Formalizing Delayed Adaptive Corruptions and the Security of Flooding Networks*

Christian Matt[1][0000−0001−5900−336X], Jesper Buus Nielsen[2][0000−0002−7074−0683], and Søren Eller Thomsen[2][0000−0002−6931−4740]

[1] Concordium, Zurich, Switzerland cm@concordium.com
[2] Concordium Blockchain Research Center, Aarhus University, Denmark
{jbn,sethomsen}@cs.au.dk

Abstract. Many decentralized systems rely on flooding protocols for message dissemination. In such a protocol, the sender of a message sends it to a randomly selected set of peers. These peers again send the message to their randomly selected peers, until every network participant has received the message. This type of protocols clearly fail in face of an adaptive adversary who can simply corrupt all peers of the sender and thereby prevent the message from being delivered. Nevertheless, flooding protocols are commonly used within protocols that aim to be cryptographically secure, most notably in blockchain protocols. While it is possible to revert to static corruptions, this gives unsatisfactory security guarantees, especially in the setting of a blockchain that is supposed to run for an extended period of time.

To be able to provide meaningful security guarantees in such settings, we give precise semantics to what we call *δ-delayed adversaries* in the Universal Composability (UC) framework. Such adversaries can adaptively corrupt parties, but there is a delay of time δ from when an adversary decides to corrupt a party until they succeed in overtaking control of the party. Within this model, we formally prove the intuitive result that flooding protocols are secure against δ-delayed adversaries when δ is at least the time it takes to send a message from one peer to another plus the time it takes the recipient to resend the message. To this end, we show how to reduce the adaptive setting with a δ-delayed adversary to a static experiment with an Erdős–Rényi graph. Using the established theory of Erdős–Rényi graphs, we provide upper bounds on the propagation time of the flooding functionality for different neighborhood sizes of the gossip network. More concretely, we show the following for security parameter κ, point-to-point channels with delay at most Δ, and n parties in total, with a sufficiently delayed adversary that can corrupt any constant fraction of the parties: If all parties send to $\Omega(\kappa)$ parties on average, then we can realize a flooding functionality with maximal delay $\mathcal{O}\big(\Delta \cdot \log(n)\big)$; and if all parties send to $\Omega\big(\sqrt{\kappa n}\big)$ parties on average, we can realize a flooding functionality with maximal delay $\mathcal{O}(\Delta)$.

* 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).

Keywords: adaptive adversaries · corruption models · universal composability · flooding networks · peer-to-peer networks · blockchain

1 Introduction

1.1 Motivation

In *Nakamoto-style blockchains* (NSBs) such as Bitcoin [36], several parties continuously try to solve cryptographic puzzles. The first party solving the puzzle "wins" the right to create a new block extending the previously longest chain. This block is then distributed to all other parties, who continue solving puzzles to create the next block. Extensive research has shown for different variation of NSBs that security can be guaranteed if a majority of the puzzles are solved by honest parties and if blocks can be propagated fast enough to ensure with high probability that the next winner has learned about the previous block before creating a new block [20,21,38,40].

Since future block creators are unpredictable, these protocols have a high resilience against adaptive corruptions. Intuitively, the only chance to exploit the adaptivity of corruptions is to corrupt a party after learning that it has solved a puzzle and subsequently prevent this party from distributing the created block. An adversary with the power to stop messages from being delivered (or changing the message) by corrupting the sender after sending but before the message is delivered, is often referred to as *strongly adaptive* [1]. On the other hand, if messages from honest senders are guaranteed to be delivered regardless of whether the sender gets corrupted before delivery, the adversary is only *weakly adaptive*, or equivalently, *atomic message send* (AMS) [19] is assumed.

Indeed, several papers [20,21,38] have proven the security of Bitcoin's consensus against adaptive corruptions, and Ouroboros Praos [15] has been developed as a proof-of-stake blockchain with resilience against fully adaptive corruptions as one of the main selling points. To achieve this, these papers have to assume atomic message dissemination. In reality, however, NSBs typically use complex peer-to-peer networks to disseminate blocks, in which each party propagates messages to only a small set of other parties (referred to as their neighbors), who will then propagate it to their neighbors and so forth. Even if the point-to-point channels between neighbors allow atomic sends, the overall network will not provide this guarantee because an adaptive adversary can simply corrupt all neighbors of the sender and thereby stop the block from being propagated. Hence, when considering the full protocol, which combines a NSB with a peer-to-peer flooding network, security against fully adaptive corruptions can no longer be guaranteed.

Formalizing delayed adaptive corruptions. To provide meaningful guarantees to blockchain protocols including their peer-to-peer network, we observe that intuitively, one needs to restrict the *corruption speed* of an adversary such that parties in the peer-to-peer network have enough time to pass on the block they receive before being corrupted. Based on this observation, we introduce a precise

model for δ-*delayed adversaries* in the Universal Composability framework [7]. Using this model, one can quantify the minimum amount of time δ it takes from when an adversary targets and starts attacking a specific party until this party is actually under adversarial control and prove the security of protocols against such corruptions. This allows us to describe exactly what kind of adversaries different P2P networks and protocols build on top can withstand.

Note that the corruption speed of an adaptive adversary also has a natural translation to reality. For an attacker to succeed in attacking some physical machine it necessarily takes some time from targeting the machine to actually hack into the network (by either physical or digital means) and take over the computer. Denial-of-service attacks are arguably faster to mount, but it still takes nonzero time to target a specific machine.

While unstructured peer-to-peer networks for message dissemination are the main focus of our paper, delayed adversaries have much broader applications and were in fact already used in other works, with varying degree of formality. For example, the original Ouroboros [28], in contrast to its successor Ouroboros Praos [15], which only requires AMS, needs that corruptions are sufficiently delayed. The same is true for Snow White [14], another early proof-of-stake blockchain. Another example is Hybrid Consensus [39], which periodically elects committees using a blockchain and remains secure if corruptions are delayed until the next committee is selected. The same applies to blockchain sharding proposals [32,30,42] in which the members of shards are periodically chosen.

Concrete analysis of flooding networks. As mentioned above, the security of NSBs crucially relies on the assumption that blocks are with high probability propagated to other parties before the next winner creates another new block. If an upper bound on the propagation time is known, the difficulty of the puzzles can be set accordingly to provide this guarantee. Setting the difficulty based on a too optimistic assumption on the delay jeopardizes the security of the system, and setting it based on a too loose upper bound degrades efficiency. Knowing a tight bound on the propagation delay is thus key for the security and efficiency of an NSB.

Even more critical for the security of NSBs are so-called eclipse attacks that prevent some parties from receiving blocks [23,33]. Furthermore, for large-scale distributed systems, the number of neighbors has a significant impact on the required communication. In particular, it is infeasible to simply send the message directly to everybody. In this work, we provide constructions for flooding networks with provable security against eclipse attacks in a well-defined adversarial model and show different trade-offs between the propagation time and neighborhood sizes.

Terminology. In the literature different terminology has been used for the process of disseminating a message to all parties. Common terminology includes "broadcast", "flood" and "multicast". In this paper, we will use the terminology "flood" for this process. Contrary to *byzantine broadcast*, there is no agreement requirement for a flooding network if the sender of a message is dishonest.

1.2 Contributions and Results

Our contributions are twofold:

1. We give precise semantics to δ-*delayed corruptions* (introduced in [39] as δ-agile corruptions) within the UC framework [8]. We define the semantics via corruption shells which allows us to prove how this type of corruptions relate to standard adaptive corruptions.
2. We define a functionality for disseminating information, Flood, that can be used to implement a secure NSB, and that we implement using a flooding protocol against a *slightly delayed* adversary. Importantly, we quantify exactly how much is meant by "slightly" in terms of guarantees provided by the underlying point-to-point channels. We provide two instantiations of our protocol with different efficiency trade-offs.

Below we lay out the specifics of the individual contributions and state our results in more detail.

Precise model for δ-delayed adversaries. We define a δ-delayed adversary as an adversary which uses at least δ time to perform a corruption. We define this notion precisely within the UC framework using the notion of time from [4]. We do so by elaborating on the notion of corruption-shells from [8].

Using the idea of corruption shells, we give semantics to both "normal" byzantine adaptive corruption and δ-delayed corruptions. We capture the semantics of byzantine adaptive corruptions in a corruption-shell, $\mathcal{B}_{\mathsf{Real}}$, for protocols and in a corruption-shell, $\mathcal{B}_{\mathsf{Ideal}}$, for ideal functionalities. Similarly, we capture the semantics of δ-delayed adversaries in a corruption-shell, $\mathcal{D}^{\delta}_{\mathsf{Real}}$, for protocols and in a corruption-shell, $\mathcal{D}^{\delta}_{\mathsf{Ideal}}$, for ideal functionalities. $\mathcal{D}^{\delta}_{\mathsf{Real}}$ and $\mathcal{D}^{\delta}_{\mathsf{Ideal}}$ accepts two inputs: *Precorrupt* and *Corrupt* (both indexed by a specific party). Both shells ensure that at least δ time has passed after receiving *Precorrupt* before reacting upon *Corrupt*. Any *Corrupt* input that is sent prematurely is ignored.

Having defined the semantics for both standard adaptive corruptions and for δ-delayed corruptions using corruption shells, we state basic results relating the two models. We show that a protocol is secure against a standard adaptive adversary *iff* it is secure against a 0-delayed adversary (Theorem 1). Furthermore, we show that if a protocol is secure against a "fast" adversary, then this implies that it is also secure against a "slow" adversary (Theorem 2). Together these results allow constructions proven secure in the standard model of adaptive adversaries to be reused when constructing new protocols secure against a δ-delayed adversary, and to compose protocols that are secure against adversaries with different delays.

Flooding networks. We define a functionality for flooding messages, $\mathcal{F}^{\Delta}_{\mathsf{Flood}}$. It ensures that all parties learn messages that an honest party has sent or has received within Δ time, and is thereby similar to the flooding functionality assumed in many consensus protocols. We realize our flooding functionality with both a naive protocol, $\pi_{\mathsf{NaiveFlood}}$, where everybody simply sends to everybody,

and a more advanced protocol, $\pi_{\mathsf{ERFlood}}(\rho)$, where all parties choose to send to other parties with probability ρ.

In order to realize the flooding functionality, we introduce a functionality for a point-to-point channel $\mathcal{F}^{\sigma,\Delta}_{\mathsf{MessageTransfer}}$. This functionality is also parameterized by a bound for the delivery time Δ, and additionally has a parameter σ describing the time an honest party needs to stay honest after starting to send the message for the delivery guarantee to apply. If $\sigma = 0$ then this corresponds to assuming AMS. On the other hand, if $\sigma \geq \delta$ and we consider δ-delayed adversaries, then this corresponds to not assuming AMS. However, having the time quantified allows us to relate this time to the delay we can tolerate when building more advanced constructions. In particular, we show that π_{ERFlood} using $\mathcal{F}^{\sigma,\Delta}_{\mathsf{MessageTransfer}}$ implements $\mathcal{F}^{\Delta'}_{\mathsf{Flood}}$ against a $(\sigma + \Delta)$-delayed adversary.

In this setting, we provide two different ways to instantiate the probability parameter ρ of π_{ERFlood}, each presenting a different efficiency trade-off. Concretely, let h denote the minimum number of parties that will stay honest throughout the execution of the protocol, let n denote the total number of parties, and let κ be the security parameter. We provide the following two instantiations:

Instantiation 1: Guaranteed delivery within $\Delta' \coloneqq 2 \cdot \Delta$ for $\rho \coloneqq \sqrt{\frac{\kappa}{h}}$.
Instantiation 2: Guaranteed delivery within $\Delta' \coloneqq \Delta \cdot \left(5 \log\left(\frac{n}{2\kappa}\right) + 2\right)$ for $\rho \coloneqq \frac{\kappa}{h}$.

