TARDIS: A Foundation of Time-Lock Puzzles in UC

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

[1] Aarhus University, Denmark
[2] IT University of Copenhagen, Denmark
[3] Monash University, Australia

Abstract. Time-based primitives like time-lock puzzles (TLP) are finding widespread use in practical protocols, partially due to the surge of interest in the blockchain space where TLPs and related primitives are perceived to solve many problems. Unfortunately, the security claims are often shaky or plainly wrong since these primitives are used under composition. One reason is that TLPs are inherently not UC secure and time is tricky to model and use in the UC model. On the other hand, just specifying standalone notions of the intended task, left alone correctly using standalone notions like non-malleable TLPs only, might be hard or impossible for the given task. And even when possible a standalone secure primitive is harder to apply securely in practice afterwards as its behavior under composition is unclear. The ideal solution would be a model of TLPs in the UC framework to allow simple modular proofs. In this paper we provide a foundation for proving composable security of practical protocols using time-lock puzzles and related timed primitives in the UC model. We construct UC-secure TLPs based on random oracles and show that using random oracles is *necessary*. In order to prove security, we provide a simple and abstract way to reason about time in UC protocols. Finally, we demonstrate the usefulness of this foundation by constructing applications that are interesting in their own right, such as UC-secure two-party computation with output-independent abort.

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

⋆⋆ This work was supported by the Concordium Foundation, by Protocol Labs grant S^2LEDGE and by the Independent Research Fund Denmark with grants number 9040-00399B (TrA^{2}C) and number 9131-00075B (PUMA).

⋆⋆⋆ This work was partially done while Rafael Dowsley was with Bar-Ilan University and was supported by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office.

† Partially funded by The Concordium Foundation; The Danish Independent Research Council under Grant-ID DFF-8021-00366B (BETHE); The Carlsberg Foundation under the Semper Ardens Research Project CF18-112 (BCM).

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

1 Introduction

The Universal Composability (UC) framework [18] is widely used for formally analyzing cryptographic protocols as it provides strong security guarantees that allow UC-secure protocols to be arbitrarily composed. This is a very useful property and enables the modular design of cryptographic protocols. However, the original UC framework is inherently asynchronous and does not support the notion of time. Katz et al. [35] introduced a clock functionality in order to define universally composable synchronous computation. Their clock functionality captures the essence of synchronized wall clocks that are available to all parties. This notion is particularly useful in reasoning about synchronous protocols in the UC framework, since the honest parties can use the clock to achieve synchronization.

However, many cryptographic protocols do not depend on concrete time provided by a wall clock, but just on the relative order of events, such as the arrival of messages or the completion of some computation. In particular, protocols in a semi-synchronous communication model (*e.g.* [24,5]) rely on the fact that there exists a finite (but unknown) upper bound for the delay in communication channels, without requiring that events (*e.g.* the arrival of a message) occur at a specific wall clock time (or even within a concrete delay) as long as they occur in a certain order. In this case, using a clock can make the design and security analysis of such protocols unnecessarily complicated.

Another important challenge lies in modeling sequential computation and computational delays in the UC framework. Since the environment may operate in many parallel sessions and activate parties arbitrarily, it obtains an unfair computational advantage in relation to the parties. For example, even if its computational power is constrained within a session, the environment can use multiple sessions to solve faster than a regular party a computational problem assumed to require at least a certain amount of computational steps (and thus time). This precludes the UC modeling and construction of primitives based on sequential computation and computational delays, such as time-lock puzzles [44].

1.1 Our Contributions

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

- **Abstract Time in UC:** we put forth a novel abstract notion of time for the UC framework capturing relative event ordering without a clock.
- **Impossibility of UC-Secure TLPs without Programmable Random Oracles:** we prove that programmable random oracles are necessary for constructing UC-secure TLPs, yielding a new separation between programmable and non-programmable random oracles.

- **First Composable Treatment of Time-Lock Puzzles (TLPs):** we introduce the first composable definition and construction of time-lock puzzles. Our construction uses a RO, as it must. However, it has a flavor of "graceful degradation": if the hash function is not modeled as a random oracle, our TLPs are still non-malleable, which is in some sense optimal without a RO.
- **First Two-Party Computation Protocol with Output Independent Abort:** we use TLPs to construct a UC-secure two-party computation protocol where the adversary cannot see the output before deciding to abort.

The incompatibility of time-lock puzzles and UC security is easy to explain. All that is needed is to recall that UC has straight-line simulation. Let $P = \mathsf{TLP}(x, T, t)$ be a timed commitment to x which can be opened in time T and is hiding for time $t < T$. Consider simultaneous message exchange. In the UC functionality Alice inputs a, Bob inputs b and only then are both given (a, b). Here is a toy protocol which does not work for many reasons. Alice and Bob each publish $P_A = \mathsf{TLP}(a, T, t)$ and $P_B = \mathsf{TLP}(b, T, t)$ and then open the puzzles or brute force them. Assume that Alice is supposed to send her puzzle first, and Bob is corrupted. In the security proof, the UC simulator needs to extract Bob's input b in order to query the ideal functionality and learn a. However, the simulator needs to learn P_B for that. P_B though is only sent by Bob *after* seeing P_A. As a result, the simulator has to produce P_A without knowing a. Rewinding is not allowed, so the simulator cannot go back and replace the puzzle P_A. The simulator had to put some a' inside P_A and is now committed to it. If a is random then with noticeable probability $a' \neq a$. In UC these problems are typically handled by having trapdoors which allow to do equivocation. Had P_A been a UC commitment we could have changed a' to a before opening. But P_A is a TLP, so there is no way to cheat. The UC environment can simply take P_A and brute-force it open. So the a' is irrevocably committed to by P_A. A shorter way to explain the problem is as follows. In a UC simulation the simulator must for all puzzles it sends agree on what is inside at the point in time where they were sent. And the UC experiment will keep running as long as the environment wants, so it can allow itself time enough to open all puzzles eventually. Hence puzzles will not afford us the power of a UC commitment which can be equivocated. Unfortunately, equivocation is exactly the power needed for simulating time-lock puzzles in the UC framework for most interesting applications.

Although the above argument only shows that one particular protocol does not work, we show that the problem cannot be circumvented by any protocol even allowing setup like a CRS. Assuming a random oracle, however, one can cheat and use the random oracle to get equivocation. Note the if we model H as a random oracle and send $\mathsf{TLP}(r), H(r) \oplus m$ in a simulation, we can reprogram H at r as long as $\mathsf{TLP}(r)$ is hiding r such that H was not queried at r. This is of course an unsatisfactory solution, but some comfort can be gained from the fact that we provably cannot do without such a cheat if we want UC security.

There is a clear need for a UC model of time-lock puzzles and other time-based primitives, since those are finding widespread use in complex scenarios like the widely distributed and concurrent blockchain setting, where there is no way

around having composable security for the protocol building blocks. Many of the proposed uses are often relatively simple *a la* the above simultaneous message exchange example. However, the security statements are often provably wrong, as TLPs for instance cannot yield composable simultaneous message exchange. Reverting to non-composable game-based definitions of the intended tasks and using non-malleable TLPs for the standard model is in principle a solution, but the proofs are typically complicated and the protocols inefficient. We therefore introduce a foundation for practical TLP-based protocols using a UC model of TLPs that allows simple analysis of practical protocols. This model is motivated in the same vein as the random oracle model, which was also proposed as a basis for analysing efficient, practical protocols.

Clearly, when using TLPs we also need a notion of time. If a TLP that can be broken in an hour is received through a network, it should not be trusted to be hiding an hour later. That requires a notion of time (*e.g.* a clock). Often the reliance on time in practical protocols using TLPs if fairly light. In line with the motivation above, we therefore provide also a simple abstract notion of time.

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

Building on this model, we introduce the first definition and construction of UC-secure time-lock puzzles. Our construction is based on the classical time-lock assumption of Rivest *et al.* [44] and uses a restricted programmable and observable global random oracle, which we prove to be *necessary.* As an application of our composable TLPs, we introduce the notion of two-party computation (2PC) with output independent abort (OIA) along with the first OIA-2PC protocol. This new security notion for secure computation guarantees that an adversary who aborts the execution cannot learn any information about the output *before* deciding to abort, only obtaining the output after this decision is made. Our new definition improves on the standard security notion with abort (realized by all known 2PC protocols), which allows for the adversary to decide whether to force the honest parties to abort without obtaining the output *after* learning the output itself. We argue that this new security notion is optimal, since fairness (*i.e.* ensuring all parties obtain the output if the adversary does so) for 2PC protocols is impossible [21].

