcal{S}$.
2. Else, send $(\mathtt{notticked})$ to the environment.

Ticked request: Upon receiving $(\mathtt{ticked}?)$ from functionality $\mathcal{F} \in \mathtt{Fu}$: If $\mathcal{F} \notin L_{\mathtt{Fu}}$, add $\mathcal{F}$ to $L_{\mathtt{Fu}}$ and send $(\mathtt{ticked})$ to $\mathcal{F}$. Otherwise send $(\mathtt{notticked})$ to $\mathcal{F}$.

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

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

implicitly providing communication channels between parties. Parties 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, a 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 [6], after every tick, each ticked functionality $\mathcal{F}$ that we define (unless mentioned otherwise) allows $\mathcal{A}$ 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 [6] 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.* [36,32]) 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 its 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 interface works identically for all ticked functionalities as follows:

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 [6] is that the functionality $\mathcal{F}$, upon receiving a request $(\mathsf{Request}, \mathsf{sid}, m)$ by party $\mathcal{P}_i$ must first internally generate unique message IDs mid to balance message delivery with the adversarial option to delay messages. $\mathcal{F}$ then internally stores the message to be delivered together with the mid in $\mathcal{Q}$ 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 (Request, $\mathsf{sid}, i, \mathsf{mid}_{i_1}, \ldots, \mathsf{mid}_{i_k}$) to $\mathcal{S}$." Finally, the **Macro** "*Send* m *with prefix* Request *and the IDs to* $\mathcal{S}$" expands to "Send (Request, $\mathsf{sid}, i, m, \mathsf{mid}_{i_1}, \ldots, \mathsf{mid}_{i_k}$) to $\mathcal{S}$."

If honest parties send messages via simultaneous broadcast (ensuring simultaneous arrival), then we will 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.

2.2 Trapdoor Verifiable Sequential Computation

Functionality $\mathcal{F}_{\mathsf{psc}}$ is presented in Figure 2 and captures the notion of a generic stand alone trapdoor verifiable sequential computation scheme (a generalization of a trapdoor VDF) in a similar way as the iterated squaring assumption from [39] is captured in [6]. More concretely, $\mathcal{F}_{\mathsf{psc}}$ allows the evaluation of Γ computational steps taking as input an initial state $\mathtt{el}$ and outputting a final state $\mathtt{el}_\Gamma$ along with a proof π. A verifier can use π to check that a state $\mathtt{el}'_\Gamma$ was indeed obtained after Γ computational steps starting from $\mathtt{el}$. Each computational step takes a tick to happen, and parties who are currently performing a computation must activate $\mathcal{F}_{\mathsf{psc}}$ in order for their computation to advance when the next tick happens. The proof π' can be verified with respect to $\mathtt{el}, \mathtt{el}_\Gamma, \Gamma$ in time essentially independent of Γ. Since current techniques (*e.g.* [38,41,25]) for verifying such a proof require non-constant computational time, we model the number of ticks necessary for each by function $g(\Gamma)$. The implementation of $\mathcal{F}_{\mathsf{psc}}$ is presented in the full version [5] due to space limitations.

$\mathcal{F}_{\mathsf{psc}}$ must be used to capture a stand alone verifiable sequential computation because, as observed in [6], exposing the actual states from a concrete computational problem would allow the environment to perform several computational steps without activating other parties (and essentially breaking the hardness assumption). However, notice that $\mathcal{F}_{\mathsf{psc}}$ 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 is 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 transition between states $\mathtt{el}$ and $\mathtt{nxt}$ is injective.

2.3 Multi-Party Message Delivery

Ticked Authenticated Broadcast In Fig. 3 we describe a ticked functionality $\mathcal{F}^{\Gamma,\Delta}_{\mathsf{BC,delay}}$ for delayed authenticated simultaneous broadcast. $\mathcal{F}^{\Gamma,\Delta}_{\mathsf{BC,delay}}$ 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, 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. If this loose start synchronization condition is not fulfilled, the

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

$\mathcal{F}_{\mathsf{psc}}$ interacts with a set of parties $\mathcal{P} = \{\mathcal{P}_i, \ldots, \mathcal{P}_n\}$, an owner $\mathcal{P}_o \in \mathcal{P}$ (if $\mathcal{P}_o = \perp$, no $\mathcal{P}_i \in \mathcal{P}$ can access **Trapdoor Solve**) and an adversary $\mathcal{S}$. It is parameterized by an adversarial slack parameter $0 \leq \epsilon \leq 1$, state space $\mathcal{ST}$, a proof space $\mathcal{PROOF}$ and a function $g : \{0,1\}^\star \mapsto \mathbb{N}$ determining the number of ticks for verifying proofs. $\mathcal{F}_{\mathsf{psc}}$ has initially empty lists prf, L and $\mathcal{Q}_{\mathsf{v}}$ (proofs being verified); and flags f_i for $i = 1, \ldots, n$ that are initially set to zero.

Trapdoor Solve: Upon receiving $(\mathsf{TdSolve}, \mathsf{sid}, \mathtt{el}_0, \Gamma)$ from $\mathcal{P}_o$ where $\Gamma \in \mathbb{N}^+$ and $\mathtt{el}_0 \in \mathcal{ST}$, sample Γ random distinct states $\mathtt{el}_j \xleftarrow{\$} \mathcal{ST}$ for $j \in \{1, \ldots, \Gamma\}$ and add $(\mathtt{el}_{j-1}, \mathtt{el}_j)$ to steps. Also sample proof $\pi \xleftarrow{\$} \mathcal{PROOF}$. Add $(\mathtt{el}_0, \Gamma, \mathtt{el}_\Gamma, \pi)$ to prf and output $(\mathsf{sid}, \mathtt{el}_0, \Gamma, \mathtt{el}_\Gamma, \pi)$ to $\mathcal{P}_o$.

Solve: Upon receiving $(\mathsf{Solve}, \mathsf{sid}, \mathtt{el}_0, \Gamma)$ from $\mathcal{P}_i \in \mathcal{P}$ where $\mathtt{el}_0 \in \mathcal{ST}$, append $(\mathcal{P}_i, \mathsf{sid}, \mathtt{el}_0, \Gamma, \mathtt{el}_0, 0)$ to L and send $(\mathsf{Solve}, \mathsf{sid}, \mathtt{el}_0, \Gamma)$ to $\mathcal{S}$.

Advance State: Upon receiving $(\mathsf{AdvanceState}, \mathsf{sid})$ from $\mathcal{P}_i \in \mathcal{P}$, set $f_i = 1$.

Tick:

- For each $(\mathcal{P}_i, \mathsf{sid}, \mathtt{el}_0, \Gamma, \mathtt{el}_c, c) \in L$, if $f_i = 1$ proceed as follows:
  1. If there is no $\mathtt{el}_{c+1}$ such that $(\mathtt{el}_c, \mathtt{el}_{c+1}) \in \mathsf{steps}$ then sample $\mathtt{el}_{c+1} \xleftarrow{\$} \mathcal{ST}$, and append $(\mathtt{el}_c, \mathtt{el}_{c+1})$ to steps.
  2. Output $(\mathtt{el}_c, \mathtt{el}_{c+1})$ to $\mathcal{S}$ and update $(\mathcal{P}_i, \mathsf{sid}, \mathtt{el}_0, \Gamma, \mathtt{el}_c, c)$ by setting $c = c + 1$.
  3. If $c \geq \epsilon\Gamma$ and $(\mathtt{el}_0, \Gamma, \mathtt{el}_\Gamma, \pi) \in \mathsf{prf}$, output $(\mathsf{GetEsPf}, \mathtt{el}_0, \Gamma, \mathtt{el}_c, \mathtt{el}_{c+1}, \ldots, \mathtt{el}_\Gamma, \pi)$ to $\mathcal{S}$.
  4. Else If $c \geq \epsilon\Gamma$ but $(\mathtt{el}_0, \Gamma, \mathtt{el}_\Gamma, \pi) \notin \mathsf{prf}$, then for $j \in \{c+1, \ldots, \Gamma\}$ sample state $\mathtt{el}_j \xleftarrow{\$} \mathcal{ST}$ and add $(\mathtt{el}_{j-1}, \mathtt{el}_j)$ to steps. Also sample proof $\pi \xleftarrow{\$} \mathcal{PROOF}$, and add $(\mathtt{el}_0, \Gamma, \mathtt{el}_\Gamma, \pi)$ to prf. Finally, output $(\mathsf{GetEsPf}, \mathtt{el}_0, \Gamma, \mathtt{el}_c, \mathtt{el}_{c+1}, \ldots, \mathtt{el}_\Gamma, \pi)$ to $\mathcal{S}$.
  5. If $c = \Gamma$, output $(\mathsf{GetPf}, \mathsf{sid}, \mathtt{el}_0, \Gamma, \mathtt{el}_\Gamma, \pi)$ to $\mathcal{P}_i$, and remove $(\mathcal{P}_i, \mathsf{sid}, \mathtt{el}_0, \Gamma, \mathtt{el}_\Gamma, \Gamma)$ from L.
- For each $(\mathcal{P}_i, \mathsf{sid}, c, \mathtt{el}_I, \Gamma, \mathtt{el}_O, \pi) \in \mathcal{Q}_{\mathsf{v}}$, if $f_i = 1$ proceed as follows:
  1. If $c = 0$: remove $(\mathcal{P}_i, \mathsf{sid}, 0, \mathtt{el}_I, \Gamma, \mathtt{el}_O, \pi)$ from $\mathcal{Q}_{\mathsf{v}}$ and set $b = 1$ if $(\mathtt{el}_I, \Gamma, \mathtt{el}_O, \pi) \in \mathsf{prf}$, otherwise set $b = 0$, and output $(\mathsf{Verified}, \mathsf{sid}, \mathtt{el}_I, \Gamma, \mathtt{el}_O, \pi, b)$ to $\mathcal{P}_i$.
  2. Else, if $c > 0$: update $(\mathcal{P}_i, \mathsf{sid}, c, \mathtt{el}_I, \Gamma, \mathtt{el}_O, \pi)$ by setting $c = c - 1$.