Both instantiations ensure that the statistical distance between the ideal and the real executions of π_{ERFlood} and $\mathcal{F}^{\Delta'}_{\mathsf{Flood}}$ is negligible in the security parameter. We provide concrete bounds for the statistical distance in Corollary 1. Furthermore, standard probability bounds ensure that each instantiation has a neighborhood of $\mathcal{O}(n \cdot \rho)$ with high probability.

1.3 Techniques

An Erdős–Rényi graph [18] is a graph where each edge appears with an equal and independent probability. Our flooding protocol π_{ERFlood} is strongly inspired by this type of graph. Our main technical contributions are thus concerned with transporting bounds for Erdős–Rényi graphs to the cryptographic setting, especially in presence of adaptive adversaries.

Concrete bounds for Erdős–Rényi graphs. The asymptotic behavior of Erdős–Rényi graphs has been thoroughly studied in the literature (for a comprehensive overview see [6]). However, bounds about a graph's behavior when the amount of nodes goes towards infinity is of little use for protocols that are supposed to be run by a finite number of parties. For a protocol imitating the behavior of such graphs, we need concrete bounds when a security parameter is increased. As a technical contribution, we prove such concrete upper-bounds for the diameter of Erdős–Rényi graphs.

Applying Erdős–Rényi graph results in the presence of adaptive adversaries. For a flooding protocol as π_{ERFlood}, it is straightforward to apply bounds about the

diameter of an Erdős–Rényi to also bound the probability that a message is not delivered in the protocol in presence of a *static* adversary. However, for an adaptive adversary that is capable of preventing certain nodes from connecting to their neighbors, it is by no means this easy. Our main technical contribution is to transfer the bounds on the diameter of an Erdős–Rényi graph to our flooding protocol in presence of an adaptive adversary. We achieve this by relating the protocol execution to 7 random experiments.

First, we relate the protocol execution to a well-defined game between an adversary and an oracle, which returns a graph. The rules of the game is that an adversary can query the oracle to reveal the edges of a node and query the oracle to remove a node from the graph. However, once either an incoming or outgoing edge to a node has been revealed, the adversary can no longer remove this node. This game mimics the powers of a slightly delayed adaptive adversary in the real protocol.

We relate this game to a similar game but with undirected edges, and do a couple of simple gamehops where we show that an adversary does not gain any additional advantage w.r.t. increasing the diameter by stopping this game at an early point nor injecting any additional edges.

As the adversary can only remove nodes for which no information has been revealed, one might be led to believe that the Erdős–Rényi graph results apply for this game. However, the adversary can still dynamically control the size of the graph that is returned. At first, this may seem innocent, but in fact, it is not. Deciding whether or not more nodes are to be included in the graph, can amplify the probability that the returned graph has a high diameter.

Therefore, we relate this game to a new game, which is similar to the other, except that the oracle now at random fixes the size of the graph beforehand. The oracle fixes the size of the graph by making a uniform guess in the range of possible sizes. In case of a correct guess (a guess identical to what the adversary anyway would end up with), the adversary is only left with the choice of which parties to include in the random graph. Finally, we show that this game is equally distributed to a game which specifically embeds an Erdős–Rényi graph of the fixed size. This allows us to apply results bounding diameter of Erdős–Rényi graphs to bound the probability that a message is not delivered timely.

Due to space constraints, many technical details are left out of this version. We refer to the full version of this paper [34] for these.

1.4 Related Work

Hybrid consensus. Hybrid Consensus [39] is a consensus protocol that uses a blockchain to periodically select committees as subsets of the parties participating in the blockchain protocol, who can subsequently produce blocks more efficiently. Once a committee has been chosen, a fully adaptive adversary can simply corrupt the majority of its members to break the security of the protocol. Hence, the protocol is only secure against corruptions that are delayed until the next committee gets selected.

To prove the security of hybrid consensus, that paper introduces τ-*agile corruptions*, which essentially correspond to the capabilities of our τ-delayed adversaries. While that paper also uses the UC framework, the definitions for the corruption model mostly remain at a high level. For example, their definitions assume there is some notion of time, which does not exist in the original UC formalism. There are also no clear definitions of how the delayed corruptions are precisely embedded in the UC execution model.

In contrast to that, our work provides a precise embedding of the corruption model in the standard UC framework. This allows us to compose protocols formulated in standard UC with protocols proven secure against δ-delayed adversaries. It is thus fair to say that the hybrid consensus paper has introduced the delayed corruption model at an intuitive, semi-formal level, while our work fills in several missing technical details to provide a precise formalization within the UC framework.

Time in UC. [26] models time using a clock functionality that is local to each protocol. This functionality synchronizes the parties by only allowing the adversary to advance time when all parties have reported that they have been activated. As this is a *local* functionality, other ideal functionalities have no access to it, and therefore need to provide their own notion of time which can clutter the final guarantees from the functionality.

[29] takes a similar approach to Katz et al., but changes the clock to be a *global functionality* in GUC [9]. This enables several different protocols to rely on the same notion of time when composed and also solves the problem of time not being available to ideal functionalities. Both functionalities and parties can query the global clock for the current time, and thus inherently makes any protocol modelled with this a synchronous protocol.

A different approach is taken in [4]. They take the standpoint that parties should be oblivious to the passing of time. To allow this they introduce a global functionality, dubbed a *ticker* (written $\bar{\mathcal{G}}_{\text{Ticker}}$), which exposes an interface to learn about the passing of time to functionalities *only*. In particular honest parties are oblivious to the passing of time. This allows time to be modelled without having synchrony as an inherent assumption. The specific timing-assumptions can then be captured by adding an extra ideal functionality which exposes relevant information to the parties.

Contrary to [26,29,4], [10] focuses on modeling and making *real* time available to parties in GUC, and use this to model the expiration of certificates in a public-key infrastructure. In their modeling, a global clock can be advanced by the environment without restrictions. In this work, our protocols do not rely on real-time, but rather on an abstract notion of time used to state assumptions and guarantees about the delivery time of channels and protocols. For the guarantees to be upheld, we rely on restrictions about how time is advanced (namely that all parties have to be activated once each abstract time step) by the environment.

We chose to rely on the modeling of time from [4]. This allows us to model general timing assumptions on the capabilities of the adversary without tying our modeling to a particular assumption on synchrony for actual protocols.

Epidemic and gossip protocols. Epidemic algorithms or gossip protocols were first considered for data dissemination by Demers et al. [17], and have been studied extensively since then, see e.g., [5,25,27,22,13,24]. In this line of work, many different protocols have been considered. Some are very closely related to our flooding protocol, where parties simply forward to a random set of parties, and some are more advanced, letting parties keep sending to new random peers until a certain number of recipients replied that they already knew the message. However, this line of work considers only random failures [27] or incomplete network topologies [13,24] and not adaptive corruptions of a malicious adversary. Hence, while some of the protocols are applicable to our setting, their analysis is not. Among other results, [27] showed how random node failures affect the success probability of a flooding process similar to ours. For this setting, they derive connectivity bounds similar to the bounds for logarithmic diameter we present in this work.

Kadcast. Kadcast [41] is a structured peer-to-peer network for blockchains. The paper claims that unstructured networks are inherently inefficient because many superfluous messages are sent to parties who already received the message from other peers. They instead propose a structured network based on Kademlia [35], in which every node has $\mathcal{O}(\log n)$ neighbors and the diameter of the graph is also $\mathcal{O}(\log n)$. Additionally, their protocol includes a parameter for controlling the redundancy and thus the resistance to attacks. Due to the structured nature, the suggested network is, however, not secure against adaptive corruptions of any kind.

The hidden graph model. Chandran et al. [11] consider *communication locality* of multi-party computation (MPC) protocols, which corresponds to the maximal number of parties each honest party needs to interact with. They construct an MPC protocol with poly-logarithmic communication locality that is secure against adaptive corruptions and that runs in a poly-logarithmic number of rounds. Their protocol uses a random communication graph, similar to our flooding protocols. To be secure against adaptive corruptions, they however need to assume that the communication graph between honest parties remains hidden, i.e., they allow honest parties to communicate securely without an adversary learning who is communicating with whom. Furthermore, they only prove very loose bounds on the locality and diameter of the obtained graph by showing that both are poly-logarithmic. In the full version of this work [34], we replicate this result but with concrete bounds.

Message dissemination relying on resource assumptions. Recently, the problem of disseminating messages assuming a constant fraction of honest resources (computational power, stake, etc.) instead of assuming a constant fraction of honest parties (as assumed in this work) has received attention. Extending on results from this work, [31] provides an efficient flooding protocol relying on a constant fraction of the resources behaving honestly. Their protocols achieve an asymptotic efficiency similar to the protocol presented in this work. [12] presents

a block dissemination protocol for the Ouroboros Praos protocol [16] that also relies on the majority of honest stake assumption. By using long-lived connections between parties, they prevent a specific denial-of-service attack possible in the protocol. However, this comes at the cost of allowing a small fraction of honest parties to be eclipsed.

2 Preliminaries

2.1 Notation

We use the infix notation ":=" for assigning a variable a (new) value, the infix notation "$\triangleq$" to emphasize that a concept is being defined formally for the first time, the infix notation "==" to denote an equality test returning a boolean value, and the infix notation "::" to denote list-extension. In our proofs we will use the acronyms LHS and RHS to refer to respectively the left-hand side and the right-hand side of an equality.

When describing functionalities we let $\mathcal{P}$ be a set of unique party identifiers (PIDs) and will leave out session-identifiers for clarity of presentation. As a convention we use the variable $t \in \mathbb{N}$ to denote the maximal number of parties an adversary can corrupt, use the variable $n := |\mathcal{P}|$ to denote the total number of parties in a protocol (except when we state and prove general results about graphs) and $h := n - t$ to denote the minimal number of honest parties. Whenever we refer to *honest* parties we will refer to parties that have not received any precorrupt or corrupt tokens.

2.2 Universally Composable Security

The UC framework is a general framework for describing and proving cryptographic protocols secure. Its main selling point is that protocols can be described and proven secure in a modular manner while ensuring that the protocol in question remains secure independently of how one may compose the protocol in question with other protocols. We build upon the journal version of UC [8] and refer to this for details about the framework. Below we recap two peculiarities of the framework that are important for our model (Section 3).

Corruptions The UC framework has no built-in semantics for corruption of parties in a protocol. Instead, it is up to each individual protocol description to describe the semantics of corruptions whenever the adversary signals that a specific party should be corrupted. Having no built-in corruption model in UC makes the composition theorem independent of a particular corruption model. This allows several different corruption models to be captured within the framework. Some machinery is however common for many different types of corruptions.

The corruption aggregation ITI. The intuition behind UC-security is to translate an attack on the protocol to an attack on the specification (the ideal functionality) and thereby show that an adversary does not gain any capabilities interacting with an implementation that another adversary did not have interacting with the ideal functionality. That is to show that any attack is not really an attack as it was already allowed by the specification. This translation between attacks is what is known as a simulator.

For this intuition to make sense when active corruptions are possible, the translation between attacks on the protocol and the specification necessarily needs to be *corruption preserving*. That is, it should not require more corruptions to attack the ideal functionality than what it takes to attack the real protocol. In order to ensure this, an additional Interactive Turing Machine Instance (ITI) called the *corruption aggregation* ITI is run aside the parties in protocol. Whenever a party is corrupted, it registers as corrupted by the corruption aggregation ITI. The environment can then query the corruption aggregation ITI in order to get an overview of who is currently corrupted. Similarly, the ideal functionality makes information about who is corrupted available to the environment. Note that the corruption aggregation ITI is only present for modeling purposes and thus not present when deploying a protocol. In that way, if the simulator corrupts differently than the adversary, the environment is immediately able to distinguish.