1.2 Related Work.

Composition frameworks with time and fairness. Composition frameworks for cryptographic protocols (e.g. UC [18], constructive cryptography [39], the reactive simulatability (RSIM) framework [42]) provide strong security guarantees for protocols under concurrent composition. In all mentioned frameworks, communication is through inherently *asynchronous* channels. Several works have

therefore studied general composition guarantees with *synchronous* communication by introducing a shared source of time or restricting adversarial scheduling. Modeling network timing assumptions such as bounded message delay and clock drift and the resulting concurrent composition guarantees for specific tasks was studied for zero-knowledge [25], [29] and MPC [33]. In the context of composition frameworks, Backes et al. [4] model traffic-related timing attacks in GNUC [31] by allowing the adversary to measure the local time at which a message arrives. In this setting, each party has a local execution time, and the EXEC function of GNUC maps the local times into a global time. Backes et al. [3] studied fairness in the RSIM framework and achieve a composable notion of fairness by restricting the adversary model to fair schedulers who deliver any message after at most a polynomial number of steps.

The work that is most closely related to ours is that of Kiayias *et al.* [36], which points out limitations of the local clock functionality of Katz *et al.* [35] and adapts it to the Global UC (GUC) framework [19] to provide all parties with access to a global clock functionality for the purpose of synchronization. Their model requires all parties executing a (semi-)synchronous protocol to keep track of current global clock time and to actively query the global clock functionality in order to advance of time. In particular, even if their model is used to define semi-synchronous communication, it implies that all parties are kept synchronized and may learn how much time has elapsed since their last activation (*i.e.* by obtaining the current time from the global clock), which is a rather strong synchrony assumption. However, many protocols cast in this model do not crucially rely on obtaining concrete time stamps or determining concrete delays between party activations, as long as messages are guaranteed to be delivered within certain delays and in a certain order (*e.g.* as in [5]). This is exactly the kind of guarantees that our model captures without explicitly exposing time keeping to parties or requiring them to keep track of concrete time sources. By doing that, our model allows us to analyse many protocols cast in their model while significantly relaxing synchrony assumptions. Moreover, our model can be used to capture delays induced by sequential computation, which is not captured by the global clock model of Kiayias *et al.*.

Another work technically related to ours is the notion of resource-fairness for protocols in UC introduced by Garay *et al.* [28]. Resource fairness ensures that honest parties who invest a certain amount of resources (*e.g.* computational time) can always recover from an abort and obtain the protocol output in case the adversary causes an abort in such a way that it learns the output. In order to realize this notion, Garay *et al.* show a generic compiler based on a “time-line” construction and a secure computation functionality. Essentially, this time-line encodes a number of computational states into a programmable common reference, which parties use in order to commit to messages that can be recovered by another party who invests enough computational steps. This idea differs from our work in that it limits TLP delay a priori, since the maximum number of computational states used to ensure delay is fixed by the CRS. This crucial difference also forces the resource-fairness framework to modify the UC framework

in such a way that environments, adversaries and simulators must have an a priori bounded running time. On the other hand, our modelling of computational time and TLPs does not make modifications or restrictions to the UC environment, as well as allowing us to define TLPs in a more natural way where there's no a priori bound to the TLP delay. In particular, this means that TLPs can be parameterized with any arbitrary delay and that honest parties are always able to solve a TLP, which also allows us to realize our notion of output independent abort in such a way that honest parties can always either retrieve the output of the computation or determine that the adversary has aborted (by solving the adversary's TLPs).

Another relation to [28] is that both papers circumvent the problem that TLPs are not UC simulatable. We do it using the simple hack of using a random oracle, to get a simple model to work with. In [28] it is done by letting the simulator depend on the running time of the environment.

As an example of how to exploit this consider a party that wants to commit to s. It secret shares it into $(s_1, \ldots, s_k)$ and makes public $P_1 = \mathsf{TLP}(s_1), \ldots, P_k = \mathsf{TLP}(s_k)$. The hardness of P_i is set to 2^i and k is the security parameter. So P_k cannot be brute-forced open. For each s_i it also gives a UC commitment to s_i and a ZK proof that the commitment is to the value in the TLP. To do fair message exchange both Alice and Bob do the above with $s = a$ and $s = b$. Then the parties open the commitments (not the TLPs) to the shares in the order $s_k, s_{k-1}, \ldots$, taking turns to reveal a share. If a party stops opening commitments, then use the TLPs to learn the remaining shares, if the hardness of the remaining TLPs is feasible. Now in the simulation, if the running time of the environment is upper bounded to some polynomial t, then there exists $i_0 < k$ such that $2^{i_0} > t$. Now the simulator can put dummy shares s_i' in P_j for $j \geq i$. When it learns the message a of Alice it can then adjust the UC commitments to be a secret sharing of a. It does not have to adjust the TLPs as the environment will not have time to open them. The fact that there is an "end of time" in the simulation allows to simulate some TLPs. On the other hand, the fact that there is an "end of time" in the simulation makes composition cumbersome. Indeed [28] gets a complicated notion of security where a protocol to be called secure must be secure in two ways. It must be secure in a so-called resource game, and it must also be so-called full simulation secure. This requires [28] to develop a new variant of the UC framework. This variant does not imply security in the normal UC model which does not have an "end of time".

It also seems hard to prove security of most practical protocols in [28]. The reason is that it seems hard to exploit the simulator's knowledge of the running time of the environment (which can be any polynomial) without using TLPs of super-polynomial running time, as in the above examples with TLPs of doubling hardness. This seems to make it hard to prove security of many simple and intuitively secure scheme like the first protocol above with two TLPs for simultaneous message exchange. Either these two TLPs have a hardness set such that real-world parties can brute-force them (and then so can the environment) or it is set so hard that the environment cannot brute-force them, then neither can

the parties. In the first case the protocols falls prey to our impossibility result. In the later case the TLPs seems useless.

We find the techniques and models in our paper and [28] complementary. Our model is built on the normal (G)UC model without modifying it and is simple to specify and use. But needs a random oracle. The paper [28] shows that even without random oracles not everything is lost. It is possible to get models and some constructions with UC like security.

Time-Lock Puzzles and Computational Delay The original construction of time-lock puzzles was proposed by Rivest, Shamir and Wagner [44]. Boneh and Naor [15] introduced the notion of timed commitments. An alternative construction of time-lock puzzles was presented by Bitansky et al. [13]. Recently, the related notion of verifiable delay functions has been investigated [14,43,48]. These constructions are closely related in that they rely on sequential computational tasks that force parties to spend a certain amount of time before they are able to obtain an output. However, none of these works have considered composability issues for such time-based primitives. In particular, the issues of malleability for these time-based primitives and the relationship between computational and communication delay are notably ignored in previous works. The lack of composabillity guarantees for time-lock puzzles is a significant shortcoming, since these primitives are mostly used as building blocks for more complex protocols and current constructions do not ensure that their security guarantees are retained when composed with other primitives to obtain such protocols. Our composable treatment of time-lock puzzles addresses theses issues by introducing constructions that can be arbitrarily composed along with a framework for analysing complex protocols whose security relies on the relative delays in computation and communication.

Concurrently to us, Katz et al. [34] as well as Ephraim et al. [26] have constructed Non-Malleable Timed Commitments. Among others, [23] have shown that UC (non-timed) Commitments imply Non-Malleable Commitments. A similar argument can be made for timed commitments as well. In that sense, our construction of UC-secure TLPs implies [34,26]. At the same time, our work crucially relies on a programmable Random Oracle (and indeed shows that it is necessary to achieve UC security). Neither [34] nor [26] require such strong assumptions and can be seen as realising the strongest notion of non-malleability achievable without using a (programable) random oracle or similar assumption: the beautiful construction of [34] does not require any Random Oracle-type assumption and builds upon RSW-TLPs in the Algebraic Group Model, while [26] use an observable Random Oracle but their construction can be realized from a generic TLP. At the same time, [26] also constructs publicly verifiable TLPs departing from generic strong trapdoor VDFs. We'd like to stress that our construction of a UC-secure TLP is publicly verifiable, although this is only shown in recent follow-up work [7]. We further note that the work of [26] shows a bound on the composability of non-malleable TLPs, but their bound does not apply to our setting as they assume a distinguisher with an arbitrary runtime, while the UC environment is computationally restricted in our setting.