Set flag $f_i = 0$ for $i = 1, \ldots, n$.

Verify: Upon receiving $(\mathsf{Verify}, \mathsf{sid}, \mathtt{el}_I, \Gamma, \mathtt{el}_O, \pi)$ from $\mathcal{P}_i \in \mathcal{P}$ where $\pi \in \mathcal{PROOF}$, add $(\mathcal{P}_i, \mathsf{sid}, g(\Gamma), \mathtt{el}_I, \Gamma, \mathtt{el}_O, \pi)$ to $\mathcal{Q}_{\mathsf{v}}$.

Fig. 2: Ticked Functionality $\mathcal{F}_{\mathsf{psc}}$ for trapdoor provable sequential computations.

functionality no longer provides any guarantees, allowing the adversary to freely manipulate message delivery (specified in **Total Breakdown**).

In comparison to the two-party secure channel functionality $\mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$ of [6], our broadcast functionality $\mathcal{F}^{\Gamma,\Delta}_{\mathsf{BC,delay}}$ uses a scheduling-based approach and ex-

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

The ticked functionality $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ is parameterized by maximal input desynchronization Γ, 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. $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ for each ssid has internal states $\mathtt{st}_{\mathsf{ssid}}, \mathsf{done}_{\mathsf{ssid}}$ that are initially $\perp$.

Init: In the beginning of the execution, $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ waits for input (Delay, Δ) from $\mathcal{S}$. Upon receiving (Delay, Δ) from $\mathcal{S}$ where $\Delta \in \mathbb{N}$ and $\Delta \geq \Gamma$, $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ proceeds to the next steps using Δ as its internal (unknown to honest parties) delay parameter.

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

plicitly 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 [6].

We briefly discuss how to implement $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Gamma,\Delta}$. We could start from a synchronous broadcast protocol like [24] or the one in [23] 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 [19] 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 [37]. 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 [23], 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.

We stress that other broadcast channels than the one in $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ may also be modeled using [6], although these may not be applicable to instantiate OIA-MPC as we do in Section 6.

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 the full version [5]. 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. We also enforce liveness and chain quality properties, and our ledger functionality can be realized by the same protocols as [3].

3 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 PV- TLP to later convince a verifier that the PV-TLP contained a certain message or that it was invalid. The verifier only needs constant time to verify this claim. The ideal functionality $\mathcal{F}_{\mathsf{TLP}}$ as presented in Figures 4 & 5 models exactly that behavior: $\mathcal{F}_{\mathsf{TLP}}$ has an extra interface for any verifier to check whether a certain solution to a given PV-TLP is correct. Moreover, $\mathcal{F}_{\mathsf{TLP}}$ allows the adversary to obtain the message from a PV-TLP with Γ steps in just $\epsilon\Gamma$ steps for $0 < \epsilon \leq 1$, modeling the slack between concrete computational complexities for honest parties and for the adversary is sequential computation assumptions.

Functionality $\mathcal{F}_{\mathsf{TLP}}$ allows the owner to create a new TLP containing message m to be solved in Γ steps by activating it with $(\mathsf{CreatePuzzle}, \mathsf{sid}, \Gamma, m)$. Other parties can request the solution of a TLP puz generated by the owner of $\mathcal{F}_{\mathsf{TLP}}$ by activating it with message $(\mathsf{Solve}, \mathsf{sid}, \mathsf{puz})$. After every tick when a party activates $\mathcal{F}_{\mathsf{TLP}}$ with message $(\mathsf{AdvanceState}, \mathsf{sid})$, one step of this party's previosly requested puzzle solutions is evaluated. When $\epsilon\Gamma$ steps have been computed, $\mathcal{F}_{\mathsf{TLP}}$ leaks message m contained in the puzzle puz to the adversary $\mathcal{S}$. When all Γ steps of a puzzle solution requested by a party are evaluated, $\mathcal{F}_{\mathsf{TLP}}$ outputs

m and a proof π that m was indeed contained in puz to that party. Finally, a party who has a proof π that a message m was contained in puz can verify this proof by activating $\mathcal{F}_{\mathsf{TLP}}$ with message $(\mathsf{Verify}, \mathsf{sid}, \mathsf{puz}, m, \pi)$.

Functionality $\mathcal{F}_{\mathsf{TLP}}$ (Part 1)

$\mathcal{F}_{\mathsf{TLP}}$ is parameterized by a computational security parameter τ, a message space $\{0,1\}^\tau$, a state space $\mathcal{ST}$, a tag space $\mathcal{TAG}$, a proof space $\mathcal{PROOF}$, a slack parameter $0 < \epsilon \leq 1$ and a function $g : \{0,1\}^\star \mapsto \mathbb{N}$ (determining how many ticks it takes to verify a proof). $\mathcal{F}_{\mathsf{TLP}}$ interacts with a set of parties $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, an owner $\mathcal{P}_o \in \mathcal{P}$ and an adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{TLP}}$ maintains flags f_i for $i = 1, \ldots, n$ that are initially set to 0 and initially empty lists steps (honest state transitions), omsg (output messages and proofs), $\mathcal{Q}_{\mathsf{v}}$ (proofs being verified), L (puzzles being solved).

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 $\mathtt{tag} \xleftarrow{\$} \mathcal{TAG}$, $\mathtt{st}_0 \xleftarrow{\$} \mathcal{ST}$, and proof $\pi \xleftarrow{\$} \mathcal{PROOF}$. If $\mathcal{P}_o$ is corrupted, let $\mathcal{S}$ provide values $\mathtt{tag}$, $\mathtt{st}_0$, and the proof π.
2. If $(\mathtt{tag}, \mathtt{st}_0, \pi) \notin \mathcal{TAG} \times \mathcal{ST} \times \mathcal{PROOF}$ or there exists $(\mathtt{st}'_0, \Gamma', \mathtt{tag}', m', \pi) \in \mathsf{omsg}$ then $\mathcal{F}_{\mathsf{TLP}}$ halts. Otherwise, append $(\mathsf{puz} = (\mathtt{st}_0, \Gamma, \mathtt{tag}), m, \pi)$ to omsg, and output $(\mathsf{CreatedPuzzle}, \mathsf{sid}, \mathsf{puz}, \pi)$ to $\mathcal{P}_o$ and $(\mathsf{CreatedPuzzle}, \mathsf{sid}, \mathsf{puz})$ to $\mathcal{S}$.

Solve: Upon receiving $(\mathsf{Solve}, \mathsf{sid}, \mathsf{puz} = (\mathtt{st}_0, \Gamma, \mathtt{tag}))$ from $\mathcal{P}_i \in \mathcal{P}$, add $(\mathcal{P}_i, \mathsf{sid}, \mathsf{puz}, \mathtt{st}_0, 0)$ to L and send $(\mathsf{Solve}, \mathsf{sid}, \mathsf{puz})$ to $\mathcal{S}$.