Identity masking function and PIDs. The UC framework allows for a very fine-grained control over what knowledge about corruptions is leaked to the environment, by parameterizing the corruptions using an *identity-masking-function*, which parties will apply to the information that they send to the corruption aggregation ITI. This can allow an adversary to corrupt only sub-protocols of a party instead of an entire party. We leave this out of the definitions below for clarity as we will always consider corruptions of entire parties (known as PID-wise corruptions within the framework).

Time There is no built-in notion of time in UC. However, the flexibility of the framework allows to model a notion of time using an ideal functionality. In this work we adopt the notion of time presented in TARDIS [4].

In TARDIS time is modelled via a global functionality dubbed a *ticker* (written $\bar{\mathcal{G}}_{\text{Ticker}}$). The ticker's job is to keep track of time and enforce that any party has enough time to perform the actions that it wishes to perform between any two time-steps. It does so by allowing parties to register by the functionality and only allows the environment to progress time once it has heard that this is okay from all registered parties.

Functionalities can query the ticker and get an answer to whether or not time has passed since the last time they asked the ticker. Importantly, this query can *only* be made by functionalities and not parties. That is, this modeling of time does not tie the protocols to be designed under a specific synchrony assumption, as parties are oblivious to time. The only way that they can observe the passing of time is by asking functionalities. This parallels the real world in that we do

not have raw access to time, only clocks. The level of information functionalities provide to parties about time is what determines possible assumptions about synchrony.

The complete ticker functionality as described in TARDIS as well as a small note about preventing fast-forwarding is provided in the full version of this work [34].

Ticked?-convention. In the remainder of this paper, we adopt the convention (also used in [4]) that when describing ideal functionalities we omit *Ticked?* queries to $\bar{\mathcal{G}}_{\text{Ticker}}$ from the description. Functionalities are instead assumed to make this query whenever they are activated and in case of a positive answer perform whatever action that is described by **Tick**. We furthermore adopt the convention that for brevity we leave out registration of functionalities and parties by the global ticker. All of the functionalities and protocols we consider will upon initialization as the first thing register by the global ticker.

Global functionalities within plain UC. Technically, the ticker functionality in TARDIS is defined within the GUC framework [9]. However, as pointed out in [2], the GUC framework has not been updated since its introduction, even though that it relies on the UC framework which has been revised and updated several times since. Furthermore, [2] points out that several technical subtleties of the composition theorem of GUC are under-specified which at best leaves its correctness unproven. The compatibility with the latest version of UC which we use in this work is thus unclear.

However, [2] introduces machinery to handle "global subroutines", which can be used to model similar global setup assumptions to global functionalities, and extends the composition theorem of UC to cover such "global subroutines" directly within the version of UC also adapted for this work. Additionally, they show how examples of global functionalities that instead can be modelled as global subroutines. One of their examples [2, Section 4.3] of such a transformation is, that they show that [3] that implements a transaction ledger using a global clock (similar to the one from [26]), instead could have been done directly within UC, by modeling the clock as a global subroutine instead of a global functionality. We note that $\bar{\mathcal{G}}_{\text{Ticker}}$ is *regular* (informally, it does not spawn new ITIs) and as all of the protocols considered in this work are $\bar{\mathcal{G}}_{\text{Ticker}}$*-subroutine respecting* (informally, all subroutines except $\bar{\mathcal{G}}_{\text{Ticker}}$ only communicate with ITIs within the session). Therefore, we can use the same approach as [2, Section 4.3] (in particular can adopt the same *identity bound* for the environment to ensure that the ticker works as expected) to keep our modeling within plain UC.

3 Delayed Adversaries within UC

In this section we describe the semantics of delayed corruptions within the UC framework. First, we introduce the semantics for δ-delayed corruptions via *corruption shells*. Next, we revisit the standard adaptive corruptions using

corruption-shells. Finally, we relate the standard notion of adaptive corruptions to a 0-delayed adversary.

We define the notion of a delayed adversary precisely within the UC-model via what we call *δ-delayed corruptions* or a *δ-delayed adversary*. For such an adversary, it takes at least δ time to execute a corruption. The delay can be thought of as either the time it takes to hack into the system or the time it takes to physically orchestrate and attack on the specific property that hosts the system. To capture this within UC we introduce an additional token that an adversary has to use when wanting to corrupt a party. The two corruption tokens that can be passed to a party are the *Precorrupt* token and the *Corrupt* token. When receiving a *Precorrupt* token, the party notes the time it received this token, t, and ignores all *Corrupt* tokens that are received before $t + \delta$. When a *Corrupt* token is received at or after time $t + \delta$, the party becomes corrupted in the usual manner.

Below we give a more precise description of how this corruption model can be captured within the UC framework.

3.1 The δ-delay Shell

It is tedious and error-prone to include code that models corruption behavior in each protocol description and ideal functionality description. We therefore separate the concern of describing corruption behaviors to that of describing the protocol, by introducing protocol transformers, dubbed *shells*, which extend a protocol that does not handle corruption tokens into one that obeys a particular corruption behavior. In particular we provide the following two shells for δ-delayed corruptions:

$\mathcal{D}^{\delta}_{\mathsf{Real}}$: This a wrapper around a protocol π. It ensures that the protocol respects δ-delayed corruptions. The wrapper preserves the functionality of π but additionally ensures that corruptions are executed as expected.

$\mathcal{D}^{\delta}_{\mathsf{Ideal}}$: This is a wrapper around an ideal functionality $\mathcal{F}$. It ensures that the functionality respects δ-delayed corruptions and preserves the functionality of $\mathcal{F}$ but additionally ensures that corruptions are executed as expected.

Both shells intuitively work in the same way: They keep track of when *Precorrupt* tokens are delivered and only accept corruption tokens for a particular party δ time later. Having two different shells is, however, necessary as the protocol shell needs to wrap the individual ITMs actually executing the protocol, whereas the ideal shell needs to wrap only the ITM running the ideal functionality.

Additionally, both shells allow the first message that is sent to a specific party to initialize the precorruption time. The delay shells for real parties ensure to use this initialization option when an inner protocol sends a message to a sub-routine for the first time. This ensures that the time of precorruption is inherited when new sub-routines are spawned and thereby induces the natural behavior for PID-wise corruptions, i.e., that any sub-routine can be corrupted no later than the routine that spawned it. The initialized precorruption time

is allowed to be negative. This allows the environment to start the protocol in a state where some parties are precorrupted in the past, and hence be able to immediately corrupt these parties at the start of the protocol (similar to letting some parties be statically corrupted).

The shells that wrap the individual party's ITMs do not have access to query the ticker for the time, whereas the ideal shells can do this freely. We solve this by additionally letting the $\mathcal{D}_{\mathsf{Real}}$ spawn a *corruption-clock* (written $\mathcal{F}_{\text{CorruptionClock}}$) which exactly allows the shells to access time. Importantly, this does not reintroduce a global synchrony assumption as our shells prevent the inner protocols from communicating with the corruption clock. The corruption clock is therefore only an artifact of our modeling and will not appear when actually running the protocol.

Functionality $\mathcal{F}_{\text{CorruptionClock}}$

The functionality maintains a counter `Time`. Initially, `Time` $:= 0$.

Time?: When receiving ($\mathit{Time?}$) from a party $p_i \in \mathcal{P}$ it returns (Time, `Time`) to p_i.
Tick: It updates `Time` $:=$ `Time` $+ 1$.

When describing $\mathcal{D}_{\mathsf{Real}}$ we will leave out calls to $\mathcal{F}_{\text{CorruptionClock}}$ for brevity, but these happens each time the shell uses any notion of time.

We amend the corruption aggregation ITI presented in [8] to also make information about the precorruptions an adversary have used, available to the environment (and similarly the ideal functionalities). This prevents a simulator from using more precorruption tokens or corrupting faster than the real adversary.

Aside from ensuring the protocol corruption delays are respected the $\mathcal{D}_{\mathsf{Ideal}}$ additionally propagates both precorruption and corruption-tokens to the "inner functionality" (the functionality that the shell is a wrapper around). This is done in order to ensure that the simulator appended to the ideal functionalities can actually gain functionality-specific powers when performing a corruption. For example it might be that a certain channel does not need to respect delivery guarantees when the sender gets corrupted (for an example of this see Section 4.1).

Below we provide formal descriptions of both shells.

Function $\mathcal{D}^{\delta}_{\mathsf{Real}}(\pi)$

The shell wraps each party $p_i \in \mathcal{P}$ in a small wrapper that maintains a variable $\texttt{PrecorruptionTime}_i$. Initially, $\texttt{PrecorruptionTime}_i := \bot$. When receiving precorruptions and corruptions the wrapper has the behavior described below. The wrapper also filters out any communication with $\mathcal{F}_{\text{CorruptionClock}}$ and on all other inputs it simply forwards the inputs/outputs to/from the original protocol.

Initialization: If p_i receives $(\mathit{Initialize}, \tau)$ as the first message, then the party updates $\texttt{PrecorruptionTime}_i := \tau$ and if $\tau \neq \bot$ then also notifies the corruption-aggregation ITI.

Precorruption: If $p_i \in \mathcal{P}$ receives $\mathit{Precorrupt}$ at time τ, then the party first notifies the corruption-aggregation ITI by sending $(\mathit{Precorrupt}, p_i)$ to this machine. It then updates $\texttt{PrecorruptionTime}_i := \tau$.

Corruption: When p_i receives $\mathit{Corrupt}$ at time τ, then p_i checks if $\texttt{PrecorruptionTime}_i + \delta \leq \tau$. If that is not the case the request is ignored. Otherwise the party first notifies the corruption-aggregation ITI by sending $(\mathit{Corrupt}, p_i)$ to this machine and then it corrupts p_i by forwarding $\mathit{Corrupt}$ to π. Each time p_i is activated after this it sends its entire local state of the inner protocol to the adversary and furthermore forwards all messages m (assuming that m includes both content and recipient) that are written on the backdoor tape of p_i.

Whenever the shell of p_i detects that the inner protocol sends a message to a new sub-routine for the first time, it sends $(\mathit{Initialize}, \texttt{PrecorruptionTime}_i)$ to the subroutine before forwarding the message of the inner protocol.
Furthermore, the shell starts a separate corruption aggregation ITI. It maintains two lists `Precorrupted` and `Corrupted` that initially are both empty. The corruption aggregation ITI has the following behavior:

Precorruption Registration: When receiving a $(\mathit{Precorrupt}, p)$ from a party p it sets $\texttt{Precorrupted} := p :: \texttt{Precorrupted}$.

Corruption Registration: When receiving a $(\mathit{Corrupt}, p)$ from a party p it sets $\texttt{Corrupted} := p :: \texttt{Corrupted}$.

Corruption Status: When receiving $\mathit{CorruptionStatus}$ from the environment it queries all sub-functionalities of the protocol for their corruption status and updates the `Precorrupted` and `Corrupted`-lists accordingly. Finally, it sends $(\texttt{Precorrupted}, \texttt{Corrupted})$ back to the environment.

Function $\mathcal{D}^{\delta}_{\mathsf{Ideal}}(\mathcal{F})$

The shell wraps the functionality in a wrapper which maintains two lists `Precorrupted` and `Corrupted` that initially are both empty. Furthermore, it has a map $\texttt{PrecorruptionTimeMap} : \mathcal{P} \to \textsc{Time}$ and a counter to keep track of time `Time` which initially is instantiated to be 0. When receiving precorruptions, corruptions and corruption-status requests it has the following behavior and on all other inputs/outputs it forwards the inputs to/from $\mathcal{F}$.