Aborts and Fairness in Secure Computation An MPC protocol is said to be fair if a party can obtain the output if and only if all other parties also obtain the output. It is a well-known fact that fair MPC in the standard communication model is impossible with a dishonest majority [21]. Given the impossibility to achieve fairness, techniques for identifying misbehaving parties responsible for causing an abort have been investigated [32,9,10]. In the last few years a line of work developed which imposes financial penalties on parties who are identified as misbehaving by using cryptocurrencies and smart contracts, thus giving financial incentives for rational parties to behave in a fair way. Protocols have been designed to punish misbehavior at any point of the protocol execution (Fair Computation with Penalties) [2,38,36] or to only punish participants that learn the output but prevent others from doing the same (Fair Output Delivery with Penalties) [1,12,37,6]. However, these protocols allow the adversary to make a decision on whether to abort or not *after* seeing the output that will be obtained by the honest parties in case the execution proceeds.

The recent work of Couteau et al. [22] studies the problem of obtaining partially-fair exchange from time-lock puzzles, but in much weaker security and adversarial models. In particular, their work does not consider composability issues and is limited to the specific problem of fair exchange rather than the general problem of secure computation considered in our results.

Random oracle separation results. Our impossibility result provides yet another separation between the programmable and non-programmable random oracle models, complementing the few previously known separations [40,47,27,11].

1.3 Our Techniques

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

Abstract Time: Our goal is to express different timing assumptions (possibly related) within the GUC framework in such a way that protocols are oblivious to them. We do so by providing the adversary with a way of advancing time in the form of *ticks*. A tick represents a discrete unit of time. Time can only be advanced, and moreover only one unit at a time. In contrast to Katz et al. [35,36], however, these ticks and thus the passing of time are not supposed to be directly visible to the protocol. Thus instead of a (global) clock that parties can ask for the current time, we add a ticking interface to ideal functionalities. This way, timing-related observable behavior becomes an assumption of the underlying functionalities, e.g. of a computational problem or a channel. Parties may now observe events triggered by elapsed time, but not the time itself. Ticked functionalities are free to interpret ticks in any way they like; this way we can synchronize and relate ticks representing elapsed time in different "units" like passed time or computation steps. The technical challenge is to ensure in a composable way that all honest parties have a chance at observing all relevant timing-related events. Katz et al. solved this issue inside the clock by keeping

track of which parties have been activated in the current time period (and thus asked for the time) and refusing to advance time if necessary. We enforce the requirement that all honest parties must be activated between ticks by defining a global ticker functionality that makes sure this constraint is obeyed. In contrast to the global clock, our global ticker does not provide any information about the time elapsed between queries by functionalities, only informing functionalities that a new tick has ocurred. From the point of view of honest parties, our global ticker is even more restricted, since it does not inform parties whether a tick has occurred or not. To further control the observable side effects of ticks, we restrict protocols and ideal functionalities to interact in the "pull model" known from Constructive Cryptography, precluding functionalities from implicitly providing communication channels between parties and instead requiring parties to actively query functionalities in order to obtain new messages. Apart from presenting a clear abstraction of time, this notion explicitly exposes issues that must be taken in consideration when implementing protocols that realize our functionalities, *i.e.* the concrete delays in real world communication channels and computation. In particular, while the theoretical protocol description and security analysis can be carried in terms of such abstract delays, our techniques clarify the relationship between concrete time-based parameters (*e.g.* timeouts vs. network delays) that must be respected in protocol implementations. We will go into this in more detail in Section 2.

Composable Treatment of Time-Lock Puzzles: To illustrate the potential uses of our framework, we present the first definition and construction of UC-secure Time-Lock Puzzles (TLP). We depart from the classical construction by Rivest et al. [44] and provide the first UC abstraction of the Time-Lock Assumption, which is modeled in a "generic group model" style, hiding the group description from the environment and limiting its access to group operations. A party acting as the "owner" of an instance of the TLP functionality can generate a puzzle containing a certain message that should be revealed after a certain number of computational steps. The functionality allows the parties to make progress on the solution of the puzzle every time that it is ticked. Once a party solves a puzzle, it can check that a certain message was contained in that puzzle. The ticks given to this functionality come externally from the adversary and we require in the framework that the parties get activated often enough. We show that our UC abstraction of the Time-Lock Assumption allows us to implement UC-secure TLPs in the restricted programmable and observable global random oracle model of Camenisch et al. [16] (which turns out to be necessary for UC-realizing TLPs). We introduce our notion of TLP in the UC model with a global ticker in Section 4 and our construction of a protocol realizing this notion in Section 5.

Two-Party Computation with Output Independent Abort: To further showcase our framework we construct the first protocol for secure two-party computation (2PC) with output-independent abort, i.e., the adversary must decide whether to abort or not before seeing the output. In order to do so, we build on techniques

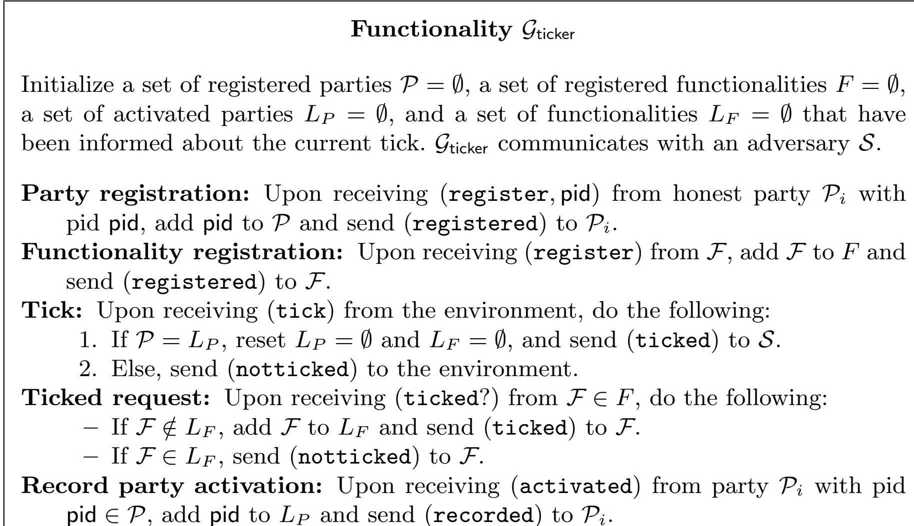
from [6]: there, the authors combine an MPC protocol with linearly secret-shared outputs and an additively homomorphic commitment by having each party commit to its share of the output and then reconstruct the output inside the commitments. In [6], the output of the secure computation is obtained by opening the final commitments resulting from the reconstruction procedure, which allows the adversary to learn the output before the honest parties do. He can then refuse to open its commitment, causing the protocol to abort, based on this information. Similarly to [6], we combine a 2PC protocol with secret-shared outputs and an additively homomorphic commitment, but we define and construct commitments with a new delayed opening interface. When a delayed opening happens, the receiver is notified after a communication delay but only receives the revealed message after an *opening delay.* Hence, we can obtain output independent abort by delayed opening the final commitments obtained after reconstructing the output and considering that a party aborts if it does not execute a delayed opening of their commitments before the other parties delayed openings reveal their messages. Finally, we show how to obtain UC-secure additively homomorphic commitments with delayed opening by modifying the scheme of Cascudo et al. [20] with the help of the delayed secure message transmission and TLP functionalities. We present these results in Section 6.

Impossibility Result. Finally, we prove that a non-programmable random oracle is not sufficient for obtaining UC-secure fair-coin flip, secure 2PC with output-independent abort or TLP. Therefore a programmable random oracle is necessary to implement these primitives, yielding a separation between the programmable and non-programmable random oracle models. This also shows that our TLP construction which requires this strong assumption is in that sense "optimal". We present this impossibility result in Section 7.

2 UC with Relative Time

This section describes our notion of abstract time. In order to obtain universal composability, we model our ideas on top of the GUC framework [19]. The goal is to capture time in such a way that parties are oblivious to it and can only observe the progression of time indirectly through events like the arrival of messages or the completion of a computation. At the same time, the passing of abstract time is completely under adversarial control. And most importantly, the notion is meant to be composable.

Timing assumptions. Our first observation is that timing assumptions are assumptions about physical systems and should thus be captured at the level of ideal functionalities. Such a timed functionality has a notion of passing time and can adapt its behavior as time progresses. This will allow us to reduce properties of a protocol that require concrete timing assumptions. Note that the time is only a proof construct, it is not visible to the actual protocol, much like physical time. Most importantly, having a notion of passing time should not imply synchrony like in the UC clock models of [36,35].



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

Initialize a set of registered parties $\mathcal{P} = \emptyset$, a set of registered functionalities $F = \emptyset$, a set of activated parties $L_P = \emptyset$, and a set of functionalities $L_F = \emptyset$ that have been informed about the current tick. $\mathcal{G}_{\mathsf{ticker}}$ communicates with an adversary $\mathcal{S}$.