Advance State: Upon receiving $(\mathsf{AdvanceState}, \mathsf{sid})$ from $\mathcal{P}_i \in \mathcal{P}$, set $f_i = 1$.

Public Verification: Upon receiving $(\mathsf{Verify}, \mathsf{sid}, \mathsf{puz} = (\mathtt{st}, \Gamma, \mathtt{tag}), m, \pi)$ from a party $\mathcal{P}_i \in \mathcal{P}$, add $(\mathcal{P}_i, \mathsf{sid}, g(\Gamma), \mathtt{st}, \Gamma, \mathtt{tag}, m, \pi)$ to $\mathcal{Q}_{\mathsf{v}}$.

Fig. 4: Ticked Functionality $\mathcal{F}_{\mathsf{TLP}}$ for publicly verifiable time-lock puzzles (Part 1).

In the full version [5], we show that the TLP from [6] realizes a slightly weaker version of $\mathcal{F}_{\mathsf{TLP}}$ and present a new Protocol π_{tlp} that realizes $\mathcal{F}_{\mathsf{TLP}}$ (*i.e.* proving Theorem 1). Protocol π_{tlp} is constructed from a standalone trapdoor VDF modeled by $\mathcal{F}_{\mathsf{psc}}$. A puzzle owner $\mathcal{P}_o$ uses the trapdoor to compute the VDF on a random input $\mathtt{st}_0$ for the number of steps Γ required by the PV-TLP, obtaining the corresponding output $\mathtt{st}_\Gamma$ and proof π. The owner then computes $\mathtt{tag}_1 = H_1(\mathtt{st}_0, \Gamma, \mathtt{st}_\Gamma, \pi) \oplus m$, $\mathtt{tag}_2 = H_2(\mathtt{st}_0, \Gamma, \mathtt{st}_\Gamma, \pi, \mathtt{tag}_1, m)$ and $\mathtt{tag} = (\mathtt{tag}_1, \mathtt{tag}_2)$, where m is the message in the puzzle, using random oracles H_1 and H_2. The final puzzle is $\mathtt{puz} = (\mathtt{st}_0, \Gamma, \mathtt{tag})$. A solver computes Γ steps of the trapdoor VDF with input $\mathtt{st}_0$ to get a proof of PV-TLP solution $\pi' = (\mathtt{st}_\Gamma, \pi)$, which can be used to check the consistency of $\mathtt{tag}$ and retrieve m. If $\mathtt{tag}$ is not consistent, π' can also be used to verify this fact.

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

Functionality $\mathcal{F}_{\mathsf{TLP}}$ (Part 2)

Tick: – For all $(\mathcal{P}_i, \mathsf{sid}, \mathsf{puz} = (\mathtt{st}_0, \Gamma, \mathtt{tag}), \mathtt{st}_c, c) \in L$, if $f_i = 1$ proceed as follows:

1. If there is no $\mathtt{st}_{c+1}$ such that $(\mathtt{st}_c, \mathtt{st}_{c+1}) \in \mathsf{steps}$:
   (a) If $\mathcal{P}_o$ is honest, sample $\mathtt{st}_{c+1} \xleftarrow{\$} \mathcal{ST}$, and append $(\mathtt{st}_c, \mathtt{st}_{c+1})$ to steps.
   (b) If $\mathcal{P}_o$ is corrupted, send $(\mathsf{sid}, \mathsf{adv}, \mathtt{st}_c)$ to $\mathcal{S}$ and wait for $(\mathsf{sid}, \mathsf{adv}, \mathtt{st}_c, \mathtt{st}_{c+1})$. If $\mathtt{st}_{c+1} \notin \mathcal{ST}$ then halt. Otherwise, append $(\mathtt{st}_c, \mathtt{st}_{c+1})$ to steps.
2. Output $(\mathtt{st}_c, \mathtt{st}_{c+1})$ to $\mathcal{S}$ and update $(\mathcal{P}_i, \mathsf{sid}, \mathsf{puz}, \mathtt{st}_c, c) \in L$ by setting $c = c+1$.
3. If $c \geq \epsilon\Gamma$ and there is no $(\mathtt{st}_c, \mathtt{st}_{c+1}), (\mathtt{st}_{c+1}, \mathtt{st}_{c+2}), \ldots, (\mathtt{st}_{\Gamma-1}, \mathtt{st}_\Gamma) \in \mathsf{steps}$:
   (a) If $\mathcal{P}_o$ is honest, sample $\mathtt{st}_j \xleftarrow{\$} \mathcal{ST}$ for $j = c+1, c+2, \ldots, \Gamma$, and output $(\mathtt{st}_{j-1}, \mathtt{st}_j)$ to $\mathcal{S}$ and append $(\mathtt{st}_{j-1}, \mathtt{st}_j)$ to steps. If $(\mathtt{st}_0, \Gamma, \mathtt{tag}, m, \pi) \notin \mathsf{omsg}$, set $m = \perp$, sample $\pi \xleftarrow{\$} \mathcal{PROOF}$ and append $(\mathtt{st}_0, \Gamma, \mathtt{tag}, \perp, \pi) \in \mathsf{omsg}$. Finally, output $(\mathsf{Solved}, \mathsf{sid}, \mathsf{puz}, m, \pi)$ to $\mathcal{S}$.
   (b) Else (if $\mathcal{P}_o$ is corrupted), send $(\mathsf{GetSts}, \mathsf{sid}, \mathsf{puz})$ to $\mathcal{S}$, wait for $\mathcal{S}$ to answer with $(\mathsf{GetSts}, \mathsf{sid}, \mathsf{puz}, \mathtt{st}_c, \mathtt{st}_{c+1}, \ldots, \mathtt{st}_\Gamma)$. For $j = c+1, \ldots, \Gamma$, if $\mathtt{st}_j \notin \mathcal{ST}$ or $(\mathtt{st}_{j-1}, \mathtt{st}'_j) \in \mathsf{steps}$, then $\mathcal{F}_{\mathsf{TLP}}$ halts, else, append $(\mathtt{st}_{j-1}, \mathtt{st}_j)$ to steps.
4. Else If $c \geq \epsilon\Gamma$ and there exist $(\mathtt{st}_c, \mathtt{st}_{c+1}), \ldots, (\mathtt{st}_{\Gamma-1}, \mathtt{st}_\Gamma) \in \mathsf{steps}$, or if $(\mathsf{puz}' = (\mathtt{st}_0, \Gamma, \mathtt{tag}'), m', \pi') \in \mathsf{omsg}$ s.t. $\mathtt{tag}' \neq \mathtt{tag}$ (*i.e.* a puzzle with same $\mathtt{st}_0, \Gamma$ has been solved) or $\mathcal{P}_o$ is corrupted and $(\mathsf{puz}, m, \pi) \notin \mathsf{omsg}$, send $(\mathsf{GetMsg}, \mathsf{sid}, \mathsf{puz})$ to $\mathcal{S}$, wait for $\mathcal{S}$ to answer with $(\mathsf{GetMsg}, \mathsf{sid}, \mathsf{puz}, m, \pi)$. If $\pi \notin \mathcal{PROOF}$ or $(\mathtt{st}'_0, \Gamma', \mathtt{tag}', m', \pi) \in \mathsf{omsg}$, $\mathcal{F}_{\mathsf{TLP}}$ halts, else, append $(\mathtt{st}_0, \Gamma, \mathtt{tag}, m, \pi)$ to omsg.
5. If $c = \Gamma$, remove $(\mathcal{P}_i, \mathsf{sid}, \mathsf{puz}, \mathtt{st}_c, c) \in L$ and send $(\mathsf{Solved}, \mathsf{sid}, \mathsf{puz}, m, \pi)$ to $\mathcal{P}_i$.

– For each $(\mathcal{P}_i, \mathsf{sid}, c, \mathtt{st}, \Gamma, \mathtt{tag}, m, \pi) \in \mathcal{Q}_{\mathsf{v}}$, if $f_i = 1$ proceed as follows: 1. If $c = 0$, remove $(\mathcal{P}_i, \mathsf{sid}, 0, \mathtt{st}, \Gamma, \mathtt{tag}, m, \pi)$ from $\mathcal{Q}_{\mathsf{v}}$, set $b = 1$ if $(\mathtt{st}, \Gamma, \mathtt{tag}, m, \pi) \in \mathsf{omsg}$, otherwise set $b = 0$ and output $(\mathsf{Verified}, \mathsf{sid}, \mathsf{puz} = (\mathtt{st}, \Gamma, \mathtt{tag}), m, \pi, b)$ to $\mathcal{P}_i$; 2. Else, if $c > 0$: update $(\mathcal{P}_i, \mathsf{sid}, c, \mathtt{st}, \Gamma, \mathtt{tag}, m, \pi) \in \mathcal{Q}_{\mathsf{v}}$ by setting $c = c - 1$.