Initialization: If the functionality receives $(\mathit{Initialize}, \tau)$ at the port belonging to p as the first message for this party, then the party updates $\texttt{PrecorruptionTimeMap}[p] := \tau$. If $\tau \neq \bot$ then it also updates

$\texttt{Precorrupted} := p :: \texttt{Precorrupted}$, and forwards $(\mathit{Initialize}, \tau)$ to the inner functionality.

Precorruption: When receiving $(\mathit{Precorrupt}, p)$ and p is a valid PID of a dummy party then it adds the current time, $\texttt{Time}$, to $\texttt{PrecorruptionTimeMap}[p] := \texttt{Time}$ and updates $\texttt{Precorrupted} := p :: \texttt{Precorrupted}$. Furthermore, it propagates $(\mathit{Precorrupt}, p)$ to $\mathcal{F}$.

Corruption: When receiving $(\mathit{Corrupt}, p)$ where p is a valid PID of a dummy party then the functionality checks if $\texttt{PrecorruptionTimeMap}[p] + \delta \leq \texttt{Time}$. If that is the case it updates $\texttt{Corrupted} := p :: \texttt{Corrupted}$ and returns to the adversary all the values received from p and output to p so far. From now on inputs from p are ignored but are instead given via the backdoor tape by the adversary. Furthermore, it propagates $(\mathit{Corrupt}, p)$ to $\mathcal{F}$.
If the request is send too early, it is ignored.

Inputs: If the functionality receives (Input, p, v) from the adversary and $p \in \texttt{Corrupted}$, then v is forwarded to $\mathcal{F}$ as if it was directly input by p to $\mathcal{F}$.

Corruption Status: When receiving $\mathit{CorruptionStatus}$ from the environment it sends $(\texttt{Precorrupted}, \texttt{Corrupted})$ back to the environment.

Tick: The functionality updates $\texttt{Time} := \texttt{Time} + 1$.

The additional (Input, p, v) command accepted by the ideal shell allows an adversary to input a message v on behalf of party p if p is corrupted. This follows how standard byzantine corruptions are treated and modelled in the UC framework.

We next formally define what it means for a protocol to securely implement a functionality against a δ-delayed adversary, see also Fig. 1 for a graphical depiction.

Definition 1 (UC-security against delayed adversaries). *Let $\delta \in \mathbb{N}$. We say that a protocol π securely implements an ideal functionality $\mathcal{F}$* against a δ-delayed adversary *when $\mathcal{D}^{\delta}_{\mathsf{Real}}(\pi)$ securely implements $\mathcal{D}^{\delta}_{\mathsf{Ideal}}(\mathcal{F})$ in the usual UC sense [8], i.e., if*

$$\forall \mathcal{A}\ \exists \mathcal{S}\ \forall \mathcal{Z}, \texttt{EXEC}(\mathcal{Z}, \mathcal{A}, \mathcal{D}^{\delta}_{\mathsf{Real}}(\pi)) \approx \texttt{EXEC}(\mathcal{Z}, \mathcal{S}, \mathcal{D}^{\delta}_{\mathsf{Ideal}}(\mathcal{F}))$$

where $\texttt{EXEC}(\mathcal{Z}, \cdot, \cdot)$ denotes the random variable describing the binary output of the environment $\mathcal{Z}$, and $\approx$ means that the statistical distance is negligible in the security parameter.

Note that security against a delayed adversary is defined both for functionalities that have special behavior defined for receiving precorruptions and functionalities that do not have any such behavior defined, as the default for protocols/functionalities is to ignore any unrecognized inputs.

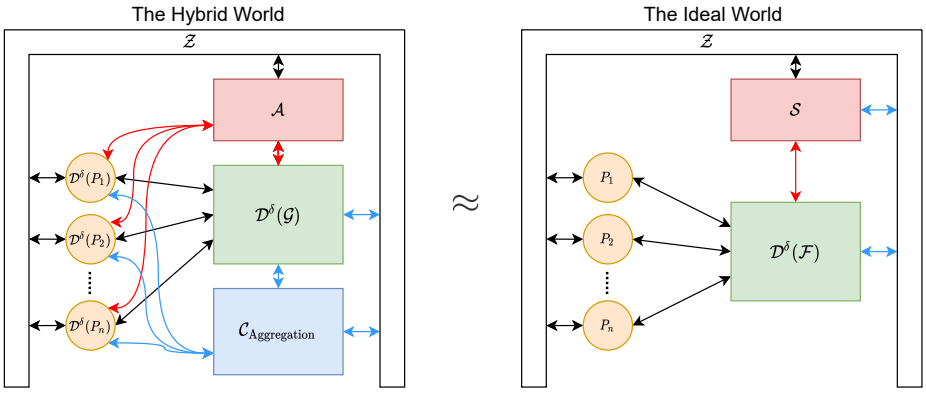

Fig. 1. A depiction of the security statement for a protocol that implements an ideal functionality $\mathcal{F}$ using the functionality $\mathcal{G}$ against a δ-delayed adversary.

3.2 Relating Corruption Models

In this section we relate the notion of a 0-delayed adversary to the standard notion of an adaptive adversary in UC. We further show that any protocol that is secure against a fast adversary is also secure against a slower adversary. These results allow us to reuse cryptographic constructions which are already proven secure modularly when implementing larger constructions.

Byzantine corruptions and 0*-delayed corruptions.* To showcase the generality of the δ-delayed corruption model, we relate this model to the standard model of adaptive Byzantine corruptions as defined in UC. To be able to precisely quantify how these notions relate, we introduce two Byzantine shells similar to the delay shells. The byzantine-shells are meant to precisely encapsulate the corruption model as presented in [8]. We believe that these are of independent interest as by using these it can be avoided to clutter the protocol and functionality description with a specific corruption model.

Function $\mathcal{B}_{\mathsf{Real}}(\pi)$

The shell adds the following behavior to each party $p_i \in \mathcal{P}$. If any other inputs are received than the ones below, it is the original code of the party that is executed.

Corruption: If $p_i \in \mathcal{P}$ receives *Corrupt* then the party first notifies the corruption-aggregation ITI by sending $(\mathit{Corrupt}, p_i)$ to this machine. Each time p_i is activated after this it sends its entire local state of the inner protocol to the adversary and furthermore forwards all messages m (assuming that m includes both content and recipient) that are written on the backdoor tape of p_i.

Furthermore the shell runs a separate corruption-aggregation ITI. It maintains a list `Corrupted` which initially is set to be the empty list and has the following behavior:

Registration: When receiving a $(\mathit{Corrupt}, p)$ from a party p it sets $\texttt{Corrupted} := p :: \texttt{Corrupted}$.

Corruption Status: When receiving $\mathit{CorruptionStatus}$ from the environment it queries all sub-functionalities of the protocol for their corruption status and updates the `Corrupted`-list accordingly. Finally, it sends `Corrupted` back to the environment.

Function $\mathcal{B}_{\mathsf{Ideal}}(\mathcal{F})$

The functionality maintains a list of corrupted parties, `Corrupted`, which initially is set to be the empty list. Upon receiving the following

Corruption: If the functionality receives $(\mathit{Corrupt}, p)$ from the adversary and p is a valid PID of the dummy parties, it updates $\texttt{Corrupted} := p :: \texttt{Corrupted}$ and returns to the adversary all the values received from p and output to p so far. From now on inputs from p are ignored but are instead given via the backdoor tape by the adversary. Furthermore it propagates $(\mathit{Corrupt}, p)$ to $\mathcal{F}$.

Inputs: If the functionality receives (Input, p, v) from the adversary and $p \in \texttt{Corrupted}$ then v is forwarded to $\mathcal{F}$ as if it was directly input by p to $\mathcal{F}$.

Corruption Status: When receiving $\mathit{CorruptionStatus}$ from the environment it sends `Corrupted` back to the environment.

Security against 0-delayed adversary implies security in the standard model and vice versa if the functionality that is implemented ignores precorruption and initialization tokens. We encapsulate this intuition in the theorem below.

Theorem 1. *Let π be a protocol and $\mathcal{F}$ an ideal functionality that ignores precorruptions and initializations. $\mathcal{B}_{\mathsf{Real}}(\pi)$ securely implements $\mathcal{B}_{\mathsf{Ideal}}(\mathcal{F})$ if and only if $\mathcal{D}^0_{\mathsf{Real}}(\pi)$ securely implements $\mathcal{D}^0_{\mathsf{Ideal}}(\mathcal{F})$.*

Formally,

$$\begin{aligned} &\forall \mathcal{A}\ \exists \mathcal{S}\ \forall \mathcal{Z}, \texttt{EXEC}(\mathcal{Z}, \mathcal{A}, \mathcal{B}_{\mathsf{Real}}(\pi)) \approx \texttt{EXEC}(\mathcal{Z}, \mathcal{S}, \mathcal{B}_{\mathsf{Ideal}}(\mathcal{F})) \\ &\iff \forall \mathcal{A}'\ \exists \mathcal{S}' \forall \mathcal{Z}', \texttt{EXEC}(\mathcal{Z}', \mathcal{A}', \mathcal{D}^0_{\mathsf{Real}}(\pi)) \approx \texttt{EXEC}(\mathcal{Z}', \mathcal{S}', \mathcal{D}^0_{\mathsf{Ideal}}(\mathcal{F})). \end{aligned} \tag{1}$$

Proof Sketch. We prove the two directions of the implication individually.

"$\Longrightarrow$": We let $\mathcal{A}'$ be any adversary and construct an adversary $\mathcal{A}$ by wrapping $\mathcal{A}'$ with a shell that forwards all inputs/outputs except precorruptions to/from $\mathcal{A}'$. Whenever $\mathcal{A}$ receives a $\mathit{Precorrupt}$ directed to p_i from $\mathcal{A}'$ it forwards

$(\mathit{Precorrupt}, p_i)$ to the environment instead. We now use the LHS of Eq. (1) to obtain a simulator $\mathcal{S}$ s.t.

$$\forall \mathcal{Z}, \mathtt{EXEC}(\mathcal{Z}, \mathcal{A}, \mathcal{B}_{\mathsf{Real}}(\pi)) \approx \mathtt{EXEC}(\mathcal{Z}, \mathcal{S}, \mathcal{B}_{\mathsf{Ideal}}(\mathcal{F})). \tag{2}$$

Given $\mathcal{S}$ we construct $\mathcal{S}'$ by running $\mathcal{S}$ inside $\mathcal{S}'$. Each time $\mathcal{S}$ outputs $(\mathit{Precorrupt}, p_i)$ to the environment then $\mathcal{S}'$ outputs $(\mathit{Precorrupt}, p_i)$ to $\mathcal{D}^0_{\mathsf{Ideal}}(\mathcal{F})$. All other inputs and outputs are forwarded to and from $\mathcal{S}$ directly. Note that precorruptions are ignored by $\mathcal{F}$ and therefore $\mathcal{F}$ does not change its behavior based upon these.
Let us now for the sake of contradiction assume that there exists some environment $\mathcal{Z}'$ that can distinguish against $\mathcal{A}'$ and $\mathcal{S}'$, i.e.,

$$\mathtt{EXEC}(\mathcal{Z}', \mathcal{A}', \mathcal{D}^0_{\mathsf{Real}}(\pi)) \not\approx \mathtt{EXEC}(\mathcal{Z}', \mathcal{S}', \mathcal{D}^0_{\mathsf{Ideal}}(\mathcal{F})) \tag{3}$$

Let us now show how to construct an environment, $\mathcal{Z}$, that can distinguish for the byzantine setting and thereby contradict Eq. (2).
We build $\mathcal{Z}$ by running $\mathcal{Z}'$ inside, and forward all inputs and outputs to $\mathcal{Z}'$. $\mathcal{Z}$ only deviates from $\mathcal{Z}'$ in the two cases below:

- Whenever a *CorruptionStatus* command is issued by $\mathcal{Z}'$ to the corruption aggregation ITI, we amend the answer with an additional list of precorruptions which we have received from $\mathcal{A}$ so far.
- Whenever a $(\mathit{Initialize}, \tau)$ command is send to some party it is not forwarded by $\mathcal{Z}$ but instead recorded as a precorruption of this party. This does not change the behavior of the protocol nor the ideal functionality as these are ignored.