Party registration: Upon receiving ($\texttt{register}, \mathsf{pid}$) from honest party $\mathcal{P}_i$ with pid pid, add pid to $\mathcal{P}$ and send ($\texttt{registered}$) to $\mathcal{P}_i$.
Functionality registration: Upon receiving ($\texttt{register}$) from $\mathcal{F}$, add $\mathcal{F}$ to F and send ($\texttt{registered}$) to $\mathcal{F}$.
Tick: Upon receiving ($\texttt{tick}$) from the environment, do the following:

1. If $\mathcal{P} = L_P$, reset $L_P = \emptyset$ and $L_F = \emptyset$, and send ($\texttt{ticked}$) to $\mathcal{S}$.
2. Else, send ($\texttt{notticked}$) to the environment.

Ticked request: Upon receiving ($\texttt{ticked?}$) from $\mathcal{F} \in F$, do the following:

- If $\mathcal{F} \notin L_F$, add $\mathcal{F}$ to L_F and send ($\texttt{ticked}$) to $\mathcal{F}$.
- If $\mathcal{F} \in L_F$, send ($\texttt{notticked}$) to $\mathcal{F}$.

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

Fig. 1. Global ticker functionality $\mathcal{G}_{\mathsf{ticker}}$.

Global ticker functionality $\mathcal{G}_{\mathsf{ticker}}$. This idea leads to natural questions. Where does this "time" come from? And if there are multiple timed functionalities, how is it coordinated between them to support the kind of reductions we want? The first question can be answered by the well-known concept of adversarial "ticks" that model discrete units of passing time. To answer the second question, we propose a global ticker functionality $\mathcal{G}_{\mathsf{ticker}}$ that receives ticks from the environment and makes them available for ticked functionalities upon request. Parties themselves have no access to the ticker.

Note that $\mathcal{G}_{\mathsf{ticker}}$ captures an assumption on the physical world and can therefore not be instantiated. It is only a tool for proofs. Similar to the synchronous setting with a global clock where the next logical round can only start after all parties have been activated, the global ticker implicitly enforces that all honest parties can finish their computations for the current tick before advancing to the next tick. This ensures all honest parties are activated and given a chance to perform computation without the need to modify the (G)UC framework. Notice that, while the assumption of a global time is a poor model of reality, we do not envision our model being used for protocols running in relativistic conditions.

While $\mathcal{G}_{\mathsf{ticker}}$ allows the ideal adversary to take actions as soon as every tick happens, it gives no information about passing time to the honest parties. The only interaction that honest parties have with $\mathcal{G}_{\mathsf{ticker}}$ is in confirming that they have been activated. A new tick only happens once all honest parties confirm they have been activated after the last tick. This mechanism ensures that the environment or the adversary do not get an unfair advantage in accessing timed functionalities while preventing the honest parties from also doing so (since in this case the honest parties will not confirm they have been activated and time will not progress).

Only other functionalities (and the ideal adversary) can detect elapsed time by querying $\mathcal{G}_{\mathsf{ticker}}$ and receiving a notification in case a new tick happened. In our model, functionalities take actions such as delivering a message or the output of a computation once a new tick happens. Hence, honest parties only perceive time through messages received by other functionalities that have their behavior conditioned on the progression of time. In particular, if one wants to instantiate synchronized clocks from $\mathcal{G}_{\mathsf{ticker}}$, it would be possible to instantiate a version of the UC clocks of [36,35] where the clock only progresses when a new tick is issued by $\mathcal{G}_{\mathsf{ticker}}$. With such a construction, parties can access the number of ticks issued up to a certain point of the execution by querying the clock functionality (but not $\mathcal{G}_{\mathsf{ticker}}$). Note that in this setting, honest parties need to query the clock functionality regularly to ensure that the clock can in turn query $\mathcal{G}_{\mathsf{ticker}}$ for ticks.

Conventions. For the sake of readability, we will omit the calls of ideal functionalities to $\mathcal{G}_{\mathsf{ticker}}$ which would in the worst case have to occur at every activation. Functionalities are instead assumed to query $\mathcal{G}_{\mathsf{ticker}}$ with (`tick`?) whenever they are activated, and the behavior upon a positive answer is described as **Tick** in the ideal functionality description.

3 Communication Delay

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

3.1 Secure Message Transmission with Delays

Secure message transmission (SMT) is the problem of securely sending a single message m from a sender $\mathcal{P}_S$ to a receiver $\mathcal{P}_R$. Secure means that the power of an eavesdropper intercepting the channel is restricted to learning some leakage $\ell(m)$ on the message and delaying the message delivery. The standard formulation of $\mathcal{F}^{\ell}_{\mathsf{smt}}$ [17, 2019 version] assumes that message delivery can be delayed infinitely by an adversary. Here, we want to add an upper bound on the message delay. The exact constraints on this upper bound will determine whether a protocol operates over synchronous, semi-synchronous or asynchronous channels, as discussed further in Section 3.2

In order to capture elapsed time according to our model, we add a **Tick** procedure to obtain a ticked ideal functionality. As mentioned in the previous section, **Tick** is run upon each activation if $\mathcal{G}_{\mathsf{ticker}}$ indicates that a new tick happened. The functionality is parameterized by a maximal delay $\Delta > 0$. Requiring $\Delta > 0$ models the fact that communication always takes time. After a message

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

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

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

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

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

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

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

Fig. 2. Ticked ideal functionality $\mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$ for secure message transmission with maximal message delay Δ.

is input to the functionality by the sender, each tick will increase a counter. The message is released to the receiver after at most Δ ticks are counted or whenever the ideal adversary instructs the functionality to release it.[4] However, a tick cannot directly trigger the activation of parties other than the adversary. Otherwise, we would be exposing the elapsed time towards the parties and implicitly synchronizing them. As a consequence, the functionality cannot send the message to the receiver as in [17]. We solve this issue by requiring the receiver to actively query the functionality for newly released messages. Finally, the adversary can adaptively request to corrupt a party $\mathcal{P} \in \{\mathcal{P}_S, \mathcal{P}_R\}$, in which case they will learn the message if the corresponding party knows it already. Note that this corruption behavior differs crucially from Canetti's formulation: Since

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

message transmission is explicitly taking time, adaptive corruptions at runtime are actually meaningful now. In particular, it is no longer possible to first observe leakage on a sent message to then corrupt the sender and change the message that was sent. The resulting ideal functionality $\mathcal{F}^{\Delta}_{\mathsf{smt,delay}}$ is shown in Fig. 2.

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

3.2 Modeling (Semi)-Synchronous Channels

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

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

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

4 Modeling Time-Lock Puzzles and Computational Delay

We will now introduce a concept for modeling sequential computation inside the UC framework that does not suffer from degradation through composition

or adversarially chosen activation of parties. As an example, we will realize the notion of a "time-lock puzzle" [44] in a composable fashion. In a time-lock puzzle (TLP), the owner generates a computational puzzle that outputs a message to the receiver when solved. The main property of the construction is that none of the solvers can obtain the message from the puzzle substantially faster than any other solvers, thus introducing problems that cannot be parallelized.

To the best of our knowledge, this has not been formalized in the UC framework before and there are multiple pitfalls that one has to avoid when formalizing TLPs. First, UC allows the environment to activate parties at its will throughout the session and it might be that an honest party does not even get activated before the puzzle was solved by the adversary. Even worse, such a modeling might permit that the environment can solve the puzzle in another session, so even by enforcing regular activation inside a session (as in the previous section) or equal computational powers between the iTM modeling the parties as well as the adversary one cannot achieve the aforementioned notion.

Ticked ideal functionalities help us to overcome both issues, and the resulting ticked time lock puzzle ideal functionality $\mathcal{F}_{\mathsf{tlp}}$ is shown in Fig. 3. It can easily be seen that the functionality fulfills our definitions as outlined before. First, any new instance of a puzzle can be tied to a specific party, namely the owner $\mathcal{P}_o$, who can initialize the puzzle by providing a number of computation steps Γ and a message m. The functionality outputs a puzzle $\mathtt{puz} = (\mathtt{st}_0, \Gamma, \mathtt{tag})$ consisting of an initial state $\mathtt{st}_0$, the number of steps Γ needed for reaching a final state and tag $\mathtt{tag}$ used to encode the message. After every tick, each party can use a puzzle state $\mathtt{st}_i$ to call the **Solve** interface, which will append the next state $\mathtt{st}_{i+1}$ to a list of messages delivered to the party after the next tick. By buffering messages containing the next states, we essentially limit all parties' (and the environment's and adversary's) ability to attempt performing more than one solving step per tick and puzzle. Notice that any party who tries to call **Solve** more than once per tick for a puzzle would have to guess the next state $\mathtt{st}_{i+1}$ in order to perform the second call, which can only be done with negligible probability. Once the final state $\mathtt{st}_\Gamma$ is reached, parties can call the **Get Message** interface in order to retrieve the message associated with the puzzle by presenting the puzzle $\mathtt{puz}$ and the final state $\mathtt{st}_\Gamma$ obtained through successive calls to **Solve**. Finally, $\mathcal{F}_{\mathsf{tlp}}$ will at the beginning of any activation query $\mathcal{G}_{\mathsf{ticker}}$ if a clock-tick happened and execute the Tick procedure if it indeed did. This will allow each party to obtain a new value, which may get it closer to the solution of the puzzle.