Set $f_i = 0$ for $i = 1, \ldots, n$.

Fig. 5: Ticked Functionality $\mathcal{F}_{\mathsf{TLP}}$ for publicly verifiable time-lock puzzles (Part 2).

4 Universally Composable Verifiable Delay Functions

We present a generic UC construction of VDFs as modeled in functionality $\mathcal{F}_{\mathsf{VDF}}$ (Figure 6) from a generic verifiable sequential computation scheme modeled in functionality $\mathcal{F}_{\mathsf{psc}}$ (Figure 2) and a global random oracle $\mathcal{G}_{\mathsf{rpoRO}}$. Our construction is presented in protocol π_{VDF} (Figure 7).

Verifiable Delay Functions We model the UC VDF in Functionality $\mathcal{F}_{\mathsf{VDF}}$. It ensures that each computational step of the VDF evaluation takes at least a fixed amount of time (one tick) and guarantees that the output obtained after a number of steps is uniformly random and unpredictable even to the adversary. However, it allows that the adversary obtains the output of evaluating a VDF for Γ steps in only $\epsilon\Gamma$ steps for $0 < \epsilon \leq 1$, modeling the slack between concrete computational complexities for honest parties and for the adversary

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

$\mathcal{F}_{\mathsf{VDF}}$ is parameterized by a computational security parameter τ, a state space $\mathcal{ST}$, a proof space $\mathcal{PROOF}$, a slack parameter $0 < \epsilon \leq 1$ and a function $g : \{0,1\}^\star \mapsto \mathbb{N}$ (determining how many ticks it takes to verify a proof). $\mathcal{F}_{\mathsf{VDF}}$ interacts with a set of parties $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, and an adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{VDF}}$ maintains flags f_i for $i = 1, \ldots, n$ that are initially set to 0 and initially empty lists steps (state transitions), $\mathcal{Q}_{\mathsf{v}}$ (proofs being verified), L (proofs being computed), and OUT (outputs).

Solve: Upon receiving $(\mathsf{Solve}, \mathsf{sid}, in, \Gamma)$ from $\mathcal{P}_i \in \mathcal{P}$ where $in \in \mathcal{ST}$ and $\Gamma \in \mathbb{N}$, add $(\mathcal{P}_i, \mathsf{sid}, in, \Gamma, in, 0)$ to L and send $(\mathsf{Solve}, \mathsf{sid}, in, \Gamma)$ to $\mathcal{S}$.

Advance State: Upon receiving $(\mathsf{AdvanceState}, \mathsf{sid})$ from $\mathcal{P}_i \in \mathcal{P}$, set $f_i = 1$.

Tick: – For each $(\mathcal{P}_i, \mathsf{sid}, in, \Gamma, \mathtt{st}_c, c) \in L$, if $f_i = 1$ proceed as follows:

1. If there is no $\mathtt{st}_{c+1}$ such that $(\mathtt{st}_c, \mathtt{st}_{c+1}) \in \mathsf{steps}$, send $(\mathsf{sid}, \mathsf{adv}, \mathtt{st}_c)$ to $\mathcal{S}$ and wait for $(\mathsf{sid}, \mathsf{adv}, \mathtt{st}_c, \mathtt{st}_{c+1})$. If $\mathtt{st}_{c+1} \notin \mathcal{ST}$ or $(\mathtt{st}_c, \mathtt{st}'_{c+1}) \in \mathsf{steps}$ for some $\mathtt{st}'_{c+1} \in \mathcal{ST}$ then halt. Otherwise, append $(\mathtt{st}_c, \mathtt{st}_{c+1})$ to steps. Finally update $(\mathcal{P}_i, \mathsf{sid}, in, \Gamma, \mathtt{st}_c, c) \in L$ by setting $c = c + 1$.
2. If $c \geq \epsilon\Gamma$ and there is no $(\mathtt{st}_c, \mathtt{st}_{c+1}), (\mathtt{st}_{c+1}, \mathtt{st}_{c+2}), \ldots, (\mathtt{st}_{\Gamma-1}, out) \in \mathsf{steps}$, sample $out \xleftarrow{\$} \mathcal{ST}$, send $(\mathsf{GetStsPf}, \mathsf{sid}, in, \Gamma, \mathtt{st}_c, out)$ to $\mathcal{S}$, wait for $\mathcal{S}$ to answer with $(\mathsf{GetStsPf}, \mathsf{sid}, \mathtt{st}_{c+1}, \ldots, \mathtt{st}_{\Gamma-1}, \Pi)$. If $\mathtt{st}_j \notin \mathcal{ST}$ or $(\mathtt{st}_{j-1}, \mathtt{st}'_j) \in \mathsf{steps}$, for $j \in \{c+1, \ldots, \Gamma-1\}$, or $\Pi \notin \mathcal{PROOF}$, or there exists $(in', \Gamma', out', \Pi) \in \mathsf{OUT}$, $\mathcal{F}_{\mathsf{VDF}}$ halts. Otherwise, append $(\mathtt{st}_{j-1}, \mathtt{st}_j)$ to steps, for $j \in \{c+1, \ldots, \Gamma-1\}$, append $(\mathtt{st}_{\Gamma-1}, out)$ to steps and (in, Γ, out, Π) to OUT.
3. If $c = \Gamma$, remove $(\mathcal{P}_i, \mathsf{sid}, in, \Gamma, out, \Gamma) \in L$, send $(\mathsf{Proof}, \mathsf{sid}, in, \Gamma, out, \Pi)$ to $\mathcal{P}_i$.

– For each $(\mathcal{P}_i, \mathsf{sid}, c, in, \Gamma, out, \Pi) \in \mathcal{Q}_{\mathsf{v}}$, if $f_i = 1$ proceed as follows: 1. If $c = 0$, remove $(\mathcal{P}_i, \mathsf{sid}, 0, in, \Gamma, out, \Pi)$ from $\mathcal{Q}_{\mathsf{v}}$, set $b = 1$ if $(in, \Gamma, out, \Pi) \in \mathsf{OUT}$, otherwise set $b = 0$ and output $(\mathsf{Verified}, \mathsf{sid}, in, \Gamma, out, \Pi, b)$ to $\mathcal{P}_i$; 2. If $c > 0$, update $(\mathcal{P}_i, \mathsf{sid}, c, in, \Gamma, out, \Pi) \in \mathcal{Q}_{\mathsf{v}}$ by setting $c = c - 1$.

Set $f_i = 0$ for $i = 1, \ldots, n$.

Verification: Upon receiving $(\mathsf{Verify}, \mathsf{sid}, in, \Gamma, out, \Pi)$ from a party $\mathcal{P}_i \in \mathcal{P}$, add $(\mathcal{P}_i, \mathsf{sid}, g(\Gamma), in, \Gamma, out, \Pi)$ to $\mathcal{Q}_{\mathsf{v}}$.

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

in sequential computation assumptions. 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 verify such proofs is variable and modeled as a function $g(\Gamma)$. Moreover, $\mathcal{F}_{\mathsf{VDF}}$ 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. 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 neither affects the soundness of the VDF nor the randomness of its output, but is necessary for simulation.

Functionality $\mathcal{F}_{\mathsf{VDF}}$ allows for a party to start evaluating the VDF for Γ steps on an input in by activating it with message $(\mathsf{Solve}, \mathsf{sid}, in, \Gamma)$. After this initial request, the party needs to activate $\mathcal{F}_{\mathsf{VDF}}$ with message $(\mathsf{AdvanceState}, \mathsf{sid})$ on Γ

different ticks in order to receive the result of the VDF evaluation. This is taken care of by the Tick interface of $\mathcal{F}_{\mathsf{VDF}}$, whose instructions are executed after every new tick, causing $\mathcal{F}_{\mathsf{VDF}}$ to iterate over every pending VDF evaluation request from parties who have activated $\mathcal{F}_{\mathsf{VDF}}$ in the previous tick. Each evaluation is performed by asking the adversary for a representation of the next intermediate state $\mathtt{st}_{c+1}$. When $\epsilon\Gamma$ steps have been evaluated, $\mathcal{F}_{\mathsf{VDF}}$ leaks the output out to the adversary $\mathcal{S}$. When all Γ steps have been evaluated by $\mathcal{F}_{\mathsf{VDF}}$, it outputs out and a proof Π that this output was obtained from in after Γ steps. Moreover, parties who have an input in and a potential proof Π that out was obtained as output after evaluating the VDF for Γ steps on this input can activate $\mathcal{F}_{\mathsf{VDF}}$ with message $(\mathsf{Verify}, \mathsf{sid}, in, \Gamma, out, \Pi)$ to verify the proof. Once a proof verification request has been made, the party needs to activate $\mathcal{F}_{\mathsf{VDF}}$ with message $(\mathsf{AdvanceState}, \mathsf{sid})$ on $g(\Gamma)$ different ticks to receive the result of the verification.