In particular, $\mathcal{Z}$ simply forwards the guess on which world it is placed in from $\mathcal{Z}'$.
We observe that

$$\mathtt{EXEC}(\mathcal{Z}, \mathcal{S}, \mathcal{B}_{\mathsf{Ideal}}(\mathcal{F})) \approx \mathtt{EXEC}(\mathcal{Z}', \mathcal{S}', \mathcal{D}^0_{\mathsf{Ideal}}(\mathcal{F})), \tag{4}$$

and

$$\mathtt{EXEC}(\mathcal{Z}, \mathcal{A}, \mathcal{B}_{\mathsf{Real}}(\pi)) \approx \mathtt{EXEC}(\mathcal{Z}', \mathcal{A}', \mathcal{D}^0_{\mathsf{Real}}(\pi)). \tag{5}$$

Together with Eq. (3) this contradicts Eq. (2) and thus concludes the case.

"$\Longleftarrow$": The proof of this case mirrors the other case. We are now given $\mathcal{A}$ and construct $\mathcal{A}'$ by sending *Precorrupt*-tokens just before *Corrupt*-tokens. From the RHS of Theorem 1 we get a simulator $\mathcal{S}'$ which we use to construct $\mathcal{S}$ by forwarding everything except *Precorrupt*-tokens. Finally, we assume for the sake of contradiction that there exists a $\mathcal{Z}$ that is able to distinguish, build an environment $\mathcal{Z}'$ using this (removing *Precorrupt*-tokens and initializations), and derive a contradiction similarly to the other case. □

Note that the above theorem allows reusing constructions that are proven secure against a standard adaptive adversary when building complex systems that are to be secure against a 0-delayed adversary.

Lifting security to weaker adversaries. If protocols that are proven secure within different corruption models are composed, it gets hard to identify the final security guarantee that is provided by the composed construction. Intuitively, one would presume that a protocol that is proven secure against an adversary able to do "fast" corruptions is also secure against an adversary only able to do "slow" corruptions. Using precise shells to quantify corruption-speed allows us to capture this intuition in the lemma below.

Theorem 2 (Lifting Security to Slower Corruptions). *Let $\delta, \delta' \in \mathbb{N}$, s.t. $\delta \leq \delta'$, let π be a protocol, and let $\mathcal{F}$ be an ideal functionality. If $\mathcal{D}^{\delta}_{\mathsf{Real}}(\pi)$ securely implements $\mathcal{D}^{\delta}_{\mathsf{Ideal}}(\mathcal{F})$, then $\mathcal{D}^{\delta'}_{\mathsf{Real}}(\pi)$ securely implements $\mathcal{D}^{\delta'}_{\mathsf{Ideal}}(\mathcal{F})$.*

Formally,

$$\begin{aligned} &\forall \mathcal{A}, \exists \mathcal{S}, \forall \mathcal{Z}, \mathtt{EXEC}(\mathcal{Z}, \mathcal{A}, \mathcal{D}^{\delta}_{\mathsf{Real}}(\pi)) \approx \mathtt{EXEC}(\mathcal{Z}, \mathcal{S}, \mathcal{D}^{\delta}_{\mathsf{Ideal}}(\mathcal{F})) \\ &\implies \forall \mathcal{A}', \exists \mathcal{S}', \forall \mathcal{Z}', \mathtt{EXEC}(\mathcal{Z}', \mathcal{A}', \mathcal{D}^{\delta'}_{\mathsf{Real}}(\pi)) \approx \mathtt{EXEC}(\mathcal{Z}', \mathcal{S}', \mathcal{D}^{\delta'}_{\mathsf{Ideal}}(\mathcal{F})). \end{aligned} \tag{6}$$

Proof. Let H be the hypothesis (LHS of the implication), and let $\mathcal{A}'$ be an adversary. We define $\mathsf{Filter}(\mathcal{A}, \delta)$ to be a wrapper around an adversary that simply filters out corruption request that are to early w.r.t. δ.

Using H we know that there exists a simulator $\mathcal{S}$ s.t.

$$\forall \mathcal{Z}, \mathtt{EXEC}(\mathcal{Z}, \mathsf{Filter}(\mathcal{A}', \delta'), \mathcal{D}^{\delta}_{\mathsf{Real}}(\pi)) \approx \mathtt{EXEC}(\mathcal{Z}, \mathcal{S}, \mathcal{D}^{\delta}_{\mathsf{Ideal}}(\mathcal{F})). \tag{7}$$

Let us now show,

$$\forall \mathcal{Z}, \mathtt{EXEC}(\mathcal{Z}, \mathsf{Filter}(\mathcal{A}', \delta'), \mathcal{D}^{\delta}_{\mathsf{Real}}(\pi)) \approx \mathtt{EXEC}(\mathcal{Z}, \mathcal{S}, \mathcal{D}^{\delta'}_{\mathsf{Ideal}}(\mathcal{F})). \tag{8}$$

Assume for the sake of contradiction that there exists an environment $\mathcal{Z}$ that is able to distinguish in Eq. (8). We use this to build an environment $\mathcal{Z}'$ which is able to distinguish in Eq. (7) with at least as big an advantage. $\mathcal{Z}'$ works by forwarding everything to and from $\mathcal{Z}$. Except if at any point in time there is a *Precorrupt*-token followed by a *Corrupt* send with strictly less than δ' between them, then $\mathcal{Z}'$ immediately guesses that it is in the ideal case.

As every time that this happens the environment is correct, and every time this does not happen the execution is exactly similar to that of Eq. (7) this implies Eq. (8).

We now define $\mathcal{S}' \triangleq \mathcal{S}$ and let $\mathcal{Z}'$ be any environment. We specialize Eq. (8) with $\mathcal{Z}'$ and obtain

$$\mathtt{EXEC}(\mathcal{Z}', \mathsf{Filter}(\mathcal{A}', \delta'), \mathcal{D}^{\delta}_{\mathsf{Real}}(\pi)) \approx \mathtt{EXEC}(\mathcal{Z}', \mathcal{S}, \mathcal{D}^{\delta'}_{\mathsf{Ideal}}(\mathcal{F})). \tag{9}$$

Furthermore,

$$\mathtt{EXEC}(\mathcal{Z}', \mathsf{Filter}(\mathcal{A}', \delta'), \mathcal{D}^{\delta}_{\mathsf{Real}}(\pi)) \approx \mathtt{EXEC}(\mathcal{Z}', \mathsf{Filter}(\mathcal{A}', \delta'), \mathcal{D}^{\delta'}_{\mathsf{Real}}(\pi)) \tag{10}$$

$$\approx \mathtt{EXEC}(\mathcal{Z}', \mathcal{A}', \mathcal{D}^{\delta'}_{\mathsf{Real}}(\pi)). \tag{11}$$

Eq. (10) holds as if early corruptions are ignored, then $\mathcal{D}^{\delta}_{\mathsf{Real}}(\pi)$ and $\mathcal{D}^{\delta'}_{\mathsf{Real}}(\pi)$ are identically distributed. Eq. (11) holds as it is not observable by the environment if the corruption is ignored by the filter or the shell. Together Eqs. (9) and (11) finishes the proof. □

Note that if one considered a simpler model with just one corruption token and a subsequent automatic effectuation of the corruption a certain time after such the token was input (instead of a model like ours with separate tokens for precorruptions and corruptions), then Theorem 2 would not hold. The reason is that in such a model a fast adversary would not have the ability to imitate a slow adversary. Hence, in such a model a fast adversary would not be strictly "stronger" than a slow adversary.

Theorems 1 and 2 together imply that any protocol that is secure against a standard adaptive adversary in UC, is also secure against any δ-delayed adversary.

4 Functionalities

In this section we define a time-bounded channel between parties as well as a flooding functionality. The functionalities that we present are:

MessageTransfer: A functionality that allows one party to send messages to another party. This is modeling a point-to-point channel.

Flood: A functionality that allows all honest parties to disseminate to all other parties.

Conventions for ideal functionalities. Our functionalities needs to maintain a counter which is incremented each time a tick happens (similarly to what $\mathcal{D}_{\mathsf{Ideal}}$ does). For clarity of presentation, we describe our functionalities without explicitly mentioning this, but instead describe them as having direct access to time. Furthermore, we define the functionalities without specifying the corruption model as we will make use of the shells described in Section 3 to make the corruption-model explicit when implementing the functionalities.

Additionally, the behavior of both our functionalities depend on which parties are precorrupted and which parties are corrupted. Therefore they both maintain two sets: `Precorrupted` and `Corrupted` which are initially empty. These are updated by the following activation rules which we do not make explicit in the functionalities below for clarity of presentation.

Precorrupt: Upon receiving $(\mathit{Precorrupt}, p_i)$ or an initialization that changes party p_i's status to precorrupted, it sets $\texttt{Precorrupted} := \texttt{Precorrupted} \cup \{p_i\}$.

Corrupt: Upon receiving $(\mathit{Corrupt}, p_i)$ it sets $\texttt{Corrupted} := \texttt{Corrupted} \cup \{p_i\}$.

Furthermore, both of our ideal functionalities are parameterized by a type of messages that can be propagated which we denote MESSAGES.

4.1 MessageTransfer

In this section we present a basic functionality that allows a party to send messages to other parties. This is similar to the point-to-point channel presented in [4], but instead of hardcoding whether we assume AMS (as done in [4]) or not, we introduce an additional parameter which is the time an honest party needs to stay honest for ensuring delivery of the message.

Functionality $\mathcal{F}^{\sigma,\Delta}_{\mathsf{MessageTransfer}}(p_s, p_r)$

The functionality is parameterized by two parties p_s (the sender) and p_r (the receiver), and a time σ which parties needs to stay honest for the delivery guarantee Δ to apply. It maintains a mailbox for p_r, $\mathsf{Mailbox} : \textsc{Messages}$.

Initialize: Initially, $\mathsf{Mailbox} := \varnothing$.
Send: After receiving (Send, m) from p_s it leaks (Leak, p_s, m) to the adversary.
Get Messages: After receiving $(\mathit{GetMessages})$ from p_r it outputs $\mathsf{Mailbox}$ to party p_r.
Set Message: After receiving $(\mathit{SetMessage}, m)$ from the adversary, the functionality sets $\mathsf{Mailbox} := \mathsf{Mailbox} \cup \{m\}$.

At any time the functionality automatically enforces the following property:

1. Let m be a message that is input for the first time by an honest party $p_s \notin \texttt{Corrupted}$ at some time τ. If $p_s \notin \texttt{Corrupted}$ at time $\tau + \sigma$, then by time $\tau + \Delta$ it is ensured that $m \in \mathsf{Mailbox}$.

The property is ensured by the functionality automatically making the minimal possible additional calls with $\mathit{SetMessage}$.

Note that building a construction using $\mathcal{F}^{0,\Delta}_{\mathsf{MessageTransfer}}$ exactly corresponds to assuming AMS whereas assuming $\mathcal{F}^{\sigma,\Delta}_{\mathsf{MessageTransfer}}$ against a δ-delayed adversary with $\delta < \sigma$ corresponds to not assuming AMS.