Observe that this model does neither restrict the actual computational power of the environment nor any other iTM. The environment can activate any party arbitrarily often, as long as the honest parties also occasionally can have the ability to access the restricted resource. Care must also be taken to allow limited ideal adversarial control over the functionality's answers to queries to **Solve** containing undefined states and queries to **Get Message** containing undefined $(\mathtt{puz}, \mathtt{st})$ tuples. While the adversary is allowed to provide an arbitrary sequence of states $\mathtt{st}_0, \ldots, \mathtt{st}_\Gamma$ and an arbitrary tag $\mathtt{tag}$, the functionality enforces the fact

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 3. Functionality $\mathcal{F}_{\mathsf{tlp}}$ for time-lock puzzles.

that, once defined, the same sequence of steps will be deterministically obtained by all honest parties invoking **Solve**. However, queries to $\mathcal{F}_{\mathsf{tlp}}$ involving undefined states and puzzles are answered with messages provided by the ideal adversary. This is necessary for capturing adversaries that construct different versions of a puzzle departing from different initial states of the original sequence $\mathtt{st}_0, \dots, \mathtt{st}_\varGamma$ or from an arbitrary state that eventually leads to this sequence.

5 Constructing Time-Lock Puzzles in UC

The functionality given in Fig. 3 from Section 4 describes how we ideally model a TLP in our framework. We will now instantiate $\mathcal{F}_{\mathsf{tlp}}$ departing from the well-known construction by Rivest et al. [44]. In order to obtain a UC-secure protocol,

we will first model the assumption that underpins Rivest *et al.*'s construction under our notion of sequential computation with ticks. Moreover, we will resort to a global random oracle, which turns out to be *necessary* for UC-realizing $\mathcal{F}_{\mathsf{tlp}}$ as discussed later in this section.

The TLP construction of Rivest *et al.* [44] is based on the assumption that it is hard to compute successive squarings of an element of $(\mathbb{Z}/N\mathbb{Z})^{\times}$ (*i.e.* the group of primitive residues modulo N) with a large N in less time than it takes to compute each of the squarings sequentially, unless the factorization of N is known. In other words, for a random element $g \xleftarrow{\$} (\mathbb{Z}/N\mathbb{Z})^{\times}$ and a large N whose factorization is unknown, this assumptions says that it is hard to compute $g^{2^{\Gamma}}$ in less time than it takes to compute Γ sequential squarings $g^2, g^{2^2}, g^{2^3}, \ldots, g^{2^{\Gamma}}$. On the other hand, if $N = pq$ is generated following the key generation algorithm of the RSA cryptosystem, one obtains a trapdoor (*i.e.* the order of $(\mathbb{Z}/N\mathbb{Z})^{\times}$) $\phi(N) = (p-1)(q-1)$ that allows for fast computation of $g^{2^{\Gamma}}$ requiring two exponentiations: first compute $t = 2^{\Gamma} \mod \phi(N)$ and then g^t. Hence, a TLP encoding a message $m \in (\mathbb{Z}/N\mathbb{Z})^{\times}$ with a number of steps Γ can be generated by a party who knows $N = pq, p, q$ by sampling a random $g \xleftarrow{\$} (\mathbb{Z}/N\mathbb{Z})^{\times}$, computing $t = 2^{\Gamma} \mod \phi(N)$, $g^{2^{\Gamma}} = g^t$ and $mg^{2^{\Gamma}}$, arriving at a puzzle $\mathtt{puz} = (g, \Gamma, mg^{2^{\Gamma}})$. From the assumption of Rivest *et al.*, it follows that any party must compute Γ sequential squarings departing from g in order to obtain $g^{2^{\Gamma}}$ and compute $m = mg^{2^{\Gamma}} g^{-2^{\Gamma}}$.

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

5.1 Modeling Rivest *et al.*'s Time-Lock Assumption [44]

We describe in Fig. 4 an ideal functionality $\mathcal{F}_{\mathsf{rsw}}$ that captures the hardness assumption used by Rivest et al. [44] to build a time-lock puzzle protocol. This functionality essentially treats group $(\mathbb{Z}/N\mathbb{Z})^{\times}$ as in the generic group model [46] and only gives handles to the group elements to all parties. In order to perform operations, the parties then need to interact with the functionality. They can ask for any number of operations to be performed between two computational ticks. However, the outcome of the operation (*i.e.* the handle of the resulting group element) will only be released after the next computational tick occurs. However, a special owner party $\mathcal{P}_o$ who initializes $\mathcal{F}_{\mathsf{rsw}}$ receives a trapdoor $\mathtt{td}$ that allows it to perform arbitrary operations on group elements. Upon learning $\mathtt{td}$ any

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

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

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

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

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

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

Multiply: Upon receiving $(\mathsf{Mult}, \mathsf{sid}, \mathtt{el}_1, \mathtt{el}_2)$ from $\mathcal{P}_i \in \mathcal{P}$:
1. If there are no a, b s.t. $(\mathtt{el}_1, a), (\mathtt{el}_2, b) \in \mathsf{group}$, then output $(\mathsf{Invalid}, \mathsf{sid})$ to $\mathcal{P}_i$.
2. Compute $c \leftarrow ab \bmod N$. If there is no $(\mathtt{el}_3, c) \in \mathsf{group}$ then sample $\mathtt{el}_3 \xleftarrow{\$} \{0,1\}^\tau$ randomly but different from all group entries and add $(\mathtt{el}_3, c)$ to group.
3. Add $(\mathcal{P}_i, (\mathtt{el}_1, \mathtt{el}_2, \mathtt{el}_3))$ to in and return $(\mathsf{Mult}, \mathsf{sid}, \mathtt{el}_1, \mathtt{el}_2)$ to $\mathcal{P}_i$.

Invert: Upon receiving $(\mathsf{Inv}, \mathsf{sid}, \mathtt{el})$ from some party $\mathcal{P}_i \in \mathcal{P}$:
1. If there is no a such that $(\mathtt{el}, a) \in \mathsf{group}$ then output $(\mathsf{Invalid}, \mathsf{sid})$ to $\mathcal{P}_i$.
2. Compute $y \leftarrow a^{-1} \bmod N$. If there is no $\mathtt{el}'$ s.t. $(\mathtt{el}', y) \in \mathsf{group}$, sample $\mathtt{el}' \xleftarrow{\$} \{0,1\}^\tau$ randomly but different from all group entries and add $(\mathtt{el}', y)$ to group.
3. Add $(\mathcal{P}_i, (\mathtt{el}, \mathtt{el}'))$ to in and return $(\mathsf{Inv}, \mathsf{sid}, \mathtt{el})$ to $\mathcal{P}_i$.

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

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

Fig. 4. Functionality $\mathcal{F}_{\mathsf{rsw}}$ capturing the time lock assumption of [44].

party gains the power to perform arbitrary operations in $\mathcal{F}_{\mathsf{rsw}}$ but parties who do not know $\mathtt{td}$ are restricted to sequential operations and have no information about the group representation. In particular, in case of an honestly generated group the order will remain completely hidden from the adversary. Finally, this functionality is treated as a global functionality in order to make sure that a simulator does not obtain an unreal advantage in computing the solution of a TLP without waiting for enough ticks.

We remark that our modeling of this time-lock assumption is corroborated by a recent result [45] showing that delay functions (such as TLPs) based on cyclic groups that do not exploit any particular property of the underlying group cannot be constructed if the order is known. It is clear that we cannot reveal any information about the group structure to the environment, since it could use this information across multiple sessions to solve TLPs quicker than the parties. Hence, in order to make it possible to UC-realize $\mathcal{F}_{\mathsf{tlp}}$ based on cyclic groups (and in particular the time-lock assumption of Rivest et al. [44]), we must model the underlying group in such a way that both its structure and its order are hidden from the environment and the parties.