Protocol π_{VDF}

Protocol π_{VDF} is parameterized by a security parameter τ and is executed by a set of parties $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ interacting with functionalities $\mathcal{G}_{\mathsf{ticker}}$,$\mathcal{G}_{\mathsf{rpoRO}}$,$\mathcal{F}_{\mathsf{psc}}$ (whose state space is $\mathcal{ST}$, proof space is $\mathcal{PROOF}$ and $\mathcal{P}_o = \perp$, *i.e.* no $\mathcal{P}_i \in \mathcal{P}$ can access the trapdoor solve interface). All parties locally keep initially empty lists L and $\mathcal{Q}_{\mathsf{v}}$.

Solve: Upon receiving input $(\mathsf{Solve}, \mathsf{sid}, in, \Gamma)$ where $in \in \mathcal{ST}$ and $\Gamma \in \mathbb{N}$, $\mathcal{P}_i$ sends $(\mathsf{Solve}, \mathsf{sid}, in, \Gamma)$ to $\mathcal{F}_{\mathsf{psc}}$, and adds $(\mathsf{sid}, in, \Gamma)$ to L.

Advance State: Upon receiving input $(\mathsf{AdvanceState}, \mathsf{sid})$, $\mathcal{P}_i$ sends $(\mathsf{AdvanceState}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{psc}}$.

Tick: $\mathcal{P}_i$ processes proofs being computed or verified as follows:

1. Upon receiving $(\mathsf{GetPf}, \mathsf{sid}, in, \Gamma, \mathtt{el}_\Gamma, \pi)$ from $\mathcal{F}_{\mathsf{psc}}$ s.t. there is $(\mathsf{sid}, in, \Gamma) \in L$:
   (a) Send $(\textsc{Hash-Query}, (in|\Gamma|\mathtt{el}_\Gamma|\pi))$ to $\mathcal{G}_{\mathsf{rpoRO}}$, getting $(\textsc{Hash-Confirm}, h)$.
   (b) Set $out = h$ and $\Pi = (\mathtt{el}_\Gamma, \pi)$; output $(\mathsf{Proof}, \mathsf{sid}, in, \Gamma, out, \Pi)$ and remove $(\mathsf{sid}, in, \Gamma)$ from L.
2. Upon receiving $(\mathsf{Verified}, \mathsf{sid}, in, \Gamma, \mathtt{el}_\Gamma, \pi, b)$ from $\mathcal{F}_{\mathsf{psc}}$ s.t. there is $(\mathcal{P}_i, \mathsf{sid}, in, \Gamma, out, \Pi = (\mathtt{el}_\Gamma, \pi)) \in \mathcal{Q}_{\mathsf{v}}$:
   (a) Send $(\textsc{Hash-Query}, (in|\Gamma|\mathtt{el}_\Gamma|\pi))$ to $\mathcal{G}_{\mathsf{rpoRO}}$, getting $(\textsc{Hash-Confirm}, h)$.
   (b) Set $b' = 0$ if $b = 0$ or $h \neq out$, otherwise set $b' = 1$. Output $(\mathsf{Verified}, \mathsf{sid}, in, \Gamma, out, \Pi, b')$ and remove $(\mathcal{P}_i, \mathsf{sid}, in, \Gamma, out, \Pi)$ from $\mathcal{Q}_{\mathsf{v}}$.

Verification: Upon receiving input $(\mathsf{Verify}, \mathsf{sid}, in, \Gamma, out, \Pi)$, $\mathcal{P}_i$ parses $\Pi = (\mathtt{el}_\Gamma, \pi)$, and sends $(\mathsf{Verify}, \mathsf{sid}, in, \Gamma, \mathtt{el}_\Gamma, \pi)$ to $\mathcal{F}_{\mathsf{psc}}$, and adds $(\mathcal{P}_i, \mathsf{sid}, in, \Gamma, out, \Pi)$ to $\mathcal{Q}_{\mathsf{v}}$.

Fig. 7: Protocol π_{VDF} realizing Verifiable Delay Functions functionality $\mathcal{F}_{\mathsf{VDF}}$ in the $\mathcal{F}_{\mathsf{psc}}, \mathcal{G}_{\mathsf{rpoRO}}$-hybrid model.

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 7. We use an instance of $\mathcal{F}_{\mathsf{psc}}$ where $\mathcal{P}_o = \perp$, mean-

ing that no party in $\mathcal{P}$ has access to the trapdoor evaluation interface. 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 and proof $\mathtt{el}_\Gamma, \pi$ are obtained, the output is defined as $out = H(\mathsf{sid}|\Gamma|\mathtt{el}_\Gamma|\pi)$ where H is an instance of $\mathcal{G}_{\mathsf{rpoRO}}$ and the VDF proof is defined as $\Pi = (\mathtt{el}_\Gamma, \pi)$. 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 $out' = H(sid|\Gamma|\mathtt{el}_\Gamma|\pi)$, 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, and the proof is presented in the full version[5] due to space limitations.

Theorem 2. *Protocol π_{VDF} (G)UC-realizes $\mathcal{F}_{\mathsf{VDF}}$ in the $\mathcal{G}_{\mathsf{ticker}}, \mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{psc}}$-hybrid model with computational security against a static adversary: there exists a simulator $\mathcal{S}$ such that for every static adversary $\mathcal{A}$ no environment $\mathcal{Z}$ can distinguish π_{VDF} composed with $\mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{psc}}$ and $\mathcal{A}$ from $\mathcal{S}$ composed with $\mathcal{F}_{\mathsf{VDF}}$.*

5 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 8. 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 $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, verifiers $\mathcal{V}$ and an adversary $\mathcal{S}$ through the following interfaces:

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

Verify: Upon receiving $(\textsc{Verify}, \mathsf{sid}, x)$ from $\mathcal{V}_j \in \mathcal{V}$, if $(\textsc{Tossed}, \mathsf{sid}, x)$ has been sent to all parties in $\mathcal{P}$ set $f = 1$, else set $f = 0$. Send $(\textsc{Verify}, \mathsf{sid}, x, f)$ 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. 8: Ticked Functionality ${\mathcal{F}_{\mathsf{RB}}}^{\Delta_{\mathsf{TLP-RB}}}$ for Randomness Beacons.

5.1 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 $\epsilon\delta$ ticks (*i.e.* the amount of ticks needed for the adversary to solve honest party TLPs) 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 broadcast 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).

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.

We describe protocol $\pi_{\mathsf{TLP-RB}}$ in Figure 9 and state its security in Theorem 3. The proof is presented in the full version[5] due to space limitations.

Theorem 3. *If* Δ *is finite (though unknown) and all* $\mathcal{P}_i \in \mathcal{P}$ *receive inputs within a delay of* δ *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* $\Delta_{\mathsf{TLP-RB}} = 3(\epsilon^{-1}\Delta + 1) + \sum_{i=1}^{\epsilon^{-1}\Delta} i$*, where* ϵ *is* $\mathcal{F}_{\mathsf{TLP}}$*'s slack parameter. 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* $\mathcal{F}_{\mathsf{RB}}^{\Delta_{\mathsf{TLP-RB}}}$*.*

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

The protocol is executed by parties $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ out of which $t < n/2$ are corrupted and verifiers $\mathcal{V}$, who interact with $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ and instances $\mathcal{F}_{\mathsf{TLP}}^{i}$ of $\mathcal{F}_{\mathsf{TLP}}$ with slack parameter ϵ for which $\mathcal{P}_i$ acts as $\mathcal{P}_o$. The initial delay parameter is δ.