4.2 Flood

The ideal functionality that we present here provides the guarantees of flooding network, i.e., that all information some honest party knows is disseminated to all other parties within a bounded time.

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

The functionality is parameterized by a set of parties $\mathcal{P}$, and a delivery guarantee Δ.

Furthermore, it keeps track of a set of messages for each party $\mathsf{Mailbox} : \mathcal{P} \to \textsc{Messages}$. These sets contain the messages that each party will receive after fetching.

Initialize: Initially, $\texttt{Corrupted} := \varnothing$ and $\mathsf{Mailbox}[p_i] := \varnothing$ for all $p_i \in \mathcal{P}$.
Send: After receiving (Send, m) from p_i it leaks (Leak, p_i, m) to the adversary.

Get Messages: After receiving ($\mathit{GetMessages}$) from p_i it outputs $\mathsf{Mailbox}[p_i]$ to party p_i.
Set Message: After receiving ($\mathit{SetMessage}, m, p_i$) from the adversary, the functionality sets $\mathsf{Mailbox}[p_i] \coloneqq \mathsf{Mailbox}[p_i] \cup \{m\}$.

At any time after all parties have been initialized the functionality automatically enforces the following two properties:

1. Let m be a message that is input for the first time to an honest party $p_i \notin \texttt{Precorrupted} \cup \texttt{Corrupted}$ at some time τ. By time $\tau + \Delta$ it is ensured that $\forall p_j \in \mathcal{P} \setminus (\texttt{Corrupted} \cup \texttt{Precorrupted})$ it holds that $m \in \mathsf{Mailbox}[p_j]$.
2. Let m be a message at some time τ is in the mailbox of an honest party $p_i \notin \texttt{Precorrupted} \cup \texttt{Corrupted}$ i.e., $m \in \mathsf{Mailbox}[p_i]$. By time $\tau + \Delta$ it is distributed to all honest mailboxes, i.e., for any party $p_j \in \mathcal{P} \setminus (\texttt{Corrupted} \cup \texttt{Precorrupted})$ it holds that $m \in \mathsf{Mailbox}[p_j]$.

The properties are ensured by the functionality automatically making the minimal possible additional calls with $\mathit{SetMessage}$.

5 Implementations of Flood

In this section we will present the following protocols that implement Flood:

$\pi_{\mathsf{NaiveFlood}}$**:** Everybody simply sends to everybody.
π_{ERFlood}**:** Everybody sends to each other party with some fixed probability ρ.

We provide two types of implementations for Flood. A naive approach where everybody sends to everybody and a more efficient one where each party sends to their neighbors with probability ρ. The latter construction allows us to reuse the theoretic foundation of Erdős–Rényi graphs in the distributed systems setting and achieve a variety of properties.

5.1 Naive Flood

We present here a protocol that implements Flood with a message complexity that is quadratic in the number of messages that is input to the system.

The protocol $\pi_{\mathsf{NaiveFlood}}$ works straightforwardly by a peer sending and relaying any non-relayed message to all other parties. As everybody sends to everybody the protocol achieves a very small diameter and resilience against *fairly fast* adaptive adversaries at the cost of a large communication overhead and neighborhood.

Protocol $\pi_{\mathsf{NaiveFlood}}$

Each pair of parties $p_i, p_j \in \mathcal{P}$ has access to a channel $\mathcal{F}^{\sigma,\Delta}_{\mathsf{MessageTransfer}}(p_i, p_j)$. Each party $p_i \in \mathcal{P}$ keeps track of a set of relayed messages $\texttt{Relayed}_i$.

Initialize: Initially, all parties initialize their channel between them and set $\texttt{Relayed}_i := \varnothing$.

Send: When p_i receives (Send, m) they now forward inputs (Send, m) to $\mathcal{F}^{\sigma,\Delta}_{\mathsf{MessageTransfer}}(p_i, p_j)$ for all $p_j \in \mathcal{P}$ and set $\texttt{Relayed}_i := \texttt{Relayed}_i \cup \{m\}$.

Get Messages: When p_i receives $(\mathit{GetMessages})$ they let M be the union of the messages they achieve by calling $(\mathit{GetMessages})$ to $\mathcal{F}^{\sigma,\Delta}_{\mathsf{MessageTransfer}}(p_i, p_j)$ for all $p_j \in \mathcal{P}$, and outputs M.

Furthermore, once in every activation each honest p_i let M be the union of the messages they achieve by calling $(\mathit{GetMessages})$ to $\mathcal{F}^{\sigma,\Delta}_{\mathsf{MessageTransfer}}(p_i, p_j)$. For any $m \in M \setminus \texttt{Relayed}_i$, p_i inputs (Send, m) to $\mathcal{F}^{\sigma,\Delta}_{\mathsf{MessageTransfer}}(p_i, p_j)$ for all $p_j \in \mathcal{P}$, and sets $\texttt{Relayed}_i := \texttt{Relayed}_i \cup \{m\}$.

An obvious attack on this protocol an adversary might try to perform is to try to corrupt the sender between the time τ that a message is sent and time $\tau + \sigma$ where the delivery guarantee from the underlying $\mathcal{F}^{\sigma,\Delta}_{\mathsf{MessageTransfer}}$ applies. An adversary that succeeds with this can violate both Properties 1 and 2 of $\mathcal{F}^{\Delta}_{\mathsf{Flood}}$. However, σ-delayed adversaries do not have sufficient time to succeed with this as the properties only needs to be upheld for parties that are neither corrupted nor precorrupted when they try to send the message. Below we explicitly[3] prove that against such adversaries the naive protocol actually realizes $\mathcal{F}^{\Delta}_{\mathsf{Flood}}$.

Lemma 1. *Let $\sigma, \Delta \in \mathbb{N}$. The protocol $\pi_{\mathsf{NaiveFlood}}$ perfectly realizes $\mathcal{F}^{\Delta}_{\mathsf{Flood}}$ in the $\mathcal{F}^{\sigma,\Delta}_{\mathsf{MessageTransfer}}$-hybrid model against a σ-delayed adversary.*

Proof. We construct a simulator $\mathcal{S}$.

1. $\mathcal{S}$ simulates all parties $p_i \in P$ inside it self.
2. When receiving (Leak, p_i, m) from $\mathcal{F}^{\Delta}_{\mathsf{Flood}}$ the simulator inputs (Send, m) to p_i (running inside $\mathcal{S}$).
3. When receiving $(\mathit{SetMessage}, m)$ from the adversary on the port belonging to functionality $\mathcal{F}^{\sigma,\Delta}_{\mathsf{MessageTransfer}}(p_i, p_j)$, $\mathcal{S}$ forwards $(\mathit{SetMessage}, m, p_j)$ to $\mathcal{F}^{\Delta}_{\mathsf{Flood}}$.
4. Whenever $\mathcal{A}$ corrupts some $p_i \in \mathcal{P}$, $\mathcal{S}$ corrupts p_i and sends the simulated internal state to $\mathcal{A}$. From then on the simulated p_i (inside $\mathcal{S}$) follows $\mathcal{A}$'s instructions.

[3] In [37, Chapter 3, p. 111], it is shown that it is enough to argue correct realization to achieve secure realization for any protocol which leaks all I/O behavior to the adversary. One may be lead to believe that this result directly applies to $\pi_{\mathsf{NaiveFlood}}$, but as $(\mathit{GetMessages})$ inputs (and corresponding outputs) are hidden from the adversary this is not the case.

5. Whenever the $\bar{\mathcal{G}}_{\text{Ticker}}$ notifies $\mathcal{S}$ about the passing of time, $\mathcal{S}$ ensures to activate $\mathcal{F}^{\Delta}_{\mathsf{Flood}}$.

As protocol, functionality, and simulator are all deterministic it is enough to argue that the I/O behavior of $\mathcal{A}$ interacting with $\pi_{\mathsf{NaiveFlood}}$ is equal to the I/O behavior of $\mathcal{S}$ interacting with $\mathcal{F}^{\Delta}_{\mathsf{Flood}}$ to argue perfect indistinguishability. The send command is invoked at the exact same times in the real execution and in the execution inside $\mathcal{S}$ this produces the exact same behavior. Furthermore, for any send command that is invoked at time τ by an honest party (neither precorrupted nor corrupted) there will be a set-message command within $\tau + \Delta$ for all honest parties in the real protocol as a σ-delayed adversary does not have time to violate the delivery property of the underlying $\mathcal{F}^{\sigma,\Delta}_{\mathsf{MessageTransfer}}(p_i, p_j)$, and therefore Property 1 is upheld. Similarly, the relaying of messages in the real protocol ensure that messages will be delivered by the adversary according to the properties of $\mathcal{F}^{\Delta}_{\mathsf{Flood}}$ in the real protocol (inside $\mathcal{S}$ and therefore also in the ideal) which ensures Property 2. □

5.2 Efficient Flood

We now present a more efficient version of Flood. The idea is simple: Instead of relaying messages to *all* parties, each party flips a coin for each neighbor that decides if a particular message should be relayed to this party. Compared to the naive implementation of Flood presented in previous section the protocol presented here will have significantly smaller neighborhoods at the cost of larger diameter in the communication graph (the parameter Δ of Flood). Furthermore, the construction is only able to tolerate adversaries that are slightly more delayed than those the naive protocol can tolerate.

The protocol π_{ERFlood} works by letting all parties relay and send messages to a different random subset of parties for each message that is to be sent/relayed. By letting the random subset be large enough we ensure that we establish a connected graph with low diameter for all messages. As the subset of parties each party chooses to send to is random, the protocol achieves quite some robustness against adaptive adversaries, as a slightly delayed adversary cannot predict whom to corrupt in order to eclipse some specific parties.

Protocol $\pi_{\mathsf{ERFlood}}(\rho)$

Each pair of parties $p_i, p_j \in \mathcal{P}$ has access to a channel $\mathcal{F}^{\sigma,\Delta}_{\mathsf{MessageTransfer}}(p_i, p_j)$. Each party $p_i \in \mathcal{P}$ keeps track of a set of relayed messages $\texttt{Relayed}_i$: Messages.

Initialize: Initially, all parties initialize their channel between them and set $\texttt{Relayed}_i := \varnothing$.

Send: When p_i receives (Send, m), they input (Send, m) to $\mathcal{F}^{\sigma,\Delta}_{\mathsf{MessageTransfer}}(p_i, p_j)$ with probability ρ for each party $p_j \in \mathcal{P}$. Finally they set $\texttt{Relayed}_i := \texttt{Relayed}_i \cup \{m\}$.

Get Messages: When p_i receives (*GetMessages*) they let M be the union of the messages they achieve by calling (*GetMessages*) to $\mathcal{F}^{\sigma,\Delta}_{\mathsf{MessageTransfer}}(p_i, p_j)$ for all $p_j \in \mathcal{P}$, and outputs M.

Furthermore, once in each activation each honest p_i let M be the union of the messages they achieve by calling (*GetMessages*) to $\mathcal{F}^{\sigma,\Delta}_{\mathsf{MessageTransfer}}(p_i, p_j)$ for all $p_j \in \mathcal{P}$. For any $m \in M \setminus \texttt{Relayed}_i$, p_i inputs (*Send*, m) to $\mathcal{F}^{\sigma,\Delta}_{\mathsf{MessageTransfer}}(p_i, p_j)$ with probability ρ for all $p_j \in \mathcal{P}$, and sets $\texttt{Relayed}_i \coloneqq \texttt{Relayed}_i \cup \{m\}$.