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

Using Rivest *et al.*'s time-lock assumptions modeled in $\mathcal{F}_{\mathsf{rsw}}$ following our sequential computation with ticks framework, we can instantiate Rivest *et al.*'s original time-lock puzzle without running into the issues described before. However, we now face different issues: 1. because all parties are forced by $\mathcal{F}_{\mathsf{rsw}}$ to do sequential computation, a simulator for Rivest *et al.*'s construction would not be able to extract m from $mg^{2^{\Gamma}}$; 2. because $\mathcal{F}_{\mathsf{rsw}}$ deterministically assigns handles to each group element, a simulator would not be able to equivocate $mg^{2^{\Gamma}}$ in such a way that it yields an arbitrary message m'. In order to address these issues, we must resort to a random oracle. More specifically, we work in the restricted programmable and observable global random oracle model $\mathcal{G}_{\mathsf{rpoRO}}$ of [16] (see the full version for the description). It turns out that a programmable random oracle is indeed necessary for UC-realizing $\mathcal{F}_{\mathsf{tlp}}$, as it implies coin flipping with output independent abort as shown in Section 6, which is impossible without a programmable random oracle as shown in Section 7.

We present Protocol π_{tlp} in Figure 5. The main idea behind this protocol is to follow Rivest *et al.*'s construction to compute $\mathtt{puz} = (\mathtt{el}_0, \Gamma, \mathtt{tag})$ while encoding the initial random group element $\mathtt{el}_0$, the message m, the final group element $\mathtt{el}_\Gamma$ and the trapdoor $\mathtt{td}$ for $\mathcal{F}_{\mathsf{rsw}}$ in a $\mathtt{tag}$ generated with the help of the random oracle. This tag is generated in such a way that a party who solves the puzzle can retrieve $\mathtt{td}, m$ and test whether the tag is consistent with these values and with initial and final group elements $\mathtt{el}_0, \mathtt{el}_\Gamma$. More specifically, the tag $\mathtt{tag} = (\mathtt{tag}_2, \mathtt{tag}_2)$ is generated by computing $h_1 = H_1(\mathtt{el}_0|\mathtt{el}_\Gamma)$, $\mathtt{tag}_1 = h_1 \oplus (m|\mathtt{td})$ and $\mathtt{tag}_2 = H_2(h_1|m|\mathtt{td})$, where $H_1(\cdot), H_2(\cdot)$ are random oracles. A party who solves this puzzle obtaining $\mathtt{el}_\Gamma$ by performing Γ sequential squarings of $\mathtt{el}_0$ can retrieve h_1, obtain $(m|\mathtt{td})$ and check that these values are consistent with h_2. Notice that this also allows a simulator who observes queries to random oracles $H_1(\cdot), H_2(\cdot)$ to extract all parameters of a puzzle (including the message)

Protocol π_{tlp}

Protocol π_{tlp} is parameterized by a security parameter τ, a state space $\mathcal{ST} = \{0,1\}^\tau$ and a tag space $\mathcal{TAG} = \{0,1\}^\tau \times \{0,1\}^\tau$. π_{tlp} is executed by an owner $\mathcal{P}_o$ and a set of parties $\mathcal{P}$ interacting among themselves and with functionalities $\mathcal{F}_{\mathsf{rsw}}$, $\mathcal{G}_{\mathsf{rpoRO1}}$ (an instance of $\mathcal{G}_{\mathsf{rpoRO}}$ with domain $\{0,1\}^{2\tau}$ and output size $\{0,1\}^{2\tau}$) and $\mathcal{G}_{\mathsf{rpoRO2}}$ (an instance of $\mathcal{G}_{\mathsf{rpoRO}}$ with domain $\{0,1\}^{4\tau}$ and output size $\{0,1\}^{\tau}$).

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

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

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

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

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

Output: Upon receiving $(\mathsf{Output}, \mathsf{sid})$ as input, a party $\mathcal{P}_i \in \mathcal{P}$ sends $(\mathsf{Output}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{rsw}}$, receiving $(\mathsf{Complete}, \mathsf{sid}, L_i)$ and outputting it. Send $(\mathtt{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$.

Fig. 5. Protocol π_{tlp} realizing time-lock puzzle functionality $\mathcal{F}_{\mathsf{tlp}}$ in the $\mathcal{F}_{\mathsf{rsw}}, \mathcal{G}_{\mathsf{rpoRO}}$-hybrid model.

and check whether it is a valid puzzle. A simulator who also has the additional (and provably necessary) power of programming the output of these random oracles can deliver an arbitrary message m' to a party who solves the puzzle. We formally state the security of π_{tlp} in Theorem 1. Due to space limitations, the proof is contained in the full version.

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

6 Secure Two-Party Computation with Output-Independent Abort

We show how to obtain 2PC with output independent abort from any 2PC with secret-shared outputs using homomorphic commitments with delayed opening.

Functionalities. We will use the following functionalities, for which we present new definitions which take time into consideration:

- The functionality $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ (Fig. 6) for 2PC with Output-Independent Abort.
- The functionality $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ (Fig. 7 and Fig. 8) for secure 2PC with secret-shared output which naturally arises from existing protocols.
- The functionality $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ (see full version) for homomorphic commitments with delayed non-interactive openings that naturally arises from homomorphic commitments that are combined with $\mathcal{F}_{\mathsf{tlp}}$.

An additional functionality $\mathcal{F}_{\mathsf{ct}}$ for coin-flipping with abort in the timed message model appears in the full version [8]. All of the functionalities assume that one of the parties is honest while the other is corrupted, but this is only for simplicity of exposition of the functionalities. We write functionalities where the parties have to send messages to trigger "regular behavior" instead of giving full one-sided control to $\mathcal{S}$ as this appears more natural. Messages to dishonest parties, on the other hand, go directly to $\mathcal{S}$ that can act upon them.

2PC with Output-Independent Abort. The functionality $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ as outlined in Fig. 6 shows how Output-Independent Abort for 2PC can be modeled. Similar to other 2PC functionalities, it allows parties to fix the circuit C to be computed, provide inputs, compute with these inputs and then obtain the result of the computation. In comparison to regular UC functionalities, there are two differences how this is handled:

- Parties using $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ do not receive messages from $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ in a push-model where they get activated upon each new message, but instead they have to pull messages from the functionality (which was also already the case for $\mathcal{F}_{\mathsf{smt,delay}}^{\Delta}$). The reasoning behind this is that the functionality is ticked and it might happen that multiple messages arrive to multiple receivers in the same "tick" round. But upon receiving a message from $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$, a party may not return activation to it. This means that another "tick" may happen before another message gets delivered, which would break the guaranteed delivery requirement. A pull-model is a solution as each party is guaranteed to get activated between any two "ticks" in our model, allowing it to receive messages if it wants to. We will also use this modeling for the other functionalities in this section.
- The functionality does not directly deliver messages to receivers, but instead internally queries them first. This is because it is necessary to use communication using $\mathcal{F}_{\mathsf{smt,delay}}^{\Delta}$, which means that the adversary may arbitrarily control how messages get delivered, and he may reorder delivery at his will within the maximal delay that $\mathcal{F}_{\mathsf{smt,delay}}^{\Delta}$ permits. We also allow the adversary to influence delivery "adaptively", meaning depending on other events outside of $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$'s scope.

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

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

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

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

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

Output: On first input $(\mathsf{Output}, \mathsf{sid})$ by both parties and if y has been stored then add $(\delta, \mathsf{sid}, \mathcal{S}, (\mathsf{Output}, y))$ to $\mathcal{O}$.

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

Scheduling: On input from $\mathcal{S}$:
- If $\mathcal{S}$ sent $(\mathsf{Deliver}, \mathsf{sid}, \mathsf{mid})$ and then remove each $(\mathsf{c}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}, m)$ from $\mathcal{Q}$ and add $(\mathcal{P}, \mathsf{sid}, m)$ to $\mathcal{M}$.
- If $\mathcal{S}$ sent $(\mathsf{Abort}, \mathsf{sid})$ and $\texttt{noabort} = \bot$ then add $(\mathcal{P}_1, \mathsf{sid}, \mathsf{Abort})$, $(\mathcal{P}_2, \mathsf{sid}, \mathsf{Abort})$ to $\mathcal{M}$ and ignore all further calls to the functionality except to **Fetch Message**.