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

1. **Commitment Phase:** For $i \in \{1, \ldots, n\}$, party $\mathcal{P}_i$ proceeds as follows:
   (a) Sample $r_i \xleftarrow{\$} \{0,1\}^\tau$ and send $(\mathsf{CreatePuzzle}, \mathsf{sid}, \delta, r_i)$ to $\mathcal{F}_{\mathsf{TLP}}^{i}$, receiving $(\mathsf{CreatedPuzzle}, \mathsf{sid}, \mathtt{puz}_i, \pi_i)$ in response.
   (b) Send $(\mathsf{Send}, \mathsf{sid}, \mathsf{ssid}, \mathtt{puz}_i)$ to $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ and send $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$.
   (c) Wait for $\mathcal{P}_j \in \mathcal{P}$ to broadcast their TLPs within $\epsilon\delta$ ticks (*i.e.* before the adversary solves $\mathtt{puz}_i$) by sending $(\mathsf{Solve}, \mathsf{sid}, \mathtt{puz}' = (\mathtt{st}', \epsilon\delta, \mathtt{tag}'))$ to $\mathcal{F}_{\mathsf{TLP}}$ (*i.e.* solving a dummy TLP with $\epsilon\delta$ steps to count the ticks) and proceeding as follows when activated: i. Send $(\mathsf{Fetch}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$, receiving $(\mathsf{Fetch}, \mathsf{sid}, L)$; ii. Check that there exist $1 + n/2$ messages $(\mathcal{P}_i, \mathsf{sid}, (\mathcal{P}_j, \mathtt{puz}_j, \mathsf{ssid}))$ in L from different $\mathcal{P}_j$ and, if yes, let $\mathcal{C} = \{\mathcal{P}_j\}_{1 \le j \le 1+n/2}$ and proceed to the **Opening Phase**, else, send $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$; iii. If $(\mathsf{Solved}, \mathsf{sid}, \mathtt{puz}', \perp, \pi')$ is received from $\mathcal{F}_{\mathsf{TLP}}$, $\epsilon\delta$ ticks have passed, so increment δ and go to Step 1(a).
2. **Opening Phase:** All parties $\mathcal{P}_i \in \mathcal{C}$ proceed as follows:
   (a) Send $(\mathsf{Send}, \mathsf{sid}, \mathsf{ssid}', r_i, \pi_i)$ to $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$.
   (b) Wait δ ticks for all $\mathcal{P}_j \in \mathcal{C}$ to broadcast their TLP solutions by sending $(\mathsf{Solve}, \mathsf{sid}, \mathtt{puz}_i)$ to $\mathcal{F}_{\mathsf{TLP}}$ and only proceeding to Step 2(c) when $(\mathsf{Solved}, \mathsf{sid}, \mathtt{puz}_i, r_i, \pi_i)$ is received from $\mathcal{F}_{\mathsf{TLP}}$, sending $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$ otherwise;
   (c) Send $(\mathsf{Fetch}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$, receiving $(\mathsf{Fetch}, \mathsf{sid}, L)$. Check that every message of the form $(\mathcal{P}_i, \mathsf{sid}, (\mathcal{P}_j, r_j, \pi_j, \mathsf{ssid}'))$ from $\mathcal{P}_j \in \mathcal{C}$ is a valid solution to $\mathtt{puz}_j$ by sending $(\mathsf{Verify}, \mathsf{sid}, \mathtt{puz}_j, r_j, \pi_j)$ to $\mathcal{F}_{\mathsf{TLP}}^{j}$ and checking that the answer received later is $(\mathsf{Verified}, \mathsf{sid}, \mathtt{puz}_j, r_j, \pi_j, 1)$. 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 $i \in \{1, \ldots, n\}$, party $\mathcal{P}_i$ proceeds as follows:
   (a) For each j such that $\mathcal{P}_j \in \mathcal{C}$ did not send a valid solution of $\mathtt{puz}_j$, send $(\mathsf{Solve}, \mathsf{sid}, \mathtt{puz}_j)$ to $\mathcal{F}_{\mathsf{TLP}}$. When activated, if $(\mathsf{Solved}, \mathsf{sid}, \mathtt{puz}_j, r_j, \pi_j)$ is received from $\mathcal{F}_{\mathsf{TLP}}$, send $(\mathsf{Send}, \mathsf{sid}, \mathsf{ssid}'', (r_j, \pi_j))$ to $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ and $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$.
   (b) Let $\mathcal{G}$ be the set of all solutions $r_j \neq \perp$ of $\mathtt{puz}_j$ such that $\mathcal{P}_j \in \mathcal{C}$ (*i.e.* $\mathcal{G}$ is the set of solutions r_j from valid TLPs posted in the commitment phase). Compute $r = \bigoplus_{r_j \in \mathcal{G}} r_j$, output $(\textsc{Tossed}, \mathsf{sid}, r)$ and send $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$.
4. **Verify:** On input $(\textsc{Verify}, \mathsf{sid}, x)$, $\mathcal{V}_j \in \mathcal{V}$ proceeds as follows:
   (a) Send $(\mathsf{Fetch}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$, receiving $(\mathsf{Fetch}, \mathsf{sid}, L)$ and determining $\mathcal{C}$ by looking for the first $1 + n/2$ messages of the form $(\mathcal{P}_i, \mathsf{sid}, (\mathcal{P}_j, \mathtt{puz}_j, \mathsf{ssid}))$;
   (b) Check that each message of the form $(\mathcal{P}_i, \mathsf{sid}, (\mathcal{P}_h, r_j, \pi_j, \mathsf{ssid}'))$ in L for $\mathcal{P}_j \in \mathcal{C}$ and $\mathcal{P}_h \in \mathcal{P}$ (*i.e.* solutions to a puzzle $\mathtt{puz}_j$ from a party $\mathcal{P}_j \in \mathcal{C}$ sent by $\mathcal{P}_j$ or by any party $\mathcal{P}_h \in \mathcal{P}$ who solved an unopened $\mathtt{puz}_j$ in the recovery phase) contains a valid solution to $\mathtt{puz}_j$ by sending $(\mathsf{Verify}, \mathsf{sid}, \mathtt{puz}_j, r_j, \pi_j)$ to $\mathcal{F}_{\mathsf{TLP}}^{j}$ and checking that the answer is $(\mathsf{Verified}, \mathsf{sid}, \mathtt{puz}_j, r_j, \pi_j, 1)$;
   (c) 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 $r_j \neq \perp$. If $x = \bigoplus_{r_j \in \mathcal{G}} r_j$, set $f = 1$, else set $f = 0$, output $(\textsc{Verify}, \mathsf{sid}, x, f)$.

Fig. 9: Protocol $\pi_{\mathsf{TLP-RB}}$ for a randomness beacon based on PV-TLPs.

5.2 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 4. *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 $\Delta_{\mathsf{TLP-RB}} = 3(\epsilon^{-1}\Delta + 1) + \sum_{i=1}^{\epsilon^{-1}\Delta} i$, where ϵ is $\mathcal{F}_{\mathsf{TLP}}$'s slack parameter. 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. The proof is presented in the full version[5] due to space limitations. □

5.3 Randomness Beacons from VDFs

It has been suggested that VDFs can be used to obtain a randomness beacon [12] via a simple protocol where parties post plaintext values $r_1, \ldots, r_n$ on a public ledger and then evaluate a VDF on input $H(r_1|\ldots|r_n)$, where $H()$ is a cryptographic hash function, in order to obtain a random output r. However, despite being used in industry [40], 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 5 (in the full version[5]), which characterizes the worst case execution time.

Theorem 5. *If $\Delta = \mathsf{maxTXDelay} + \mathsf{emptyBlocks} \cdot \mathsf{slackWindow}$ (computed from $\mathcal{F}_{\mathsf{Ledger}}$'s parameters) is finite (but unknown), Protocol $\pi_{\mathsf{VDF-RB}}$ UC-realizes ${\mathcal{F}_{\mathsf{RB}}}^{\Delta_{\mathsf{TLP-RB}}}$ in the $\mathcal{F}_{\mathsf{VDF}}, \mathcal{F}_{\mathsf{Ledger}}$-hybrid model with computational security against static adversaries corrupting $t < n/2$ parties for $\Delta_{\mathsf{TLP-RB}} = 2(\epsilon^{-1}\Delta+1)+\sum_{i=1}^{\epsilon^{-1}\Delta} i$, where ϵ is $\mathcal{F}_{\mathsf{VDF}}$'s slack parameter. There is a simulator $\mathcal{S}$ s.t. for every static adversary $\mathcal{A}$, and any environment $\mathcal{Z}$, $\mathcal{Z}$ cannot distinguish an execution of $\pi_{\mathsf{VDF-RB}}$ by $\mathcal{A}$ composed with $\mathcal{F}_{\mathsf{VDF}}, \mathcal{F}_{\mathsf{Ledger}}$ from an ideal execution with $\mathcal{S}$ and ${\mathcal{F}_{\mathsf{RB}}}^{\Delta_{\mathsf{TLP-RB}}}$.*

6 MPC with (Punishable) Output-Independent Abort

In this section we will describe how to construct a protocol that achieves MPC with output-independent abort. The starting point of this construction will be MPC with secret-shared output[7], which is a strictly weaker primitive, as well as the broadcast as modeled in $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ and Commitments with Delayed Openings $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta,\zeta}$. In the full version[5], we subsequently show how to financially penalize cheating behavior in the protocol (POIA-MPC).