Depending on the parameter ρ the protocol π_{ERFlood} can achieve a variety of properties. We provide two different instantiations that uses the channel $\mathcal{F}^{\sigma,\Delta}_{\mathsf{MessageTransfer}}$ and all works against a $(\sigma + \Delta)$-delayed adversary. Before going into detail with the actual proof, we provide some intuition for why the protocol is secure against exactly a $(\sigma + \Delta)$-delayed adversary. The main intuition is that such an adversary cannot influence how the communication graph between the parties that are honest are created. If a party decides to send a message at some time τ then the set of parties that receives this message will have completed forwarding the message at time $\tau + \sigma + \Delta$, which is the earliest point on this party can be corrupted based upon this party's role in the specific communication graph. Therefore an adversary cannot make use of the adaptive corruptions to disrupt the propagation of a message.

Each of the instantiations that are presented below provides a trade-off between the diameter of the graph, the average size of the neighborhood and the probability that the graph in fact has these properties. Instantiation 1 ensures a diameter of 2 with a neighborhood of just $\Omega(\sqrt{n\kappa})$ and Instantiation 2 ensures a logarithmic diameter with a neighborhood of average size $\Omega(\kappa)$.

Theorem 3. *Let $\Delta \in \mathbb{N}$ be any delay, let $\sigma \in \mathbb{N}$, let $t < n$ be the maximum number of parties an adversary can corrupt, and let $\kappa \in \mathbb{R}$ be the security parameter. The protocol $\pi_{\mathsf{ERFlood}}(\rho)$ securely implements $\mathcal{F}^{\Delta'}_{\mathsf{Flood}}$ against a $(\sigma + \Delta)$-delayed adversary using $\mathcal{F}^{\sigma,\Delta}_{\mathsf{MessageTransfer}}$. More precisely when r is an upper bound on the number of different messages input (either via Send or via SetMessage), the statistical distance between the real and ideal executions is bounded by the probability p_{bad} for either of the following instantiations:*

1. *Let $\rho \coloneqq \sqrt{\frac{\kappa}{h}}$ and let $\Delta' \coloneqq 2\Delta$ then*

$$p_{bad} \leq r \cdot (t+1) \cdot n^2 \cdot e^{-\kappa \cdot \frac{(h-2)}{h}}. \tag{12}$$

2. *Let $\alpha \in \mathbb{R}$, $\gamma, \delta_1, \delta_2 \in [0,1]$, and $\rho \coloneqq \frac{\kappa}{h}$. Furthermore, let $t_0 \coloneqq \frac{\log\left(\frac{\gamma n}{(1-\delta_1)\kappa}\right)}{\log((1-\delta_2)\alpha)} + 1$ and $\Delta' \coloneqq \Delta \cdot (t_0 + 1)$. If*

$$e^{-\kappa\gamma} + \frac{\gamma\alpha}{1-\gamma} \leq 1, \quad \frac{\gamma n}{(1-\delta_1)\kappa} > 1, \quad \text{and} \quad (1-\delta_2) \cdot \alpha > 1, \tag{13}$$

then

$$p_{bad} \leq r \cdot (t+1) \cdot \left(n \cdot \left(e^{-\frac{\delta_1^2 \kappa}{2}} + t_0 e^{-\frac{\delta_2^2 \alpha (1-\delta_1) \kappa}{2}} \right) + e^{-h \cdot (\kappa \gamma^2 - 2)} \right). \quad (14)$$

Proof Sketch. For an adversary we construct a simulator similar to how it is done in the proof of Lemma 1. The only times this is not a perfect simulation is when one of the properties of $\mathcal{F}_{\mathsf{Flood}}^{\Delta'}$ are violated in π_{ERFlood} which will never happen when the environment interacts with $\mathcal{F}_{\mathsf{Flood}}^{\Delta'}$. The main idea of the proof is to argue about the probability that a message m, that is input via either *Send* or *SetMessage*, is not propagated to all parties within Δ' time. We will argue about this via 7 random experiments:

$\mathsf{FloodToER}_1$: An experiment where an adversary interacts with an oracle to learn edges in a directed graph. Only nodes that have an edge to them can have their edges revealed to the adversary but the adversary can inject additional edges in order to be able to reveal more nodes. The adversary has the possibility to remove up to t nodes, but at the point of removal the adversary cannot have learned any edges connecting to the removed node. If at any point there is a cut in the graph the adversary can stop the game.

$\mathsf{FloodToER}_2$: An experiment similar to $\mathsf{FloodToER}_1$ except now the edges are undirected.

$\mathsf{FloodToER}_3$: An experiment similar to $\mathsf{FloodToER}_2$ except the adversary cannot stop the game before all parties have been revealed.

$\mathsf{FloodToER}_4$: An experiment similar to $\mathsf{FloodToER}_3$ except the adversary cannot inject edges between parties.

$\mathsf{FloodToER}_5$: An experiment similar to $\mathsf{FloodToER}_4$ except that the oracle secretly and uniformly predetermines the size of the returned graph, $s \in \{h, \ldots, n\}$. The adversary can however still decide whether or not to remove a particular node given that it does not violate the size that the oracle has determined.

$\mathsf{FloodToER}_6$: An experiment similar to $\mathsf{FloodToER}_5$ except now the oracle also predetermines a Erdős–Rényi graph of the predetermined size and embeds this into the final graph that is returned.

Erdős–Rényi: An experiment that chooses a graph of a certain size and includes each edge independently with probability ρ.

Let $d \coloneqq \frac{\Delta'}{\Delta}$. We now argue via the following steps:

1. If there is an adversary that prevents timely delivery of m in the real world with some probability, then there exists an adversary that can make $\mathsf{FloodToER}_1$ return a graph where the distance from the sender to some node is larger than d with at least as high a probability.
2. If any adversary can make $\mathsf{FloodToER}_1$ return a graph with a diameter larger than d with probability p, then there exists some adversary that can make $\mathsf{FloodToER}_2$ return a graph where the distance from the sender to some node is larger than d with at least as high a probability.

3. If any adversary can make $\mathsf{FloodToER}_2$ return a graph with a diameter larger than d with probability p, then there exists some adversary that can make $\mathsf{FloodToER}_3$ return a graph where the distance from the sender to some node is larger than d with at least as high a probability.
4. If any adversary can make $\mathsf{FloodToER}_3$ return a graph with a diameter larger than d with probability p, then there exists some adversary that can make $\mathsf{FloodToER}_4$ return a graph with a diameter larger than d with at least as high a probability.
5. If any adversary can make the $\mathsf{FloodToER}_4$ game return a graph with a diameter larger than d with probability p, then the same adversary can make $\mathsf{FloodToER}_5$ return a graph with a diameter larger than d with probability at least $p \cdot (t+1)$.
6. The experiments $\mathsf{FloodToER}_5$ and $\mathsf{FloodToER}_6$ are distributed identically.
7. The probability that $\mathsf{FloodToER}_6$ returns a graph with larger diameter than d must be less than the probability that an Erdős–Rényi graph with the *worst* size has a larger diameter than d.
8. We can now use the Erdős–Rényi graph results to bound the probability that an adversary can prevent the delivery of m in the real world.

We finally do a union bound over the number of different messages that is input to the functionality. The detailed proof can be found in the full version [34]. □

As the results in Theorem 3 are hard to interpret we additionally provide the following corollary which instantiates some of the many constants and makes some simplifying but non-optimal estimates. We emphasize that if one wants to optimize for a particular use-case (i.e., small diameter or very small failure probability) then Theorem 3 can be used to obtain tighter bounds.

Corollary 1. *Let $\Delta \in \mathbb{N}$ be any delay, let $\sigma \in \mathbb{N}$, let $t < n$ be the maximum number of parties an adversary can corrupt, and let $\kappa \in \mathbb{R}$ be the security parameter. The protocol $\pi_{\mathsf{ERFlood}}(\rho)$ securely implements $\mathcal{F}_{\mathsf{Flood}}^{\Delta'}$ against a $(\sigma+\Delta)$-delayed adversary using $\mathcal{F}_{\mathsf{MessageTransfer}}^{\sigma,\Delta}$. More precisely when r is an upper bound on the number of different messages input (either via Send or via SetMessage), the statistical distance between the real and ideal executions is bounded by the probability p_{bad} for either of the following instantiations:*

1. *Let $\rho := \sqrt{\frac{\kappa}{h}}$ and let $\Delta' := 2\Delta$ then*

$$p_{bad} \leq r \cdot (t+1) \cdot n^2 \cdot e^{-\kappa \cdot \frac{(h-2)}{h}}. \tag{15}$$

2. *Let $\rho := \frac{\kappa}{h}$, and $\Delta' := \Delta \cdot (5 \log\left(\frac{n}{2\kappa}\right) + 2)$, if $\frac{n}{2\kappa} > 1$ then*

$$p_{bad} \leq r \cdot (t+1) \cdot \left(7n \log\left(\frac{n}{2\kappa}\right) e^{-\frac{\kappa}{18}} + e^{-\frac{h(\kappa-18)}{9}}\right). \tag{16}$$

Proof. Instantiation 1 immediately follows from Theorem 3 (Instantiation 1). To derive instantiation Instantiation 2 we again use Theorem 3 (Instantiation 2) and select

$$\delta_1 := \delta_2 := \gamma := \frac{1}{3} \quad \text{and} \quad \alpha := \frac{7}{4}.$$

With these parameters we see that Eq. (13) is fulfilled when $\kappa \geq 1$. Furthermore, we see that

$$\begin{aligned} p_{\text{bad}} &\leq r \cdot (t+1) \cdot \left(n \cdot \left(e^{-\frac{\kappa}{18}} + \left(5 \log\left(\frac{n}{2\kappa} \right) + 1 \right) e^{-\frac{7\kappa}{108}} \right) + e^{-\frac{h(\kappa-18)}{9}} \right) \\ &\leq r \cdot (t+1) \cdot \left(7n \log\left(\frac{n}{2\kappa} \right) e^{-\frac{\kappa}{18}} + e^{-\frac{h(\kappa-18)}{9}} \right). \end{aligned}$$ □

The number of neighbors any party will need to send to when they send/relay a message in $\pi_{\mathsf{ERFlood}}(\rho)$ concentrates around $n \cdot \rho$ (this follows from the Chernoff bound and the union bound). Concretely, for Instantiation 1 we get that the number of neighbors is upper-bounded by $\mathcal{O}(\sqrt{\kappa n})$ except with a negligible probability, and for Instantiation 2 we get that the number of neighbors is upper-bounded by $\mathcal{O}(\kappa)$ except with a negligible probability.

A note on changing from TCP to UDP. Results about Erdős–Rényi graphs can be transferred to a setting without reliable message-transmission. Let us, instead of reliable transmission assume that there is an independent failure probability β for each message that is send via $\mathcal{F}_{\mathsf{MessageTransfer}}$ and ρ is an instantiation of $\pi_{\mathsf{ERFlood}}(\rho)$ that ensures a certain diameter assuming reliable transfer. If we let $\rho' \coloneqq \frac{\rho}{1-\beta}$ then $\pi_{\mathsf{ERFlood}}(\rho')$ with unreliable transfer is ensured to have the same diameter as $\pi_{\mathsf{ERFlood}}(\rho)$ with reliable transfer. This is because that the probability for a successful propagation from party p_i to p_j will then be $\rho' \cdot (1-\beta) = \rho$, which ensures that we in this more difficult setting inherent the original results for $\pi_{\mathsf{ERFlood}}(\rho)$.

6 Conclusion and Future Work

We have formally defined the model of δ-delayed adversaries within the UC framework. This has allowed us to precisely characterize and prove the security guarantees of the flooding protocol, π_{ERFlood}. Thereby, we have taken a first step at putting the widely assumed flooding functionalities on firm ground.