Tick:
1. For each query $(0, \mathsf{mid}, \mathsf{sid}, \mathcal{P}, m) \in \mathcal{Q}$:
   (a) Remove $(0, \mathsf{mid}, \mathsf{sid}, \mathcal{P}, m)$ from $\mathcal{Q}$.
   (b) Add $(\mathcal{P}, \mathsf{sid}, m)$ to $\mathcal{M}$.
2. Replace each $(\mathsf{c}, \mathsf{mid}, \mathsf{sid}, \mathcal{P}, m)$ in $\mathcal{Q}$ with $(\mathsf{c} - 1, \mathsf{mid}, \mathsf{sid}, \mathcal{P}, m)$.
3. For each entry $(\mathsf{c}, \mathsf{sid}, \mathcal{S}, y) \in \mathcal{O}$, proceed as follows:
   - If $\mathsf{c} = 0$, send $(\mathsf{OutputOrAbort}, \mathsf{sid})$ to $\mathcal{S}$. Sample a fresh mid and set $\texttt{noabort} \leftarrow \top$. If $\mathcal{S}$ responds with $(\mathsf{Abort}, \mathsf{sid})$ then add $(\Delta, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_j, (\mathsf{Abort}))$ to $\mathcal{Q}$ for the honest party $\mathcal{P}_j$, otherwise add $(\Delta, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_j, (\mathsf{Output}, y))$. Finally send $(\mathsf{Output}, \mathsf{sid}, \mathsf{mid}, y)$ to $\mathcal{S}$.
   - If $\mathsf{c} > 0$, replace $(\mathsf{c}, \mathsf{sid}, \mathcal{S}, y)$ with $(\mathsf{c} - 1, \mathsf{sid}, \mathcal{S}, y)$ in $\mathcal{O}$.

Fig. 6. The $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ Functionality for 2PC with Output-Independent Abort.

Towards achieving this pull-model and adversarial reordering of messages, $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ has three internal lists $\mathcal{Q}$, $\mathcal{M}$ and $\mathcal{O}$. $\mathcal{Q}$ contains all the buffered messages which can be delivered in the future, while messages in $\mathcal{M}$ can be retrieved right now

by the respective receivers. Whenever $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ notices that a tick happened it will run **Tick**, which will then move all messages from $\mathcal{Q}$ to $\mathcal{M}$ which get available in the next round, and which can be retrieved via the interface **Fetch Message**.

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

We let **Tick** be responsible to realize the output-independent abort property of $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$. To see why this is the case, observe that once both parties activate the output phase the functionality stores a message to $\mathcal{S}$ that represents the output in $\mathcal{O}$. In comparison to $\mathcal{Q}$, $\mathcal{S}$ cannot make $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ output values in $\mathcal{O}$ any faster. Once this message will be delivered to $\mathcal{S}$, the functionality will then ask $\mathcal{S}$ if the honest party should obtain the output or not. It will also give $\mathcal{S}$ control over when the output message should be delivered to the honest party. Observe that once $\mathcal{S}$ obtained the output then the Abort command cannot be used anymore.

Two-Party Computation with Secret-Shared Output. In Fig. 7 and Fig. 8 we describe a 2PC functionality $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ which will be the foundation for our compiler that will realise $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$. $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ has the same initialization, input and computation interfaces as other 2PC functionalities. The two main differences between a standard 2PC functionality and $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ are: first, $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ is again a "ticked" functionality, meaning that it similarly to $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ considers a 2PC protocol that implements communication via $\mathcal{F}_{\mathsf{smt,delay}}^{\Delta}$. Second, $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ does not directly output the outcome of the computation. Instead, it reveals a secret-sharing of it to both parties. The parties can then manipulate shares using the functionality, generate additional random shares or reconstruct them.

We will not show in this work how to realize $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$. This is because it's output-sharing property is rather standard (albeit not always modeled as explicitly as here) and it follows directly from any 2PC protocol that is entirely based on secret-sharing [41] or BMR protocols that secret-share the output [30,6].

Additively Homomorphic Commitments with Delayed Openings. In order to implement $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ we also need a special commitment scheme that allows for delayed openings. The functionality is naturally ticked, as its implementation will use both $\mathcal{F}_{\mathsf{smt,delay}}^{\Delta}$ and $\mathcal{F}_{\mathsf{tlp}}$. Due to space limitations, the functionality as well as its implementation is delayed to the full version [8]. In addition to regular commit and opening procedures, the functionality has a special **Delayed Open** command which releases the message in a commitment after a delay δ. The adversary $\mathcal{A}$ may introduce a (communication) delay of maximum Δ ticks before the honest party receives the delayed opening notification (or it may decide to abort the opening process altogether). However, $\mathcal{A}$ cannot choose to abort the delayed opening anymore once the honest party has received the notification. $\mathcal{A}$ will learn the opening δ ticks after $\mathcal{P}_R$ initiated the delayed opening (as he

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

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

Init: On input $(\mathsf{Init}, \mathsf{sid}, C)$ by $\mathcal{P}_i \in \{\mathcal{P}_1, \mathcal{P}_2\}$:

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

Input: On first input $(\mathsf{Input}, \mathsf{sid}, i, x_i)$ by $\mathcal{P}_i$ for $i \in \{1, 2\}$:

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

Computation: On first input $(\mathsf{Compute}, \mathsf{sid})$ by $\mathcal{P}_i \in \{\mathcal{P}_1, \mathcal{P}_2\}$ and if both x_1, x_2 were accepted:

1. Add $(\Delta, \mathsf{mid}, \mathsf{sid}, \mathcal{P}_{3-i}, (\mathsf{Compute}))$ to $\mathcal{Q}$ for an unused mid.
2. If both parties sent $(\mathsf{Compute}, \mathsf{sid})$ compute $\boldsymbol{y} = (y_1, \ldots, y_m) \leftarrow C(x_1, x_2)$ and store $\boldsymbol{y}$. Then send $(\mathsf{Compute}, \mathsf{sid}, \mathcal{P}_i, \mathsf{mid})$ to $\mathcal{S}$.

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

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

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

Tick:

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

Fig. 7. 2PC with Secret-Shared Output and Linear Share Operations.

receives messages immediately), while an honest receiver $\mathcal{P}_R$ might have to wait $\delta + \Delta$ ticks in total as the ticking for the delayed opening of a commitment can only happen once the opening notification arrives on the receiver's side.

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

6.1 Achieving Output-Independent Abort for 2PC in UC

Intuitively, the protocol realizing $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}$ works as follows: first, both parties use $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ to perform the secure computation. They then don't directly obtain an output, but instead each get a vector of shares $\boldsymbol{s}_i$. Afterwards, the parties will commit to $\boldsymbol{s}_i$ using $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ and use the homomorphic property of $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$

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

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

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

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

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

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

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

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

Fig. 8. 2PC with Secret-Shared Output and Linear Share Operations, Part 2.

to show consistency between the values in $\mathcal{F}_{\mathsf{2pcoia}}^{\Delta,\delta}, \mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$. For this, they sample a random matrix using $\mathcal{F}_{\mathsf{ct}}$ and perform an identical linear operation on both functionalities.

At this stage the protocol might still fail and an adversary might still abort, but no information will leak as the consistency check does only reveal a uniformly random value. Finally, both parties use the **Delayed Open** to reveal their share $\boldsymbol{s}_i$ which allows each party to reconstruct the output. At this stage, $\mathcal{A}$ might decide not to activate **Delayed Open**, but we can set the parameters of $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ such that it will have to do so before the commitment of the honest party opens. If it does not activate its delayed opening before that point, then the honest party will decide that an abort happened and just ignore any future messages

Protocol π_{2pcoia}

This protocol is for two parties $\mathcal{P}_1, \mathcal{P}_2$ and uses the functionalities $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$, $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ and $\mathcal{F}_{\mathsf{ct}}$. The parties compute the circuit C over $\mathbb{F}$ with output length m. We assume that the commitment functionality $\mathcal{F}_{\mathsf{ahcom}}^{\Delta,\delta}$ commits to vectors of length m.
Throughout the protocol, we say "$\mathcal{P}_i$ ticks" when we mean that it sends $(\mathsf{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$. We say that "$\mathcal{P}_i$ waits" when we mean that it, upon each activation, first checks if the event happened and if not, sends $(\mathsf{activated})$ to $\mathcal{G}_{\mathsf{ticker}}$.

Init: Each $\mathcal{P}_i$ sends $(\mathsf{Init}, \mathsf{sid}, C)$ to $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ and ticks. Then it waits and queries $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ for an output $(\mathsf{Init}, \mathsf{sid}, C)$.
Input: Each $\mathcal{P}_i$ sends $(\mathsf{Input}, \mathsf{sid}, i, x_i)$ to $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ and ticks. Then it waits and queries $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ for an output $(\mathsf{Input}, \mathsf{sid}, \mathcal{P}_{3-i})$.
Computation: Each $\mathcal{P}_i$ sends $(\mathsf{Compute}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ and ticks. Then it waits and queries $\mathcal{F}_{\mathsf{2pcsso}}^{\Delta}$ for an output $(\mathsf{Compute}, \mathsf{sid})$.
Output:

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

Fig. 9. Protocol π_{2pcoia} For 2PC with Output-Independent Abort.

of $\mathcal{A}$. The full protocol π_{2pcoia} can be found in Fig. 9. In the full version [8], we show the following theorem:

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

7 The Impossibility Result

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

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

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

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

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

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

Impossibility of UC Time-lock Puzzles. It also follows that UC time-lock puzzles are impossible in the excluded settings. Namely, we have shown that given UC time-lock puzzles one can UC securely realize $\mathcal{F}_{\mathsf{2pcoia}}$, which was excluded above. Due to space constraints, proofs are left to the full version [8].