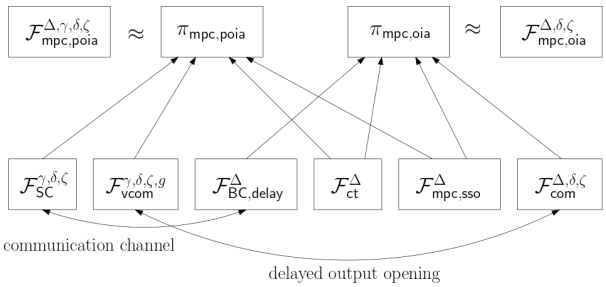

Fig. 10: How MPC with (Punishable) Output-Independent Abort is constructed.

6.1 Functionalities for Output-Independent Abort

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

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

In the full version [5], we also introduce the following functionalities:

1. $\mathcal{F}_{\mathsf{ct}}^{\Delta}$ for coin-flipping with abort.
2. $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta,\zeta}$ for commitments with delayed non-interactive openings.
3. $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta,\zeta,g}$ for commitments with verifiable delayed non-interactive openings.
4. $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta,\zeta}$ which is an abstraction of a smart contract.
5. $\mathcal{F}_{\mathsf{mpc,poia}}^{\Delta,\gamma,\delta,\zeta}$ for POIA-MPC.

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 $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta,\zeta}$ 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 $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta,\zeta}$. Then, we will show how to

[7] For the sake of efficiency we focus on an output phase that uses additive secret sharing. However, the core MPC computation could use any secret sharing scheme, while only the output phase is restricted to additive secret sharing. This approach can be generalized by using a generic MPC protocol that computes an additive secret sharing of the output as part of the evaluated circuit, although at an efficiency cost. We remark that efficient MPC protocols matching our requirements do exist, e.g. [29].

implement $\mathcal{F}_{\mathsf{mpc,poia}}^{\Delta,\gamma,\delta,\zeta}$ via the protocol $\pi_{\mathsf{mpc,poia}}$ (a generalization of $\pi_{\mathsf{mpc,oia}}$) which uses $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta,\zeta}, \mathcal{F}_{\mathsf{ct}}^{\Delta}, \mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$ as well as $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta,\zeta,g}$. As mentioned in Fig. 10, $\mathcal{F}_{\mathsf{vcom}}^{\gamma,\delta,\zeta,g}$ and $\mathcal{F}_{\mathsf{SC}}^{\gamma,\delta,\zeta}$ are modifications of $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta,\zeta}$ and $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$. We now describe the functionalities required to build $\pi_{\mathsf{mpc,oia}}$ 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 $\mathsf{Compute}$.

Tick:

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

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

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

Fig. 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 [9] or multiparty BMR [29].

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. We describe the functionality $\mathcal{F}^{\Delta,\delta,\zeta}_{\mathsf{com}}$ for commitments with delayed non-interactive openings in the full version[5]. The functionality distinguishes between a sender $\mathcal{P}_{\mathsf{Send}}$, which can make commitments, and a set of receivers, which obtain the openings. Compared to regular commitments with a normal **Open** that immediately reveals the output to all

parties, $\mathcal{P}_{\mathsf{Send}}$ is also allowed to perform a **Delayed Open**, where there is a delay between the choice of a sender to open a commitment (or not) and the actual opening towards receivers and the adversary.

While both **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, $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta,\zeta}$ 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 honest parties learn the opening of the commitment. This means that for honest parties, it may take up to $\Delta + \delta$ ticks depending on when DOpen is obtained. The simulator will already learn the opening after $\zeta \leq \delta$ ticks, similar to how it might solve $\mathcal{F}_{\mathsf{TLP}}$ faster. $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta,\zeta}$ ensures that *all honest parties will learn the delayed opening simultaneously.*

In the full version[5], we provide a secure instantiation of a publicly verifiable version of $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta,\zeta}$. Since we do not require homomorphic operations, this means that it can be realized with a much simpler protocol than the respective two-party functionality in [6].

MPC with Output-Independent Abort. In Fig. 13 and Fig. 14 we describe the functionality $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta,\zeta}$ 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 $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta,\zeta}$. 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 ζ ticks, decide whether to reveal its shares or not. Then, once these ζ 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 $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta,\zeta}$ 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 $\mathcal{F}_{\mathsf{ct}}^{\Delta}$, which we present in the full version [5]. Note that $\mathcal{F}_{\mathsf{ct}}^{\Delta}$ can easily be realized in the $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}, \mathcal{F}_{\mathsf{com}}^{\Delta,\delta,\zeta}$-hybrid model.

6.2 Building MPC with Output-Independent Abort

We will now describe how to construct an MPC protocol that guarantees output-independent abort. Although this might appear like a natural generalization

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

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

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 $\{x_i\}_{i\in[n]}$ 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 $\mathcal{S}$ 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: Ticked $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta,\zeta}$ Functionality for MPC with Output-Independent Abort.

of [6], 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 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 $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$.

Functionality $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta,\zeta}$ (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 δ. If $J \neq \emptyset$ then send message J with prefix Abort to each party $\mathcal{P} \setminus I$ via $\mathcal{Q}$ with delay δ.
   (c) Send $(\mathsf{Output}, \mathsf{sid}, y)$ and the IDs to $\mathcal{S}$.
4. If $\mathtt{st} = \mathtt{sync}$ and $f = \top$ then set $\mathtt{st} = \mathtt{wait}(\zeta)$ 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: Ticked $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta,\zeta}$ Functionality for MPC with Output-Independent Abort.

The purpose of the heartbeat is to ensure that honest parties are synchronized throughout the protocol, allowing them to later achieve agreement on the corrupt parties.

2. The parties provide 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 $\boldsymbol{y}$. They also sample a blinding value $\boldsymbol{r}_i \in \mathbb{F}^{\lambda}$ for each $\mathcal{P}_i$ inside $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$. $\boldsymbol{y}_i, \boldsymbol{r}_i$ is opened to $\mathcal{P}_i$.
3. Next, the parties commit to both $\boldsymbol{y}_i, \boldsymbol{r}_i$ using $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta,\zeta}$ 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 *ready*. Once parties receive *ready* from all other parties and are thus synchronized, they simultaneously perform a delayed open of $\boldsymbol{y}_i, \boldsymbol{r}_i$ 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 $\boldsymbol{t}_i$ 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

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

All parties $\mathcal{P}$ have access to one instance of the functionalities $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$, $\mathcal{F}_{\mathsf{ct}}^{\Delta}$ and $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$. Furthermore, each $\mathcal{P}_i \in \mathcal{P}$ has it's own $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta,\zeta,i}$ 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 $(\mathsf{Send}, \mathsf{sid}, c, beat)$ to the functionality $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ (with c as ssid). Throughout $\pi_{\mathsf{mpc,oia}}$, each $\mathcal{P}_i$ waits for $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ to return $(\mathcal{P}_j, c, beat)$ for all other $\mathcal{P}_j \in \mathcal{P}$. If it does, then each $\mathcal{P}_i$ increases c by 1 and sends $(\mathsf{Send}, \mathsf{sid}, c, beat)$ to $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$. Otherwise the parties abort.

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

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

Share:

1. Set $\mathcal{T}_y = \{\mathsf{cid}_{y,j}\}_{j\in[m]}$, $\mathcal{T}_r = \{\mathsf{cid}_{r,k}\}_{k\in[\lambda]}$ and $\mathcal{T}_t = \{\mathsf{cid}_{t,k}\}_{k\in[\lambda]}$.
2. Each $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{ShareOutput}, \mathsf{sid}, \mathcal{T}_y)$ to $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$ and ticks. Then it waits until it obtains a message $\{y_{i,\mathsf{cid}}\}_{\mathsf{cid}\in\mathcal{T}_y}$ with prefix OutputShares from $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$.
3. Each $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{ShareRandom}, \mathsf{sid}, \mathcal{T}_r)$ to $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$ and ticks. It then waits until it obtains a message $\{r_{i,\mathsf{cid}}\}_{\mathsf{cid}\in\mathcal{T}_r}$ with prefix RandomShares from $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$. 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 $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta,\zeta,i}$ and ticks. It then waits for messages $(\mathsf{Commit}, \mathsf{sid}, \mathsf{cid}_j)$ from $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta,\zeta,j}$ 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 $\mathcal{F}_{\mathsf{ct}}^{\Delta}$ 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 $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$.
7. Each $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{Reveal}, \mathsf{sid}, \mathcal{T}_t)$ to $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$ and 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 $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}$. 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.