Several interesting directions for future work remain. In this work, we have explored a particular type of flooding protocol based upon Erdős–Rényi graphs. However, as discussed earlier, there exist several more complex constructions for different gossip networks in the literature. Analyzing such protocols against δ-delayed corruptions could potentially yield protocols that are even more efficient than what is presented here while also providing a well-understood security guarantee. Another direction could be to optimize for security instead of efficiency. The flooding protocol that we have presented is only secure against adversaries that are delayed by at least $(\sigma + \Delta)$. An interesting question that arises is whether this is inherent for flooding networks, or whether it is possible to implement a flooding network that is secure against a 0-delayed adversary.

Furthermore, this work has considered adaptive but not *mobile* adversaries, which can again uncorrupt parties. For some notion of mobility, it seems that π_{ERFlood} could be secure even in the presence of such mobile adversaries. Extending

the model of δ-delayed adversaries to include some notion of mobility would be useful in order to better understand guarantees of blockchain protocols that are supposed to run for a very long time.

Acknowledgements. We thank Ran Canetti for explaining a subtle detail of the UC framework, Sabine Oechsner for discussions in the initial phase of the project, and the anonymous reviewers of Eurocrypt and Crypto for their feedback.

References

1. Abraham, I., Chan, T.H., Dolev, D., Nayak, K., Pass, R., Ren, L., Shi, E.: Communication complexity of byzantine agreement, revisited. In: PODC. pp. 317–326. ACM (2019)
2. Badertscher, C., Canetti, R., Hesse, J., Tackmann, B., Zikas, V.: Universal composition with global subroutines: Capturing global setup within plain UC. In: TCC (3). Lecture Notes in Computer Science, vol. 12552, pp. 1–30. Springer (2020)
3. Badertscher, C., Maurer, U., Tschudi, D., Zikas, V.: Bitcoin as a transaction ledger: A composable treatment. In: CRYPTO (1). Lecture Notes in Computer Science, vol. 10401, pp. 324–356. Springer (2017)
4. Baum, C., David, B., Dowsley, R., Nielsen, J.B., Oechsner, S.: TARDIS: A foundation of time-lock puzzles in UC. In: EUROCRYPT (3). Lecture Notes in Computer Science, vol. 12698, pp. 429–459. Springer (2021)
5. Birman, K.P., Hayden, M., Özkasap, Ö., Xiao, Z., Budiu, M., Minsky, Y.: Bimodal multicast. ACM Trans. Comput. Syst. **17**(2), 41–88 (1999). https://doi.org/10.1145/312203.312207
6. Bollobás, B.: Random graphs. No. 73 in Cambridge studies in advanced mathematics, Cambridge University Press, 2 edn. (2001)
7. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd FOCS. pp. 136–145. IEEE Computer Society Press, Las Vegas, NV, USA (Oct 14–17, 2001). https://doi.org/10.1109/SFCS.2001.959888
8. Canetti, R.: Universally composable security. J. ACM **67**(5), 28:1–28:94 (2020)
9. Canetti, R., Dodis, Y., Pass, R., Walfish, S.: Universally composable security with global setup. In: TCC. Lecture Notes in Computer Science, vol. 4392, pp. 61–85. Springer (2007)
10. Canetti, R., Hogan, K., Malhotra, A., Varia, M.: A universally composable treatment of network time. In: CSF. pp. 360–375. IEEE Computer Society (2017)
11. Chandran, N., Chongchitmate, W., Garay, J.A., Goldwasser, S., Ostrovsky, R., Zikas, V.: The hidden graph model: Communication locality and optimal resiliency with adaptive faults. In: Roughgarden, T. (ed.) ITCS 2015. pp. 153–162. ACM, Rehovot, Israel (Jan 11–13, 2015). https://doi.org/10.1145/2688073.2688102
12. Coretti, S., Kiayias, A., Moore, C., Russell, A.: The generals' scuttlebutt: Byzantine-resilient gossip protocols. Cryptology ePrint Archive, Report 2022/541 (2022), https://ia.cr/2022/541
13. Crisóstomo, S., Schilcher, U., Bettstetter, C., Barros, J.: Analysis of probabilistic flooding: How do we choose the right coin? In: ICC. pp. 1–6. IEEE (2009)
14. Daian, P., Pass, R., Shi, E.: Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. In: Goldberg, I., Moore, T. (eds.) FC 2019. LNCS, vol. 11598, pp. 23–41. Springer, Heidelberg, Germany, Frigate Bay, St. Kitts and Nevis (Feb 18–22, 2019). https://doi.org/10.1007/978-3-030-32101-7_2

15. David, B., Gazi, P., Kiayias, A., Russell, A.: Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part II. LNCS, vol. 10821, pp. 66–98. Springer, Heidelberg, Germany, Tel Aviv, Israel (Apr 29 – May 3, 2018). https://doi.org/10.1007/978-3-319-78375-8_3
16. David, B., Gazi, P., Kiayias, A., Russell, A.: Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In: EUROCRYPT (2). Lecture Notes in Computer Science, vol. 10821, pp. 66–98. Springer (2018)
17. Demers, A.J., Greene, D.H., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H.E., Swinehart, D.C., Terry, D.B.: Epidemic algorithms for replicated database maintenance. In: Schneider, F.B. (ed.) 6th ACM PODC. pp. 1–12. ACM, Vancouver, BC, Canada (Aug 10–12, 1987). https://doi.org/10.1145/41840.41841
18. Erdős, P., Rényi, A.: On the evolution of random graphs. In: PUBLICATION OF THE MATHEMATICAL INSTITUTE OF THE HUNGARIAN ACADEMY OF SCIENCES. pp. 17–61 (1960)
19. Garay, J.A., Katz, J., Kumaresan, R., Zhou, H.S.: Adaptively secure broadcast, revisited. In: Gavoille, C., Fraigniaud, P. (eds.) 30th ACM PODC. pp. 179–186. ACM, San Jose, CA, USA (Jun 6–8, 2011). https://doi.org/10.1145/1993806.1993832
20. Garay, J.A., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: Analysis and applications. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015, Part II. LNCS, vol. 9057, pp. 281–310. Springer, Heidelberg, Germany, Sofia, Bulgaria (Apr 26–30, 2015). https://doi.org/10.1007/978-3-662-46803-6_10
21. Garay, J.A., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol with chains of variable difficulty. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part I. LNCS, vol. 10401, pp. 291–323. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 20–24, 2017). https://doi.org/10.1007/978-3-319-63688-7_10
22. Haas, Z.J., Halpern, J.Y., Li, L.: Gossip-based ad hoc routing. IEEE/ACM Trans. Netw. **14**(3), 479–491 (2006). https://doi.org/10.1145/1143396.1143399
23. Heilman, E., Kendler, A., Zohar, A., Goldberg, S.: Eclipse attacks on bitcoin's peer-to-peer network. In: Jung, J., Holz, T. (eds.) USENIX Security 2015. pp. 129–144. USENIX Association, Washington, DC, USA (Aug 12–14, 2015)
24. Hu, R., Sopena, J., Arantes, L., Sens, P., Demeure, I.M.: Fair comparison of gossip algorithms over large-scale random topologies. In: SRDS. pp. 331–340. IEEE Computer Society (2012)
25. Karp, R.M., Schindelhauer, C., Shenker, S., Vöcking, B.: Randomized rumor spreading. In: 41st FOCS. pp. 565–574. IEEE Computer Society Press, Redondo Beach, CA, USA (Nov 12–14, 2000). https://doi.org/10.1109/SFCS.2000.892324
26. Katz, J., Maurer, U., Tackmann, B., Zikas, V.: Universally composable synchronous computation. In: TCC. Lecture Notes in Computer Science, vol. 7785, pp. 477–498. Springer (2013)
27. Kermarrec, A., Massoulié, L., Ganesh, A.J.: Probabilistic reliable dissemination in large-scale systems. IEEE Trans. Parallel Distributed Syst. **14**(3), 248–258 (2003)
28. Kiayias, A., Russell, A., David, B., Oliynykov, R.: Ouroboros: A provably secure proof-of-stake blockchain protocol. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part I. LNCS, vol. 10401, pp. 357–388. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 20–24, 2017). https://doi.org/10.1007/978-3-319-63688-7_12
29. Kiayias, A., Zhou, H., Zikas, V.: Fair and robust multi-party computation using a global transaction ledger. In: EUROCRYPT (2). Lecture Notes in Computer Science, vol. 9666, pp. 705–734. Springer (2016)

30. Kokoris-Kogias, E., Jovanovic, P., Gasser, L., Gailly, N., Syta, E., Ford, B.: OmniLedger: A secure, scale-out, decentralized ledger via sharding. In: 2018 IEEE Symposium on Security and Privacy. pp. 583–598. IEEE Computer Society Press, San Francisco, CA, USA (May 21–23, 2018). https://doi.org/10.1109/SP.2018.000-5
31. Liu-Zhang, C.D., Matt, C., Maurer, U., Rito, G., Thomsen, S.E.: Practical provably secure flooding for blockchains. Cryptology ePrint Archive, Paper 2022/608 (2022), https://eprint.iacr.org/2022/608
32. Luu, L., Narayanan, V., Zheng, C., Baweja, K., Gilbert, S., Saxena, P.: A secure sharding protocol for open blockchains. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016. pp. 17–30. ACM Press, Vienna, Austria (Oct 24–28, 2016). https://doi.org/10.1145/2976749.2978389
33. Marcus, Y., Heilman, E., Goldberg, S.: Low-resource eclipse attacks on Ethereum's peer-to-peer network. Cryptology ePrint Archive, Report 2018/236 (2018), https://eprint.iacr.org/2018/236
34. Matt, C., Nielsen, J.B., Thomsen, S.E.: Formalizing delayed adaptive corruptions and the security of flooding networks. Cryptology ePrint Archive, Paper 2022/010 (2022), https://eprint.iacr.org/2022/010
35. Maymounkov, P., Mazières, D.: Kademlia: A peer-to-peer information system based on the XOR metric. In: Druschel, P., Kaashoek, M.F., Rowstron, A.I.T. (eds.) Peer-to-Peer Systems, First International Workshop, IPTPS 2002, Cambridge, MA, USA, March 7-8, 2002, Revised Papers. Lecture Notes in Computer Science, vol. 2429, pp. 53–65. Springer (2002). https://doi.org/10.1007/3-540-45748-8_5
36. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
37. Nielsen, J.B.: On protocol security in the cryptographic model. Ph.D. thesis, Aarhus University (2003)
38. Pass, R., Seeman, L., shelat, a.: Analysis of the blockchain protocol in asynchronous networks. In: Coron, J., Nielsen, J.B. (eds.) EUROCRYPT 2017, Part II. LNCS, vol. 10211, pp. 643–673. Springer, Heidelberg, Germany, Paris, France (Apr 30 – May 4, 2017). https://doi.org/10.1007/978-3-319-56614-6_22
39. Pass, R., Shi, E.: Hybrid consensus: Efficient consensus in the permissionless model. In: Richa, A.W. (ed.) 31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria. LIPIcs, vol. 91, pp. 39:1–39:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.DISC.2017.39
40. Ren, L.: Analysis of Nakamoto consensus. Cryptology ePrint Archive, Report 2019/943 (2019), https://eprint.iacr.org/2019/943
41. Rohrer, E., Tschorsch, F.: Kadcast: A structured approach to broadcast in blockchain networks. In: Proceedings of the 1st ACM Conference on Advances in Financial Technologies, AFT 2019, Zurich, Switzerland, October 21-23, 2019. pp. 199–213. ACM (2019). https://doi.org/10.1145/3318041.3355469
42. Zamani, M., Movahedi, M., Raykova, M.: RapidChain: Scaling blockchain via full sharding. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018. pp. 931–948. ACM Press, Toronto, ON, Canada (Oct 15–19, 2018). https://doi.org/10.1145/3243734.3243853