References

1. M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Fair two-party computations via bitcoin deposits. In *FC 2014 Workshops*, LNCS. Springer, Heidelberg, Mar. 2014.
2. M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Secure multi-party computations on bitcoin. In *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2014.
3. M. Backes, D. Hofheinz, J. Müller-Quade, and D. Unruh. On fairness in simulatability-based cryptographic systems. In *FMSE 2005*, pages 13–22. ACM, 2005.
4. M. Backes, P. Manoharan, and E. Mohammadi. TUC: Time-sensitive and modular analysis of anonymous communication. In *CSF 2014Computer Security Foundations Symposium*. IEEE Computer Society Press, 2014.
5. C. Badertscher, P. Gazi, A. Kiayias, A. Russell, and V. Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In *ACM CCS 2018*. ACM Press, Oct. 2018.
6. C. Baum, B. David, and R. Dowsley. Insured MPC: Efficient secure computation with financial penalties. In *FC 2020*, LNCS. Springer, Heidelberg, Feb. 2020.
7. C. Baum, B. David, R. Dowsley, J. B. Nielsen, and S. Oechsner. Craft: Composable randomness and almost fairness from time. Cryptology ePrint Archive, Report 2020/784, 2020. `https://eprint.iacr.org/2020/784`.
8. C. Baum, B. David, R. Dowsley, J. B. Nielsen, and S. Oechsner. TARDIS: Time and relative delays in simulation. Cryptology ePrint Archive, Report 2020/537, 2020. `https://eprint.iacr.org/2020/537`.
9. C. Baum, E. Orsini, and P. Scholl. Efficient secure multiparty computation with identifiable abort. In *TCC 2016-B, Part I*, LNCS. Springer, Heidelberg, Oct. / Nov. 2016.
10. 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*, LNCS. Springer, Heidelberg, Aug. 2020.
11. M. Bellare, R. Dowsley, B. Waters, and S. Yilek. Standard security does not imply security against selective-opening. In *EUROCRYPT 2012*, LNCS. Springer, Heidelberg, Apr. 2012.

12. I. Bentov and R. Kumaresan. How to use bitcoin to design fair protocols. In *CRYPTO 2014, Part II*, LNCS. Springer, Heidelberg, Aug. 2014.
13. N. Bitansky, S. Goldwasser, A. Jain, O. Paneth, V. Vaikuntanathan, and B. Waters. Time-lock puzzles from randomized encodings. In *ITCS 2016*. ACM, Jan. 2016.
14. D. Boneh, J. Bonneau, B. Bünz, and B. Fisch. Verifiable delay functions. In *CRYPTO 2018, Part I*, LNCS. Springer, Heidelberg, Aug. 2018.
15. D. Boneh and M. Naor. Timed commitments. In *CRYPTO 2000*, LNCS. Springer, Heidelberg, Aug. 2000.
16. J. Camenisch, M. Drijvers, T. Gagliardoni, A. Lehmann, and G. Neven. The wonderful world of global random oracles. In *EUROCRYPT 2018, Part I*, LNCS. Springer, Heidelberg, Apr. / May 2018.
17. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. `http://eprint.iacr.org/2000/067`.
18. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*. IEEE Computer Society Press, Oct. 2001.
19. R. Canetti, Y. Dodis, R. Pass, and S. Walfish. Universally composable security with global setup. In *TCC 2007*, LNCS. Springer, Heidelberg, Feb. 2007.
20. I. Cascudo, I. Damgård, B. David, N. Döttling, R. Dowsley, and I. Giacomelli. Efficient UC commitment extension with homomorphism for free (and applications). In *ASIACRYPT 2019, Part II*, LNCS. Springer, Heidelberg, Dec. 2019.
21. R. Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *18th ACM STOC*. ACM Press, May 1986.
22. G. Couteau, B. Roscoe, and P. Ryan. Partially-fair computation from timed-release encryption and oblivious transfer. Cryptology ePrint Archive, Report 2019/1281, 2019. `https://eprint.iacr.org/2019/1281`.
23. I. Damgård and J. Groth. Non-interactive and reusable non-malleable commitment schemes. In *35th ACM STOC*. ACM Press, June 2003.
24. B. David, P. Gazi, A. Kiayias, and A. Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *EUROCRYPT 2018, Part II*, LNCS. Springer, Heidelberg, Apr. / May 2018.
25. C. Dwork, M. Naor, and A. Sahai. Concurrent zero-knowledge. In *30th ACM STOC*. ACM Press, May 1998.
26. N. Ephraim, C. Freitag, I. Komargodski, and R. Pass. Non-malleable time-lock puzzles and applications. Cryptology ePrint Archive, Report 2020/779, 2020. `https://eprint.iacr.org/2020/779`.
27. M. Fischlin, A. Lehmann, T. Ristenpart, T. Shrimpton, M. Stam, and S. Tessaro. Random oracles with(out) programmability. In *ASIACRYPT 2010*, LNCS. Springer, Heidelberg, Dec. 2010.
28. J. A. Garay, P. D. MacKenzie, M. Prabhakaran, and K. Yang. Resource fairness and composability of cryptographic protocols. In *TCC 2006*, LNCS. Springer, Heidelberg, Mar. 2006.
29. O. Goldreich. Concurrent zero-knowledge with timing, revisited. In *34th ACM STOC*. ACM Press, May 2002.
30. C. Hazay, P. Scholl, and E. Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. In *ASIACRYPT 2017, Part I*, LNCS. Springer, Heidelberg, Dec. 2017.
31. D. Hofheinz and V. Shoup. GNUC: A new universal composability framework. *Journal of Cryptology*, (3), July 2015.
32. Y. Ishai, R. Ostrovsky, and V. Zikas. Secure multi-party computation with identifiable abort. In *CRYPTO 2014, Part II*, LNCS. Springer, Heidelberg, Aug. 2014.

33. Y. T. Kalai, Y. Lindell, and M. Prabhakaran. Concurrent general composition of secure protocols in the timing model. In *37th ACM STOC*. ACM Press, May 2005.
34. J. Katz, J. Loss, and J. Xu. On the security of time-lock puzzles and timed commitments. In *TCC 2020, Part III*, LNCS. Springer, Heidelberg, Nov. 2020.
35. J. Katz, U. Maurer, B. Tackmann, and V. Zikas. Universally composable synchronous computation. In *TCC 2013*, LNCS. Springer, Heidelberg, Mar. 2013.
36. A. Kiayias, H.-S. Zhou, and V. Zikas. Fair and robust multi-party computation using a global transaction ledger. In *EUROCRYPT 2016, Part II*, LNCS. Springer, Heidelberg, May 2016.
37. R. Kumaresan and I. Bentov. How to use bitcoin to incentivize correct computations. In *ACM CCS 2014*. ACM Press, Nov. 2014.
38. R. Kumaresan, T. Moran, and I. Bentov. How to use bitcoin to play decentralized poker. In *ACM CCS 2015*. ACM Press, Oct. 2015.
39. U. Maurer. Constructive cryptography - A new paradigm for security definitions and proofs. In *TOSCA 2011*, volume 6993 of *Lecture Notes in Computer Science*, pages 33–56. Springer, 2011.
40. J. B. Nielsen. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In *CRYPTO 2002*, LNCS. Springer, Heidelberg, Aug. 2002.
41. J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A new approach to practical active-secure two-party computation. In *CRYPTO 2012*, LNCS. Springer, Heidelberg, Aug. 2012.
42. B. Pfitzmann and M. Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *2001 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2001.
43. K. Pietrzak. Simple verifiable delay functions. In *ITCS 2019*. LIPIcs, Jan. 2019.
44. R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. 1996.
45. L. Rotem, G. Segev, and I. Shahaf. Generic-group delay functions require hidden-order groups. In *EUROCRYPT 2020, Part III*, LNCS. Springer, Heidelberg, May 2020.
46. V. Shoup. Lower bounds for discrete logarithms and related problems. In *EUROCRYPT'97*, LNCS. Springer, Heidelberg, May 1997.
47. H. Wee. Zero knowledge in the random oracle model, revisited. In *ASIACRYPT 2009*, LNCS. Springer, Heidelberg, Dec. 2009.
48. B. Wesolowski. Efficient verifiable delay functions. In *EUROCRYPT 2019, Part III*, LNCS. Springer, Heidelberg, May 2019.