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, will arrive at each honest party before the honest party's delayed opening result is available to the adversary -

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 $(\mathsf{DOpen}, \mathsf{sid}, \mathsf{cid}_j)$ to $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta,\zeta,j}$ for each $\mathcal{P}_j \in \mathcal{P}$ and ticks. It also stops sending *beat* to $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$.
2. Each $\mathcal{P}_i \in \mathcal{P}$ waits until $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta,\zeta,i}$ returns $(\mathsf{DAdvOpened}, \mathsf{sid}, \mathsf{cid}_i)$. Then $\mathcal{P}_i$ checks if it obtained a message with prefix DOpen from all other $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta,\zeta,j}$. Let $J_1 \subset \mathcal{P}$ be the set of parties such that $\mathcal{P}_i$ did not obtain DOpen before it received $\mathsf{DAdvOpened}$.
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 $\mathcal{F}_{\mathsf{com}}^{\Delta,\delta,\zeta,j}$. 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.

which will be ensured by the appropriate choice of $\zeta > \Delta$. 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. At the same time, honest parties have their DOpen message delivered after Δ steps already and will never be identified as cheaters.

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 $\mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$ 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 [6,4] to achieve their verifiable or output-independent abort in UC. Clearly, our solution can also be used to improve these protocols and to simplify their constructions. The full protocol can be found in Fig. 15 and Fig. 16. We now prove the following Theorem:

Theorem 6. *Let λ be the statistical security parameter and $\zeta > \Delta$. 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 $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta,\zeta}$ in the $\mathcal{F}_{\mathsf{mpc,sso}}^{\Delta}, \mathcal{F}_{\mathsf{com}}^{\Delta,\delta,\zeta}, \mathcal{F}_{\mathsf{ct}}^{\Delta}, \mathcal{F}_{\mathsf{BC,delay}}^{\Gamma,\Delta}$-hybrid model against any static adversary corrupting up to $n-1$ parties in $\mathcal{P}$. The transcripts are statistically indistinguishable.*

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 $\mathcal{F}_{\mathsf{mpc,oia}}^{\Delta,\delta,\zeta} \circ \mathcal{S}$. The proof is presented in the full version[5] due to space limitations.

References

1. M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Fair two-party computations via bitcoin deposits. In *FC 2014 Workshops*, 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*, 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*, Aug. 2017.
4. C. Baum, B. David, and R. Dowsley. Insured MPC: Efficient secure computation with financial penalties. In *FC 2020*, Feb. 2020.
5. C. Baum, B. David, R. Dowsley, R. Kishore, J. B. Nielsen, and S. Oechsner. Craft: Composable randomness beacons and output-independent abort mpc from time. Cryptology ePrint Archive, Paper 2020/784, 2020. https://eprint.iacr.org/2020/784.
6. C. Baum, B. David, R. Dowsley, J. B. Nielsen, and S. Oechsner. TARDIS: A foundation of time-lock puzzles in UC. In *EUROCRYPT 2021, Part III*, Oct. 2021.
7. C. Baum, E. Orsini, and P. Scholl. Efficient secure multiparty computation with identifiable abort. In *TCC 2016-B, Part I*, Oct. / Nov. 2016.
8. C. Baum, E. Orsini, P. Scholl, and E. Soria-Vazquez. Efficient constant-round MPC with identifiable abort and public verifiability. In *CRYPTO 2020, Part II*, Aug. 2020.
9. R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT 2011*, May 2011.
10. I. Bentov and R. Kumaresan. How to use bitcoin to design fair protocols. In *CRYPTO 2014, Part II*, Aug. 2014.
11. N. Bitansky, S. Goldwasser, A. Jain, O. Paneth, V. Vaikuntanathan, and B. Waters. Time-lock puzzles from randomized encodings. In *ITCS 2016*, Jan. 2016.
12. D. Boneh, J. Bonneau, B. Bünz, and B. Fisch. Verifiable delay functions. In *CRYPTO 2018, Part I*, Aug. 2018.
13. D. Boneh and M. Naor. Timed commitments. In *CRYPTO 2000*, Aug. 2000.
14. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, Oct. 2001.
15. R. Canetti, Y. Dodis, R. Pass, and S. Walfish. Universally composable security with global setup. In *TCC 2007*, Feb. 2007.
16. I. Cascudo and B. David. SCRAPE: Scalable randomness attested by public entities. In *ACNS 17*, July 2017.
17. I. Cascudo and B. David. ALBATROSS: Publicly AttestabLe BATched Randomness based On Secret Sharing. In *ASIACRYPT 2020, Part III*, Dec. 2020.
18. R. Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *18th ACM STOC*, May 1986.
19. B. A. Coan, D. Dolev, C. Dwork, and L. J. Stockmeyer. The distributed firing squad problem. *SIAM J. Comput.*, 18(5):990–1012, 1989.
20. G. Couteau, A. W. Roscoe, and P. Y. A. Ryan. Partially-fair computation from timed-release encryption and oblivious transfer. In *ACISP 21*, Dec. 2021.
21. 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*, Apr. / May 2018.

22. L. De Feo, S. Masson, C. Petit, and A. Sanso. Verifiable delay functions from supersingular isogenies and pairings. In *ASIACRYPT 2019, Part I*, Dec. 2019.
23. D. Dolev, R. Reischuk, and H. R. Strong. Early stopping in byzantine agreement. *J. ACM*, 37(4):720–741, 1990.
24. D. Dolev and H. R. Strong. Polynomial algorithms for multiple processor agreement. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, pages 401–407. ACM, 1982.
25. N. Ephraim, C. Freitag, I. Komargodski, and R. Pass. Continuous verifiable delay functions. In *EUROCRYPT 2020, Part III*, May 2020.
26. C. Freitag, I. Komargodski, R. Pass, and N. Sirkin. Non-malleable time-lock puzzles and applications. In *TCC 2021, Part III*, Nov. 2021.
27. J. A. Garay, P. D. MacKenzie, M. Prabhakaran, and K. Yang. Resource fairness and composability of cryptographic protocols. In *TCC 2006*, Mar. 2006.
28. S. D. Gordon and J. Katz. Partial fairness in secure two-party computation. *Journal of Cryptology*, (1), Jan. 2012.
29. C. Hazay, P. Scholl, and E. Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. In *ASIACRYPT 2017, Part I*, Dec. 2017.
30. Y. Ishai, R. Ostrovsky, and V. Zikas. Secure multi-party computation with identifiable abort. In *CRYPTO 2014, Part II*, Aug. 2014.
31. J. Katz, J. Loss, and J. Xu. On the security of time-lock puzzles and timed commitments. In *TCC 2020, Part III*, Nov. 2020.
32. J. Katz, U. Maurer, B. Tackmann, and V. Zikas. Universally composable synchronous computation. In *TCC 2013*, Mar. 2013.
33. A. Kiayias, A. Russell, B. David, and R. Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *CRYPTO 2017, Part I*, Aug. 2017.
34. A. Kiayias, H.-S. Zhou, and V. Zikas. Fair and robust multi-party computation using a global transaction ledger. In *EUROCRYPT 2016, Part II*, May 2016.
35. R. Kumaresan and I. Bentov. How to use bitcoin to incentivize correct computations. In *ACM CCS 2014*, Nov. 2014.
36. E. Kushilevitz, Y. Lindell, and T. Rabin. Information-theoretically secure protocols and security under composition. In *38th ACM STOC*, May 2006.
37. Y. Lindell, A. Lysyanskaya, and T. Rabin. Sequential composition of protocols without simultaneous termination. In A. Ricciardi, editor, *PODC 2002*, 2002.
38. K. Pietrzak. Simple verifiable delay functions. In *ITCS 2019*, Jan. 2019.
39. R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto, 1996.
40. VDF Alliance Team. Vdf alliance, 2020. https://www.vdfalliance.org/what-we-do.
41. B. Wesolowski. Efficient verifiable delay functions. In *EUROCRYPT 2019, Part III*, May 2019.